# MCNelectron

Open-source Monte Carlo code for simulation of coupled electron-photon transport

**v1.2.6**

User‘s Manual

by Andrius Poškus

(Vilnius University, Department of Solid State Electronics)

2022-01-25

E-mail: andrius.poskus@ff.vu.lt or andrius.poskus@live.com
Web: http://web.vu.lt/ff/a.poskus/mcnelectron/

# Contents

## 1. Introduction

MCNelectron is an open-source single-event Monte Carlo code for simulation of coupled electron-photon transport. MCNelectron was written by Andrius Poškus (Vilnius University, Faculty of Physics, Department of Solid State Electronics). MCNelectron uses a public-domain Fibonacci series random number generator by George Marsaglia and Arif Zaman (downloaded from www.netlib.org). MCNelectron is distributed under the GNU General Public License.

MCNelectron can be downloaded from http://web.vu.lt/ff/a.poskus/mcnelectron/.

### *1.1. Comparison with MCNP6*

The physics models implemented in MCNelectron are essentially the same as those applied by MCNP6 [1]. MCNelectron implements constructive solid geometry on regions bounded by planes, spheres, circular cylinders and circular one-sheet cones. In this respect, MCNelectron is more limited than MCNP and other significant general-purpose Monte Carlo codes, which can model general quadric surfaces and tori. The maximum number of bounding surfaces per one cell in MCNelectron is 5000, which is sufficient for modeling complex and irregular objects with a large number of faces, edges and vertices. Specification of complex geometries may be simplified using the union and complement operators, coordinate transformations and macrobodies (as in MCNP). In addition, MCNelectron includes a built-in compiler/interpreter of the user's code embedded in the input file for procedural generation of input directives. Before starting the simulation, MCNelectron checks the geometry specification of each cell for consistency and locates intersecting cells, if they exist (the initial checking for geometry errors is more comprehensive than in MCNP).

The source of particles is defined by specifying its position, direction and energy separately and independently. There are three types of spatial distribution (point source, radially symmetric distribution on a plane with Gaussian fall-off, and uniform distribution inside an arbitrary cell) and three types of angular distribution (parallel beam, uniform distribution inside a cone, and isotropic source). The choice of source geometries is significantly more limited than in MCNP, but adequate for most applications. The source energy distribution is defined by its histogram data (energy tally).

The tallying capability of MCNelectron is much more limited than that of MCNP. MCNelectron has three types of tallies: plane-crossing tally, cell-entry tally (with the corresponding energy transfer tallies) and pulse-height tally (with the corresponding energy-deposition tally). A cell-entry tally can be used to calculate energy distribution of particles crossing any subset of faces of any cell. It can also be used to calculate a tally of particles crossing any single surface (this would require defining a cell with a single bounding surface). A plane-crossing tally is limited to planar surfaces, but is more flexible in other respects: it allows using other control variables besides energy (such as angle of incidence or coordinate on a plane), calculating "two-dimensional" tallies, which have two independent control variables, and defining a set of parallel equidistant "tallying planes". All tallies have equidistant bins. In addition to the tallies, MCNelectron outputs the following overall statistics both for the entire system and for individual cells:

- total numbers of various types of interaction events,
- total numbers of secondary electrons and photons created in various types of interactions,
- total energies of secondary electrons and photons created in various types of interactions,
- total energy losses of electrons and photons due to various types of interactions,
- the average energy loss per one secondary electron.

Only publicly-available information about physics models and simulation methods implemented in MCNP was used when writing MCNelectron. The author of MCNelectron has never had access to source code of any version of MCNP or any other Monte Carlo simulation software. MCNelectron has been written in order to circumvent some of the limitations of MCNP6:

- MCNP6 is much slower in single-event mode than in condensed-history mode when simulating electron transport (see also Section 1.2). The single-event electron transport in MCNP6 is a relatively new feature (it was introduced in 2013, whereas the condensed-history method for electron transport was implemented as early as in 1990s, starting with MCNP4). MCNelectron

has been written with emphasis on computational efficiency of single-event simulations and outperforms MCNP6 under identical simulation conditions in single-event mode;

- MCNP6 does CPU-only computations, whereas MCNelectron can perform so-called "hybrid" computations, when both the central processing unit (CPU) and Nvidia graphics processing units (GPU) are employed in parallel. This is achieved using CUDA (see Section 1.3 for details);
- Simulation of low-energy (less than 1 keV) electron and photon transport in MCNP6 is done using only ENDF/B library data (MCNelectron allows replacing ENDF/B cross sections of some types of electron interactions by alternative cross sections from other libraries);
- MCNP6 does not allow setting the low-energy cutoff limit for electrons to values less than 10 eV (MCNelectron allows that);
- MCNP6 outputs only the statistics pertaining to *tracked* electrons and photons. Hence, if initial energy of a knock-on electron is less than the cutoff energy, that electron will not be counted in the total number of knock-on electrons that is reported in the MCNP output file. MCNelectron counts the number of ionization *events*, regardless of the knock-on electron energy. This allows to take into account the knock-on electrons with energies less than the cutoff energy;
- MCNP6 does not allow turning off elastic scattering of electrons (MCNelectron allows that);
- MCNP6 does not allow turning off tracking of particle coordinates (MCNelectron allows that).

The latter two options are useful for speeding up the simulation when calculating the average energy loss per one secondary electron due to complete absorption of incident particle energy in an infinite medium (in such a case, coherent scattering of photons should be turned off, too). However, even in the case of the maximum complexity of simulation, i.e., when all possible types of interactions are "turned on" and particle coordinates are tracked, a CPU-only simulation done with MCNelectron is shorter than a simulation done with MCNP6 in single-event mode. Under identical simulation conditions, the time of a CPU-only simulation with MCNelectron v1.2.6 (built in the "Release" configuration of Microsoft Visual Studio 2010) is usually less than half of the simulation time with MCNP6.1.1 in single-event mode, depending on the simulated system. The simulation time can be additionally reduced using interaction forcing (see Section 4.8).

By default, the data used by MCNelectron are taken from the ENDF/B-VII.1 library, except for cross sections of inner-shell electroionization (they were calculated according to the distorted-wave Born approximation), cross sections of electron elastic scattering and corresponding angular distributions (they were obtained from relativistic (Dirac) partial-wave calculations), and cross sections of positron bremsstrahlung (they reflect the effect of partial suppression of positron bremsstrahlung in comparison with electron bremsstrahlung, which is not taken into account by the ENDF/B data). Those replacements of ENDF/B data are "on" by default, because they improve the accuracy of the simulation results. However, the user has an option to revert to the ENDF/B data when simulating each of the mentioned interactions. In addition, it is possible to replace the ENDF/B low-energy electroionization and excitation cross sections by third-party cross sections provided by the user (with optional interpolation between those low-energy alternative cross sections and high-energy ENDF/B cross sections). The data that are always taken from the ENDF/B library (i.e., cannot be replaced) include electron bremsstrahlung cross sections, excitation energy transfer, all photon interaction cross sections and atomic relaxation data.

### *1.2. Single-event method vs. condensed-history approximations*

In most available general-purpose Monte Carlo codes, electron transport is simulated using a combination of two methods – single-event (SE) and condensed-history (CH), with a possibility to choose between the two methods via a user-adjustable parameter. The SE method simulates individual electron collisions separately, whereas the CH approximation is based on breaking the electron's path into many steps, chosen long enough to encompass many collisions, which are treated by multiple-scattering theories. The need for the CH approximation arises from the fact that the average energy loss in one ionizing collision of an electron with energy much higher than the

ionization threshold is of the order of 10 eV or less in most materials, i.e., relatively small. Consequently, as it follows from the law of conservation of momentum, the average angular deflection of the incident electron is also relatively small. Hence an electron with energy of the order of 10 keV experiences several thousand inelastic collisions with small-angle deflections, and an even greater number of elastic collisions before coming to a halt. As electron energy is increased, the number of collisions increases proportionally. This large number of collisions can make the detailed event-by-event simulation prohibitively long when simulating interaction of high-energy electrons with thick targets. The CH approximations make it possible to shorten the simulation significantly while preserving the essential physical features of the simulated process.

In comparison with the condensed-history approximations, the single-event method is conceptually much simpler. In the case of the SE method, motion of a particle is simulated by repeating those four steps:

1) calculate the total cross section and use it to calculate distance to collision,
2) select target (if there are several types of atoms in the material),
3) select the type of interaction,
4) calculate energies and directions of the primary and secondary particles after the interaction.

This scheme is always applied to photons, because photon transport is faster than electron transport, i.e., the number of photon collisions is typically much less than the number of electron collisions.

Since the SE method is less dependent on implementation-specific theoretical assumptions than the CH approximation, the SE method is potentially more accurate. Its accuracy is mainly determined by accuracy of cross sections used for the simulation.

MCNelectron simulates coupled electron-photon transport using only the single-event method (both for electrons and for photons), except at extremely low electron energies, when the diffusion model of electron elastic scattering may be optionally applied.

### *1.3. CUDA support in MCNelectron*

There are two versions of MCNelectron: the CPU-only version (the names of the 32-bit and 64-bit executable files are MCNelectron.exe and MCNelectron_x64.exe, respectively) and the version with CUDA support (MCNelectron_CUDA.exe and MCNelectron_CUDA_x64.exe). The CUDA version of MCNelectron can perform so-called “hybrid” computations, when both the central processing unit (CPU) and Nvidia graphics processing units (GPU) are employed in parallel. The CUDA version of MCNelectron can also perform GPU-only or CPU-only computations (in CPU-only mode, MCNelectron_CUDA.exe produces the same results as MCNelectron.exe).

CUDA (Compute Unified Device Architecture) is a parallel computing platform and application programming model created by Nvidia Corporation [2]. It allows using Nvidia video cards for general-purpose processing. The GPUs used for the simulation need not to be dedicated for general-purpose computations: the video card that outputs images to a computer monitor may be used at the same time for the simulation. MCNelectron implements automatic workload balancing between the CPU and available CUDA devices during the simulation, so that the CPU and the GPU (or several GPUs, if there are two or more Nvidia video cards in the system) spend approximately equal times for the simulation. The GPUs must have CUDA compute capability 2.0 or higher. The improvement of performance when GPUs are used together with the CPU in comparison with a CPU-only computation depends on a particular CPU/GPU combination present in the system. For example, the simulation time on a single Nvidia GeForce GTX 780 Ti video card is about 1/3 of the time required for the same simulation using a single thread on an Intel Core i7-4930K processor. However, from a practical point of view it is more meaningful to compare the simulation times when the CPU is at full load, i.e., when all cores of the CPU are used in parallel, because in practice one is interested in utilizing all available computing power as fully as possible. Since the Intel Core i7-4930K processor has 6 cores and uses hyper-threading, the maximum CPU load is achieved with 12 or more threads, and then the CPU is approximately 8 times faster than a single CPU thread

(performance does not scale linearly with the number of CPU threads). Hence, an improvement in processing speed in hybrid computations is about 3 / 8 = 37 % when a single GTX 780 Ti video card is used together with the CPU, and about 70 % when two GTX 780 Ti video cards are used. This is evident in Table 1, which compares simulation times using an Intel Core i7-4930K processor and two Nvidia GeForce GTX 780 Ti video cards. In this case, the simulation setup consisted of $5 \cdot 10^7$ electrons with energy 15 keV, incident normally on a thick layer of germanium. The electron and photon cutoff energy was 1 keV. The operating system was 64-bit Windows 8.1.

**Table 1**. Comparison of simulation times

| Program | Number of CPU threads | Mode | Number of GPUs used | Time (min) |
|---|---|---|---|---|
| MCNP6.1 | 16 | CH | 0 | 15.4 |
| MCNP6.1 | 16 | SE | 0 | 52.3 |
| MCNelectron v.1.2.0 | 16 | SE | 0 | 20.0 |
| MCNelectron v.1.2.0 | 11 | SE | 1 | 15.7 |
| MCNelectron v.1.2.0 | 11 | SE | 2 | 13.1 |

The performance gains caused by using CUDA are strongly dependent on the amount of branching in the code that is executed by the GPUs. The GPU architecture can be defined as "Single Instruction, Multiple Data streams" (SIMD), which means that in order to achieve maximum performance a majority of the CUDA threads should always perform the same instruction simultaneously, albeit on different data. This is in contrast with the "Single Instruction, Single Data stream" (SISD) architecture implemented on each core of a multi-core CPU. Because of the SISD architecture, the different cores of a CPU are completely independent and branching does not affect the performance gains caused by using multiple CPU cores simultaneously. However, the code that is executed by CUDA devices must be structured quite differently. CUDA groups all GPU threads into so-called "warps", each consisting of 32 threads. To quote "CUDA C Programming Guide" [2]: "If threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path." Thus, all threads of each warp should perform the same sequence of instructions in order to minimize branching (also called "warp divergence"). In the case of MCNelectron, warp divergence is reduced by breaking each particle history into "states" (e.g., one state is generation of a source particle, another one is calculation of cross section, yet another one is calculation of the distance to collision, etc.), assigning each warp a particular state, and ensuring that at every moment of time the number of particles with identical states is sufficient to completely "populate" each warp (i.e., one particle of a given state per one GPU thread of the corresponding warp). Each thread of a warp then invokes the corresponding "state handler". Since the state is the same for all threads in a warp, their state handlers perform roughly the same sequence of instructions, hence branching is reduced. The new state after executing a state handler may be different for different threads of a given warp. This necessitates re-assigning states to warps and re-distributing the particles among warps in the next iteration. This step is done serially, hence it limits the maximum performance that can be achieved. In MCNelectron v1.2.6, the mentioned re-distribution step usually takes at least 15 % of entire processing time.

## 2. Cross section data files

The MCNelectron distribution package includes ENDF/B cross section data for 100 chemical elements with atomic numbers from 1 to 100. They are in the subfolder "Data\ENDF". Those data were downloaded in 2018 from the National Nuclear Data Center (address of the download web page: http://www.nndc.bnl.gov/endf/b7.1/download.html), using the following three links:

http://www.nndc.bnl.gov/endf/b7.1/zips/ENDF-B-VII.1-electrons.zip,
http://www.nndc.bnl.gov/endf/b7.1/zips/ENDF-B-VII.1-photoat.zip,
http://www.nndc.bnl.gov/endf/b7.1/zips/ENDF-B-VII.1-atomic_relax.zip.

Each of those files contains one of the three needed sublibraries of the ENDF/B-VII.1 library:

- Electro-atomic
- Photo-atomic
- Atomic relaxation

There must be three files with ENDF data for each of the chemical elements that compose the simulation setup:

- a file with electron interaction cross sections,
- a file with photon interaction cross sections,
- a file with atomic relaxation data.

Those files must be in three subfolders of the folder specified in the input file or on the command line after the keyword DIR (for a complete list of keywords, see Sections 4.1 – 4.4, 4.6 – 4.10). The names of those subfolders are fixed: they must be "electrons", "photoat" and "atomic_relax", respectively. Format of the names of the mentioned files is also fixed. That format becomes clear after looking at those three names of files containing the ENDF data for argon:

e-018_Ar_000.endf
photoat-018_Ar_000.endf
atom-018_Ar_000.endf

(the number after the hyphen is the atomic number of the chemical element). For example, if the ENDF data are stored in the folder C:\ENDF\, then the full path names corresponding to the above-mentioned three files with the data for argon would be the following:

C:\ENDF\electrons\e-018_Ar_000.endf
C:\ENDF\photoat\photoat-018_Ar_000.endf
C:\ENDF\atomic_relax\atom-018_Ar_000.endf

***Note***: If the ENDF/B data were downloaded using the three links to electro-atomic, photo-atomic and atomic relaxation data given above, then it is sufficient to place those files into a chosen folder and unpack them (the names of extracted files and subfolders will be as described above). If the ENDF/B data were downloaded from other locations, then the file names may be different.

The files with alternative low-energy electron interaction cross sections must have the same format as the files posted on the LXCat website (www.lxcat.net). Those files may have any names and may be stored in any location, because MCNelectron requires specification of full paths to those files for each of the chemical elements separately. The alternative low-energy electroionization and atomic excitation cross sections must be in different files. Thus, if alternative cross sections are used, there must be two additional files for each of the chemical elements: a file with alternative electroionization cross sections and a file with alternative excitation cross sections. There may be several electroionization or excitation processes defined in each of those files (the file format must be the same as format of files posted on www.lxcat.net). Instructions for retrieving cross sections from www.lxcat.net :

1) Select menu "DATA CENTER", then select the menu item "browse and download".
2) In the menu on the right, select "SCATTERING CROSS SECTION" and click "NEXT".
3) In the menu on the right, select or unselect the necessary databases and click "NEXT".
4) Select the necessary chemical element and click "NEXT".
5) Select either "Excitation" or "Ionization" and click "NEXT".
6) Ensure that all excitation or ionization processes are selected and click "NEXT".
7) Left-click the button "DOWNLOAD DATA" and save the page, or right-click that button and use the context menu item "Save Link As…". The full path of the created file will have to be specified in the file with alternative cross sections information (its format is described in Section 4.9).

***Note***: Alternatively, data can be retrieved from www.lxcat.net by selecting the menu item "download for offline use" in Step 1. However, then the downloaded file will contain data for all interaction processes. Since MCNelectron requires the excitation and ionization cross sections to be in separate files, the file retrieved by the latter method will have to be edited with a text editor.

## 3. Compiling MCNelectron

MCNelectron is a Windows application (the oldest supported version of Windows is Windows XP SP2; all newer versions of Windows are supported). There is a separate distribution package (a WinRAR self-extracting archive) for the CPU-only version and the CUDA version (the self-extracting archive with the CUDA version is split in three parts). Each of those two packages contains the C++ source files, the final executable files ("MCNelectron.exe" or "MCNelectron_CUDA.exe", respectively, and their 64-bit counterparts, whose names end with "_x64.exe") and a Microsoft Visual Studio 2010 project for compiling and linking MCNelectron. The mentioned two file names correspond to the executables built using the "Release" configuration of Microsoft Visual Studio 2010. The names of the executables built using the "Debug" configuration are formed by appending "_debug" to the name of the corresponding release executable (for example, "MCNelectron_debug.exe"). The simulation time with the "debug" executables is longer by a factor of 2 to 3 than with the "release" ones (the "debug" executables are not included in the distribution packages). The following source files are needed for compiling the CPU-only version of MCNelectron:

MCNelectron.cpp,
RandGen.cpp,
RandGen.h,
get_executable_name.cpp,
29 files in the subfolder "Compiler".

The following source files are needed for compiling the CUDA version of MCNelectron:

MCNelectron_CUDA.cu,
MCNelectron_CUDA2.cu,
MCNelectron_CUDA.h,
RandGen.cu,
RandGen.h,
RandGen_CUDA.cu,
RandGen_CUDA.h,
get_executable_name.cpp,
29 files in the subfolder "Compiler".

In order to be able to build MCNelectron_CUDA, the CUDA Toolkit must be installed. The current version of the CUDA Toolkit can be downloaded from https://developer.nvidia.com/cuda-downloads. The older versions are available from https://developer.nvidia.com/cuda-toolkit-archive. The release version of MCNelectron_CUDA (both 32- and 64-bit) must be built on 64-bit Windows 7 or later (the debug version of MCNelectron_CUDA may be built both on 32-bit and 64-bit Windows). At least 24 GB of available physical memory are recommended in order to achieve the shortest build time of the release version of MCNelectron_CUDA, if a CUDA Toolkit version prior to v8.0 is used. In the case of CUDA Toolkit 8.0, even this amount of memory seems to be insufficient: at the time of this writing all attempts to build the release version of MCNelectron_CUDA with CUDA Toolkit 8.0 have failed with the compiler error message "ptxas fatal : Memory allocation failure". The debug version of MCNelectron_CUDA can be built with 2 GB or less available memory.

The executables that are included in the distribution packages were built using the stack size of 10 MB (the default stack size of 1 MB is insufficient).

MCNelectron_CUDA v1.2.6 was built using CUDA Toolkit 7.5. In order to build the project using a different version of the CUDA Toolkit, the following changes should be made in the file "MCNelectron_CUDA.vcxproj" before opening the project: the number "7.5" in "CUDA

7.5.props" and "CUDA 7.5.targets" must be replaced by the required version number of the CUDA Toolkit. "MCNelectron_CUDA.vcxproj" is an ASCII file, hence it can be edited using the Notepad.

The Microsoft Visual Studio 2010 project for building 64-bit MCNelectron_CUDA produces an executable file that will run on CUDA devices with compute capabilities 2.0, 3.0, 3.5, 3.7, and 5.0. Because of this wide range of target architectures, the build time is more than twice longer and the executable file is several times larger than in the case of a single compute capability. The build time of the release version of 64-bit MCNelectron_CUDA v1.2.6 on a computer with Intel Core i7-4930K processor and 32 GB of RAM running 64-bit Windows 10 was approximately 6 hours. The set of compute capabilities of the release version of 32-bit MCNelectron_CUDA is 2.0 and 3.5, and building it on the same computer required approximately 3 hours. The amount of memory used for building MCNelectron_CUDA was greater than 28 GB in both cases. If the code is intended to be used on a particular type of CUDA devices, then the build time may be shortened and the size of the executable file may be decreased by specifying only the compute capability of that device in CUDA C compiler options. This is done by selecting the Visual Studio menu command "Project / MCNelectron_CUDA Properties". When this command is selected, the dialog window with the project property pages is opened. Then, using the left-hand pane of that dialog window, the page "Configuration Properties / CUDA C/C++ / Device" must be activated. The target architectures are entered in the cell "Code Generation". The above-mentioned set of five target architectures is specified by the following line:

```
compute_20,sm_20;compute_30,sm_30;compute_35,sm_35;compute_37,sm_37;compute_50,sm_50;
```

In order to produce a code for CUDA devices with a specific compute capability, e.g., 3.5, that line must be replaced by

```
compute_35,sm_35;
```

An example of a CUDA device with compute capability 3.5 is a Nvidia GeForce GTX 780 video card. An example of a device with compute capability 2.0 is a Nvidia GeForce GTX 580 video card.

## 4. Running Monte Carlo simulations with MCNelectron

MCNelectron is a Windows console application (the Windows console is opened by running the Windows command-line interpreter "cmd.exe"). In the simplest case, the command line is

```
MCNelectron.exe  in <input_file_name>  out <output_file_name>
```

(if the CUDA version is used, "MCNelectron" must be replaced by "MCNelectron_CUDA", and if the 64-bit version is used, then the characters "_x64" must be inserted before ".exe"). The command-line keywords "in" and "out" may be replaced with "in=" and "out=" (to make the MCNelectron command line more similar to the MCNP command line). Note that there must be no space before the equality sign, but there must be a space after it.

In addition, it is possible to specify the name of the file with the alternative cross sections information after the command-line keyword "alt". If the name of the "alt" file has not been specified, then only the ENDF/B cross section data files and the data files contained in the MCNelectron distribution package (subfolder "Data") will be used. If the name of the "alt" file has been specified, then a part of low-energy electron interaction cross sections will be read from files with alternative cross sections supplied by the user. The file with alternative cross sections information contains the names of files where those cross sections are stored and the interpolation information (for more information about the format of this file, see Section 4.9).

Yet another filename that may be specified on the command line is the name of the so-called "log file", which contains all input directives used for the simulation. Its name must be specified after the keyword "log". In order for the specified log file to be created, the command-line option "writeLog 1" must also be specified. By default, the log file is not created (this is equivalent to the command-line option "writeLog 0"). Creating the log file makes sense only when some of the input directives are generated by the user's code embedded in the input file, because otherwise the log file would be an exact copy of the input file (contents of the input file are included in the output file, too). Consequently, usage of the log file is discussed in Section 4.12, which deals with procedural generation of MCNelectron input directives using the MCNEcode programming language.

The names of the output file, log file and the file with alternative cross sections information may be specified in the input file, too (see Section 4.1). Then the name of the input file is the only filename that must be specified on the command line. If the name of the file is specified both on the command line and in the input file, then priority is given to the name specified on the command line (in such case, a corresponding message is displayed). The names of the output file and the log file may be omitted altogether. In such a case, those files would be assigned the default names. The default name of the output file is obtained by appending the suffix "_out" to the input file name. The default name of the log file is obtained by appending the suffix "_log" to the input file name.

The command-line keyword "check" may be used to make MCNelectron quit after checking the simulation setup for errors (without running the Monte Carlo simulation), or for preventing all checking of input directives (in the latter case, MCNelectron would also quit after processing the input file). The first mentioned command-line option is selected by specifying "check -1", and the second mentioned command-line option is selected by specifying "check 0". The default selection is "check 1", which indicates that MCNelectron must check the simulation setup for errors and then run the simulation if there are no errors. Specifying the command-line option "check 0" makes sense only when MCNelectron is used for running the user's code embedded in the input file, rather than for the original purpose of MCNelectron (Monte Carlo simulation of coupled electron-photon transport). In the case "check 0", the names of the output file and the log file cannot be specified in the input file (they must be specified on the command line, unless the default names are acceptable).

In order to stop the simulation and output the current values of all required statistics, the 'Esc' and 'Q' keys must be pressed (in that order). If only 'Esc' was pressed, then it is possible to "cancel" the escape key by pressing any key except 'Esc' and 'Q'. After that, the 'Esc' key would have to be pressed again in order to interrupt the program. The simulation can also be "killed" by pressing the keys "Ctrl+C" (in that case, no data would be output).

By default, MCNelectron runs in single-thread mode on the CPU only (i.e., without using CUDA). In order to use multiple concurrent threads during the simulation, their number must be specified on the command line by inserting the keyword "tasks", followed by the number of threads, for example:

```
MCNelectron.exe  tasks 4  in input.txt  out output.txt
```

In the latter example, four independent threads would be used on the CPU during the simulation. On multi-processor (or multi-core) systems, multithreading can reduce the simulation time significantly. The number of CPU threads can be assigned any value that does not exceed the number of streams of random numbers (see the description of the keyword NSTREAMS in Section 4.1). However, there is no point in using a number of CPU threads that is greater than the number of logical processors in the system (a further increase of the number of threads will not cause a decrease of the simulation time). The final results of the simulation do not depend on the number of threads that was used during the simulation. In the case of the CUDA version of MCNelectron, the number of GPU threads used by each available CUDA device can also be specified on the command line using the keyword "threads_CUDA", e.g., "threads_CUDA 960" (for more information about CUDA-specific keywords, see Section 4.10).

Another optional keyword that may be inserted in the command line is "output_cs 1" or "output_cs 0". If "output_cs 1" is used, then files with values of electron interaction cross sections are created before starting the simulation. In such a case, MCNelectron creates a folder "Cross sections" in the current folder, with subfolders for each of the chemical elements. Each of those subfolders contains the following text files:

- a file with values of elastic, bremsstrahlung, excitation, total ionization, total inelastic and total interaction cross sections corresponding to the energy interval that is used during the simulation,
- files with values of ionization cross section for each electronic subshell and for all available energies of the incident electron (including the energies that are not needed for the simulation),
- a file with values of the energy loss during atomic excitation for all available energies of the incident electron.

Each of those files contains two or more columns of numbers. The first column contains electron energy values (eV). The other columns contain cross section values (b) or energy loss values (eV). If the options "ELASTIC_DPW 0" and "ELASTIC_SMALL_ANGLE <n>" with non-zero "<n>" have been specified in the input file, then the first of the mentioned files contains an additional column with the values of the large-angle elastic scattering cross section (corresponding to the values of the cosine of the scattering angle from –1 to 0.999999).

It is possible to override the specification of the source energy distribution defined in the input file by using the command-line keyword "E", followed by an energy value, (e.g., "E 10"). In such a case, a source of monoenergetic particles will be simulated. The number of incident source particles specified in the input file may be overridden on the command line using the keyword "N".

During the simulation, the CPU-only version of MCNelectron displays the time elapsed since the start of simulation (in seconds), the current number of finished histories, as well as the sequence number of the last source particle and the current number of banked particles for each of the first three or five active threads (those numbers are updated every second). In addition, if "control tallies" are used (see Section 4.7.4), the corresponding largest error is shown. MCNelectron_CUDA displays some real-time statistical information about the CUDA devices used for the simulation (for more information about the CUDA device statistics, see Section 4.11). At the end of the simulation, a part of the contents of the output file is reproduced in the console window. The output format is similar to format of a section of a MCNP output file. All statistics under "electrons" include the contribution from positrons, too. To avoid text wrapping (which would reduce readability of the output data on the screen), the screen buffer width must be at least 170 (this can be achieved, e.g., by entering "mode 170" at the command prompt in the console window before starting the simulation).

The current release version of MCNelectron_CUDA v1.2.6 has been tested on CUDA devices with compute capabilities 2.0, 3.5 and 5.0. All tests were successful, except when the amount of memory on the video card was not sufficient for storing all the data. As a rule of thumb, the amount of memory allocated by MCNelectron_CUDA (which is displayed at the start of the simulation) should be less than half of available global memory on the CUDA device.

### *4.1. Input file format (keywords that are not related to geometry and materials)*

Input file example:

```
MAT   54 0.1  18 0.2  2 0.7
CONC -1
THICK 1e-5
E_UNIT MeV
PART E
CUT_E 1e-5
CUT_P 1e-6
TRACKPOS 0
DIR C:\ENDF\
N 1000
1
```

Each non-empty line of the input file, excluding the comment lines and the lines with the definition of the source energy spectrum (the final line in the above example), starts with a keyword. Each keyword indicates the beginning of an "input directive". Keywords are case-insensitive. Single-line comments in the input file are specified by the double slash "//" (the part of the line after the double slash is ignored by the program). The line-continuation character is '&' (if this character is encountered in the input file, the remaining part of the current directive is read from the next line). No more than 1000 characters are read from each line. Empty lines are ignored.

Below are descriptions of the keywords that are common to both the CPU-only and CUDA versions of MCNelectron, excluding the keywords used for specifying the geometry and the materials of the simulation setup (they will be described in Sections 4.2 – 4.4):

E_UNIT specifies the unit of energy that is used in the input and output files. It must be one of three units: “MeV”, “keV”, or “eV”.

CUT_E is used to specify the low-energy cutoff value for electrons.

CUT_P is used to specify the low-energy cutoff value for photons.

PART should be followed by the letter ‘E’ if source particles are electrons, ‘P’ for photons, or “E+” for positrons.

TASKS is used to specify the number of concurrent simulation threads on the CPU (for example, “TASKS 4”). I.e., this keyword has the same function as the mentioned command-line keyword “tasks”. If the keyword “TASKS” is specified both on the command line and in the input file, then priority is given to the number of threads specified on the command line (in such a case, a corresponding message is displayed in the console window). ***Note***: If CUDA devices are used, then one CPU thread is dedicated to communication with them and is not used for the actual simulation of particle interactions. This thread is not included in the number of threads specified after the keyword TASKS.

The following 19 keywords are used as “switches”, which “turn on” or “turn off” certain features of the simulation. A non-zero value of the integer number that follows each of those keywords means that the feature is “turned on”, and the zero value means that it is “turned off”. Those features are:

TRACKPOS – tracking of particle coordinates (i.e., position and direction of motion),

COH – coherent scattering of photons (if TRACKPOS is 0, then the option “COH 1” is applied to source photons only),

ELASTIC – elastic scattering of electrons and positrons,

TRACK_E – tracking of secondary electrons and positrons,

TRACK_P – tracking of secondary photons,

TRACK_P_E – tracking of photon-produced electrons and positrons,

TRACK_K – tracking of knock-on electrons,

TRACK_B – tracking of bremsstrahlung photons,

TRACK_X – tracking of X-ray photons (both fluorescence and electron X-rays),

ION_DWBA – replacement of the ENDF/B inner-shell electron impact ionization cross sections by those calculated using the distorted-wave Born approximation (DWBA). The latter cross sections are the same ones that are used by the PENELOPE code system. They were calculated using the code by D. Bote, F. Salvat, A. Jablonski, and C. J. Powell, which was published in 2009 [3]. Those cross sections are stored in the file “Data\IonDWBA.dat” in binary format. ***Note***: The option “ION_DWBA 1” is not allowed when a file with alternative cross sections information is used (command-line keyword “alt”).

ELASTIC_DPW – replacement of the ENDF/B electron elastic scattering cross sections by those obtained from relativistic (Dirac) partial-wave calculations. Those cross sections were calculated using the code ELSEPA by F. Salvat, A. Jablonski and C. J. Powell [4] (the same cross sections are used by the PENELOPE code system). They are stored in binary files “Data\ElasticDPW_totalCS.dat” and “Data\ElasticDPW_angularDistr.dat”. The latter file contains angular distributions calculated for electron energies from 10 eV to 1 GeV on a logarithmic scale (the increment of the base-10 logarithm of energy is equal to 0.1).

BREMS_ANGULAR_DISTR – non-isotropic angular distribution of bremsstrahlung,

BREMS_BINARY_DATA – using binary bremsstrahlung angular distribution data (otherwise, the bremsstrahlung data are loaded from text files specified by the user),

BREMS_POSITRON_CORRECTION – partial suppression of positron bremsstrahlung in comparison with electron bremsstrahlung,

PE_ANGULAR_DISTR – non-isotropic angular distribution of photoelectrons,

PP_ENERGY_DISTR – non-uniform distribution of positron and electron energies during pair production,

PP_ANGULAR_DISTR – angular distribution of electrons and positrons created in pair production events,

INCOH_DOPPLER – Doppler broadening of energy distribution of incoherently scattered photons,

INCOH_SUBTRACT_BINDING_E – apply the binding energy correction to the energy of the Compton recoil electron (i.e., subtract the binding energy of the subshell from which the electron was ejected).

When TRACKPOS is 0 and the source emits photons, the geometrical parameters (defined using the keywords THICK, SURFACE and BEAM) are only used to determine if a source photon collided with an atom of the material. All subsequent interactions in the same history are treated as though the layer was infinite in all directions. If the source emits electrons or positrons, then in the case of TRACKPOS equal to 0 the geometrical parameters are not used at all. The option "TRACKPOS 0" should be used to calculate the number of secondary electrons when energy of the source particle is completely absorbed in the target material. For keywords COH, ELASTIC, TRACK_P_E, TRACK_K, TRACK_B and TRACK_X, a negative value has a special meaning: it means that the corresponding switch should be set to a default value (the latter may depend on values of other switches). The last eight "switches" (from "BREMS_ANGULAR_DISTR" to "INCOH_SUBTRACT_BINDING_E") control simulation of several physical effects, for which the ENDF/B-VII.1 library provides neither tabular probabilities nor an analytic prescription. For some of those effects, there are additional keywords controlling some details of the simulation. Those keywords and the corresponding simulation methods are described in Section 5 of this document.

SEED is used to specify the seed of the "type 1" random number generator (this is the mentioned Fibonacci series random number generator; the "type 2" random number generator is the XORWOW generator from the cuRAND library, which can be used only by CUDA devices). The seed may be any integer number from 0 to 30081 inclusive. The seed may be also specified on the command line (similarly to the above-mentioned keyword TASKS).

DIR is used to specify the folder with ENDF data. If that folder name does not contain the full path (i.e., if it does not start with the backslash '\' and does not contain the colon ':'), then it is assumed to be a subfolder of the folder where the MCNelectron executable is located. The ENDF folder may be also specified on the command line.

OUT is used to specify the name of the output file. If that name does not contain the full path (i.e., if it does not start with the backslash '\' and does not contain the colon ':'), then it is assumed to be in the current folder (which may be different from the folder where the MCNelectron executable is located). The format of the output file is described in Section 6.

LOG is used to specify the name of the log file. If that name does not contain the full path (i.e., if it does not start with the backslash '\' and does not contain the colon ':'), then it is assumed to be in the current folder (which may be different from the folder where the MCNelectron executable is located). If the name of the log file is specified in the input file, then the "LOG" directive must precede all other input directives (otherwise the program would quit with an error message). For more information about using the log file, see Section 4.12.

ALT is used to specify the name of the file with alternative cross sections information. If that name does not contain the full path (i.e., if it does not start with the backslash '\' and does not contain the colon ':'), then it is assumed to be in the current folder or in a subfolder of the current folder (which may be different from the folder where the MCNelectron executable is located). For more information about using the "alt" file, see Section 4.9.

NSTREAMS is used to specify the number of independent streams of random numbers. For optimum performance, the number of streams should be a multiple of the number of threads. The maximum allowed number of streams is $2^{16} = 65536$. This parameter may be also specified on the command line. ***Note***: The maximum number of unique random number streams that can be generated by the Fibonacci series generator using a given seed (specified after the mentioned keyword "SEED") is 31329. Consequently, the streams with sequence numbers from 31330 to $2 \cdot 31329 = 62658$ will be generated using the next seed (i.e., the seed that exceeds the specified seed by 1), and the streams with sequence numbers from 62659 to 65536 will be generated using the seed that exceeds the specified seed by 2 (if the seed value obtained by adding 1 or 2 is greater than 30081, then it is reduced by 30082). This should be kept in mind when re-doing

a simulation with a different seed: if the number of random number streams is greater than 31329 and the seeds differ only by 1, then some of the streams will be the same in both simulations. As a result, some of the simulated histories will be also exactly the same, which is usually undesirable. In order to ensure that different simulations do not re-use the same random number streams when the total number of streams exceeds 31329, the values of the seed in those simulations should differ at least by 2 or 3.

N is used to specify the number of incident particles. This number includes only the source particles that enter a cell of the geometry. If a source particle does not enter any cell of the geometry during the simulation, then the total number of processed source particles will not be incremented. This parameter may be also specified on the command line.

COH_THR is used to specify the coherent scattering angular sampling threshold value. This value should not be greater than 0.001 or less than 0. A larger value may shorten the simulation when the angular distribution of photon coherent scattering is extremely peaked in the forward direction (i.e., when photon energy is very large). A smaller value leads to more accurate sampling of angular deflections in such a situation. However, a very small (e.g., zero) value of this parameter can make the rejection-based sampling algorithm very inefficient. This means that a very large number of tries may be needed before accepting a sampled value of angular deflection after a coherent scattering event.

SIGMA_ERR controls the relative error introduced by removal of the energies for which the electron cross sections are stored in the cross section tables that are created before starting each simulation. This removal of the data points allows to reduce the time of the energy lookup and to reduce the amount of computer memory used for the simulation. The consecutive energy values are removed until the maximum relative deviation of the linearly interpolated cross sections of all interaction types (including all chemical elements composing the given material) from the original cross sections becomes greater than the number specified after this keyword. After that, the current energy value is designated as the initial energy of the next interval for the linear interpolation, and the process is repeated. The allowed values of the maximum relative error are from 0 to 0.1. If this number is 0, then all available data points will be used.

Keywords NTRACKS, NCOLLISIONS_TRACK, SKIP_TRACKS_WITHOUT_COLLISIONS, MAXGEN_TRACKS and CHUNKSIZE_TRACKS are related to output of particle track data. They are described in Section 4.6.

The keywords starting with “FORCE_” are related to interaction forcing (one of the variance reduction techniques) and are described in Section 4.8.

DIFFUSION_TOTAL_TO_INELASTIC_RATIO_THR controls an option to apply a diffusion model of elastic scattering of electrons at extremely low electron energies, when the elastic scattering cross section is much larger than cross sections of all other types of electron interactions. When the diffusion model is applied, the program first generates the distance to the next inelastic interaction event (i.e., the path traveled to that event, not the displacement), and samples the electron coordinates corresponding to that event from a Gaussian distribution with a root-mean-square displacement proportional to the square root of the generated path. This reduces the “electron trapping” effect (which could otherwise slow the simulation significantly) at the expense of fidelity of the simulated electron trajectory when the electron energy is only a few electronvolts above the ionization threshold of the material, or when the energy is below the ionization threshold. The transition to the diffusion model occurs at a user-specified value of the total-to-inelastic cross section ratio. The keyword “DIFFUSION_TOTAL_TO_INELASTIC_RATIO_THR” is used to specify the mentioned threshold value of that ratio. The allowed values of that threshold are from 10 to $10^{18}$. The diffusion model of elastic scattering is never applied if the distance from the electron to the nearest surface of the cell is less than the mean free path times the square root of this parameter. In order to “turn off” application of the diffusion model, this parameter must be set to any sufficiently large value (e.g., to the maximum possible value 1e18).

DIFFUSION_DIST_TO_RMSPATH_RATIO_THR controls application of the diffusion model of elastic scattering when the electron is close to the surface of the cell. The above-mentioned Gaussian distribution is accurate only when the electron is at a depth $d$ large enough to make the probability that the electron escapes from the cell before the next inelastic collision occurs practically zero. Consequently, the diffusion model of elastic scattering may introduce significant errors when estimating the time and position of electron's escape from the cell if the electron is close to the surface. In order to decrease those errors, a single diffusion step mentioned above is replaced by a sufficiently large number of shorter diffusion steps, which are less than the root-mean-square displacement and also less than $d$ (but still greater than the mean free path) if $d$ is less than a user-specified number of r.m.s. displacements (the latter r.m.s. displacement is estimated as described above). This number is specified after the latter keyword. The allowed values of that number are from 2 to 10.

SMOOTH_CDF_ELASTIC is a "switch" that "turns on" or "turns off" piecewise-linear approximation of the probability density function (PDF) of the cosine of the electron elastic scattering angle that is used for the simulation. If the integer number following this keyword is 1, then the mentioned PDF is continuous (piecewise-linear), whereas the corresponding cumulative distribution function (CDF) is piecewise-quadratic, i.e., "smooth". If the integer number following this keyword is 0, then the PDF is discontinuous and it is represented by a bar graph with the height of each bar equal to the average PDF in the corresponding interval of the argument values (similarly to MCNP6.1 in single-event mode). In this case, the CDF is piecewise-linear. The option "SMOOTH_CDF_ELASTIC 0" makes sampling of the random values from the CDF tables slightly faster (at the expense of a slight reduction in accuracy). If the number following this keyword is −1, then the result will depend on the setting controlled by the previously-described keyword ELASTIC_DPW: in the case "ELASTIC_DPW 1" (the default), the result will be equivalent to "SMOOTH_CDF_ELASTIC 1", and in the case "ELASTIC_DPW 0" the result will be equivalent to "SMOOTH_CDF_ELASTIC 0".

SMOOTH_CDF_INELASTIC is a "switch" that "turns on" or "turns off" piecewise-linear approximation of all probability density functions (PDF) that are specified in tabular format, excluding the PDF of the cosine of the electron elastic scattering angle (see the description of the keyword SMOOTH_CDF_ELASTIC above). It functions similarly to the previously-described keyword SMOOTH_CDF_ELASTIC (however, "SMOOTH_CDF_INELASTIC" cannot be followed by a negative number).

SMOOTH_CDF is a "switch" that functions as a substitute for the two keywords SMOOTH_CDF_ELASTIC and SMOOTH_CDF_INELASTIC described above:
"SMOOTH_CDF 1": same as "SMOOTH_CDF_ELASTIC 1", "SMOOTH_CDF_INELASTIC 1";
"SMOOTH_CDF 0": same as "SMOOTH_CDF_ELASTIC 0", "SMOOTH_CDF_INELASTIC 0";
"SMOOTH_CDF -1": same as "SMOOTH_CDF_ELASTIC -1", "SMOOTH_CDF_INELASTIC 1".

ELASTIC_SMALL_ANGLE controls calculation of the cross section of elastic scattering into the range of the cosine of the scattering angle from 0.999999 to 1, when the ENDF/B elastic scattering data are used ("ELASTIC_DPW 0"). In the case "ELASTIC_DPW 1" (which is the default), this setting is ignored. This keyword must be followed by an integer number equal to 0, 1, 2, or 3. Each of those options is an attempt to "strike the balance" between two conflicting requirements: accuracy of the simulation of elastic scattering and similarity of the simulation results to those obtained with MCNP6.1 in single-event mode. The reason why those two requirements are conflicting is the fact that the ENDF/B elastic scattering cross sections (which are used by MCNP6.1) are the so-called "cutoff" cross sections, which correspond to the range of the cosine of the scattering angle ($\mu$) from $\mu = -1$ to $\mu = 0.999999$. The tables provided in [5] include the "total" elastic scattering cross sections, which correspond to scattering into the range from $\mu = -1$ to $\mu = 1$. The latter cross sections become significantly larger than the cutoff cross sections at electron energies of the order of 10 MeV or greater (for example, in the case of Au, the ratio of the two cross sections is approximately 4 at 30 MeV, and approximately 30

at 100 MeV). As the electron energy is increased, the "total" elastic scattering cross section tends to a constant value, whereas the cutoff cross section decreases rapidly (see the graphs in [5]). However, MCNP6.1.1 uses the ENDF/B elastic scattering cross sections as though they were the "total" ones. The meanings of the numbers that can be specified after "ELASTIC_SMALL_ANGLE" are explained below:

0 – the ENDF/B elastic scattering cross sections are assumed to correspond to the entire range of the cosine (from −1 to 1), and the small-angle PDF (corresponding to $\mu > 0.999999$) is linearly extrapolated from larger angles;

1 – the same interpretation of the ENDF/B cross sections as in the case "0", but small-angle scattering ($\mu > 0.999999$) is simulated using the analytical PDF derived from multiple-scattering theories and normalized on the basis of the continuity of the angular PDF [1];

2 – the ENDF/B cross sections are assumed to correspond to the cosine values from −1 to 0.999999, and the probability of small-angle scattering is calculated by the same method as in the case "1", i.e., using the analytical angular PDF, which is required to be continuous at $\mu = 0.999999$;

3 – the large-angle cross sections are taken from the ENDF/B library (as in the case "2"), whereas the small-angle cross sections are calculated as the difference of the original EEDL elastic scattering cross sections (which correspond to scattering into the range from $\mu = -1$ to $\mu = 1$) and the ENDF/B cross sections. In this case, the small-angle (analytical) PDF is normalized using the known small-angle cross sections (instead of the continuity condition used in "1" and "2"). Consequently, a discontinuity of the PDF is possible at $\mu = 0.999999$. The EEDL elastic scattering cross sections are stored in the ASCII file "Data\EEDL_elastic.dat".

The following three keywords control the error introduced into the angular cumulative distribution function (CDF) of elastic scattering when some of the points in the CDF tables are removed (this removal of points is needed to speed up the process of searching for a probability value in the CDF table). The random variable that is sampled using those CDF tables is the cosine of the scattering angle. That cosine will be denoted $\mu$.

ELASTIC_CDF_CUTOFF specifies the minimum value of the CDF, below which all points are removed (except for the initial point, which corresponds to CDF = 0 and $\mu = -1$). If at least one point is removed, then the initial value of the probability density function (PDF) is replaced by the average PDF over the interval of $\mu$ from −1 up to the value of $\mu$ corresponding to the first CDF value that is above the mentioned cutoff.

ELASTIC_CDF_STEP is the maximum allowed difference of two adjacent CDF values in the CDF table. A point can be removed only if the corresponding CDF value exceeds the previous accepted CDF value by less than this "step". However, this condition is not sufficient for removal of a point (see description of the next keyword).

ELASTIC_PDF_ERROR controls another condition that must be tested in order to determine if a point can be removed: the deviations of PDF values corresponding to the removed points from the linearly interpolated PDF (i.e., from the straight line joining the two accepted PDF values over the interval where the removed points are) must not exceed this parameter multiplied by the average PDF calculated over the same interval. I.e., this parameter may be interpreted as the maximum allowed relative error of PDF introduced by removal of points. If ELASTIC_PDF_ERROR is zero, no points will be removed, i.e., the original CDF and PDF tables will be used during the simulation. ***Note*:** This removal of points is useful only in the case "ELASTIC_DPW 1" (see above for a description of the keyword "ELASTIC_DPW"); otherwise, it would not cause a significant improvement of performance, because the tables of elastic scattering angular PDF in the ENDF/B library are already small enough. Consequently, the following convention is used: if this parameter is entered with the minus sign, then it will be applied only in the case "ELASTIC_DPW 1", otherwise it will be assumed to be zero. In order

to force the reduction of points in the ENDF/B elastic scattering angular CDF tables, too, this parameter must be entered without the minus sign.

The 15 keywords that will be described next are related to "mapping" of various tables of sorted non-equidistant values used for cross section lookup (e.g., the table of electron energies used for selecting the angular CDF of elastic scattering corresponding to the current energy, or the table of CDF used for selecting the elastic scattering angle corresponding to the current value of probability) to "index arrays". Each element of an index array corresponds to a particular "index value", and all index values are equidistant (further on, each interval between two adjacent index values will be called a "bin"). Each element of the index array is equal to the sequence number of the largest element of the corresponding original table that does not exceed the corresponding index value. If the bins can be made narrow enough to accommodate no more than one value of the original table (a "breakpoint"), then the index array can significantly shorten the look-up procedure, because the sequence number of the index value corresponding to the current value of the control variable (for example, energy) is determined simply by dividing the latter value by the bin width. After that, only one comparison operation is needed in order to locate the two adjacent elements of the corresponding original table that "bracket" the current value of the control variable. In MCNelectron, each bin is allowed to contain up to 2 breakpoints, hence the maximum number of comparison operations that may be needed is 2. This mapping may be not only linear (as in the above example), but also logarithmic (which is optimal when the value of the control variable increases exponentially as a function of its sequence number), or "reverse logarithmic" (which is optimal when the deviation of the control variable from its maximum value decreases exponentially as a function of its sequence number).

There are six types of tables that can be mapped to index arrays:

1) the table of incident electron energy values used for estimating the cross sections of various types of electron interactions,
2) the table of electron energies, where each energy corresponds to a table of angular CDF of elastic scattering,
3) tables of angular CDF of elastic scattering mentioned above,
4) the table of incident electron energy values used for estimating the energy transfer during atomic excitation,
5) the table of incident electron energies, where each energy corresponds to a table of energy CDF of a knock-on electron (emitted during impact ionization),
6) tables of energy CDF of knock-on electrons mentioned above.

The latter 5 types of tables are defined for each chemical element separately. The first mentioned table is used for sampling both the chemical element and the type of the interaction. Regardless of the table type, at least two parameters must be defined: "index type" and "index exponent". The corresponding keywords end with "_INDEXTYPE" and "_INDEXEXPONENT", respectively. The list of 12 keywords used for specifying the index type and the index exponent for each of the mentioned six types of tables is provided below:

1) SIGMA_E_INDEXTYPE, SIGMA_E_INDEXEXPONENT
2) ELASTIC_E_INDEXTYPE, ELASTIC_E_INDEXEXPONENT
3) ELASTIC_CDF_INDEXTYPE, ELASTIC_CDF_INDEXEXPONENT
4) EXC_E_INDEXTYPE, EXC_E_INDEXEXPONENT
5) ION_E_INDEXTYPE, ION_E_INDEXEXPONENT
6) ION_SEC_E_CDF_INDEXTYPE, ION_SEC_E_CDF_INDEXEXPONENT

The index type defines the type of mapping. The types of mapping, sorted by increasing complexity, are the following:

index type = 1: linear, with at most 1 breakpoint in a bin,
index type = 2: linear, with at most 2 breakpoints in a bin,
index type = -1: logarithmic, with at most 1 breakpoint in a bin,
index type = -2: logarithmic, with at most 2 breakpoints in a bin,

index type = -3: reverse logarithmic, with at most 1 breakpoint in a bin,
index type = -4: reverse logarithmic, with at most 2 breakpoints in a bin.

If the index type is 0, then no mapping will be applied to the given type of tables, i.e., values will be found in those tables using the method of binary search.

The index exponent defines the maximum number of index values, i.e., the maximum size of the index array. The latter number is calculated as $2^{\text{index exponent}} + 1$. Having specified the index type and the index exponent, the program will test all index types from the simplest one up to the specified one (in the same order as above) in an attempt to find the simplest possible type of mapping that can be obtained without exceeding the specified maximum size of the index array (if no more than 1 breakpoint is required, then the types of mapping with 2 breakpoints will be skipped). If such type of mapping is found, the program will successively halve the size of the index array (i.e., decrease the index exponent by 1) in an attempt to find the smallest index exponent that can be used without exceeding the specified maximum number of breakpoints in a bin. If all attempts fail, index type will be reset to 0, which means that no mapping will be applied to the given type of tables, i.e., values will be found in those tables using the method of binary search.

In addition to the index type and the index exponent, the two types of tables with CDF values (i.e., tables of angular CDF of elastic scattering and tables of energy CDF of knock-on electrons) need one more parameter specified. That parameter indicates if the first interval in the table (i.e., the interval between the first two points) should be excluded from mapping. If so, then before starting the lookup process the program will check if the current value of the control variable belongs to that interval, and the index array will be used only if it is determined that the value is outside that interval. The corresponding two keywords are

ELASTIC_CDF_EXCLUDEFIRST and ION_SEC_E_CDF_EXCLUDEFIRST

They must be followed by "1" or "0", depending on whether the first interval should be excluded or not. ***Note*:** These keywords have an effect only in the case of linear mapping (in the case of logarithmic or reverse logarithmic mapping of CDF tables, the first interval is always excluded).

The last parameter related to index arrays is a "switch" that controls output of information about used index arrays (i.e., types of mapping and maximum sizes of index arrays for each type of tables and each type of mapping) in the console window before starting the simulation. The corresponding keyword is

OUTPUT_INDEX_ARRAY_INFO

(it must be followed by "1" or "0", as for all other switches).

The following keywords are related to tallies and definition of the source energy distribution:

PULSE_HEIGHT is used to specify calculation of a pulse-height tally, i.e., the tally of absorbed energy, and optionally the corresponding energy-deposition tally. It may be specified either for the entire system or for a particular cell (see Section 4.7 for more information about the tallies).

ECELL_ENTRY<n>, PCELL_ENTRY<n> or CELL_ENTRY<n>, where "<n>" is a positive integer number, specifies a cell-entry tally for electrons, photons or all types of particles, respectively, i.e., an energy tally of particles crossing a user-specified subset of faces of a particular cell, and optionally the corresponding energy transfer tally. The electron tally data include the positrons, too. See Section 4.7 for more information about the tallies.

ETALLY, PTALLY or TALLY, or the same keyword with an appended positive integer number, specifies a plane or a set of parallel equidistant planes, as well as one or more tallies of the number of electrons, photons or all types of particles crossing those planes, respectively. The electron tally data include the positrons, too. Additional details are in Section 4.7.

E, THETA, MU, R, PHI, X, or Y, or the same keyword with an appended positive integer number, specifies a range of values of one of seven quantities (energy, angle of incidence, cosine of the incident angle, radial coordinate, azimuth angle, X coordinate, or Y coordinate respectively) to be used in specifications of cell-entry and plane-crossing tallies (see Section 4.7 for more information about the tallies and about defining a range of values).

TALLIES_PER_SOURCE is a “switch” that specifies whether the counts in the tallies should be divided by the total number of source particles.

TALLIES_E_DEPOSITION is a “switch” that specifies whether the files with the pulse-height tally data should include the values of the energy deposited in each bin by default (this setting can be overridden for individual pulse-height tallies). See Section 4.7 for more information about the tallies.

TALLIES_E_FLUX is a “switch” that specifies whether the files with the cell-entry or plane-crossing tally data should include the values of energy transfer by default (this setting can be overridden for individual tallies). See Section 4.7 for more information about the tallies.

TALLIES_OUTPUT is used to switch the periodic saving of the tally data during the simulation “on” or “off”. This keyword must be followed by the number 2, 1, or 0. If the number is 2, then the data of all tallies defined in the input file will be written to files during the simulation. If the number is 1, then only the data of the tallies with the defined maximum error (i.e., the “control tallies”) will be output (such tallies are described in Section 4.7.4). If the number is 0, then the tally data will not be output during the simulation (the tally data will be written to files only after ending the simulation). This setting can be overridden for individual tallies (see Section 4.7.6).

TALLIES_OUTPUT_INTERVAL is used to set the time period of saving the tally data to files. This time must be specified in seconds. The smallest allowed interval is 1 s. This setting can be overridden for individual tallies (see Section 4.7.6).

E_CONTIGUOUS is a “switch” that specifies the order of energy and probability values in the definition of the source energy spectrum (see below).

Any non-empty line that is not a comment line and that starts not with one of the mentioned keywords is interpreted as a part of the definition of the energy spectrum of incident particles. Such lines must contain only numbers. If there is only one such line and if it contains only one number (as in the above example), then a source of monoenergetic particles will be simulated, and that number will be interpreted as the source energy value. In the above example, the source emits 1000 electrons with energy 1 MeV. If there are two or more numeric values in total, then one half of them should have the meaning of energy and the other half should have the meaning of corresponding weights. Each weight is proportional to probability that the source particle energy will belong to the energy interval with limits equal to the last two energy values (i.e., those numbers define the table of the particle energy distribution). The mentioned weights need not to be normalized to 1 (MCNelectron does that automatically). The first weight must be zero. The relative order of energies and weights is defined using the mentioned keyword “E_CONTIGUOUS”. If the integer number that follows E_CONTIGUOUS is non-zero, then the first half of all numbers will be interpreted as energy values, and the second half will be interpreted as the weights, listed in the same order as energies. Otherwise, energies and corresponding weights must be in pairs: “Energy No.1 Weight No.1 Energy No.2 Weight No.2”, etc.

Only the definition of the source energy spectrum, the number of source particles and some of the keywords related to geometry and materials (see Section 4.2) must be present in the input file. All other keywords may be absent. Then the corresponding parameters will be assigned the default values. In addition, if the source energy value and the number of source particles have been specified on the command line using the keywords “E” and “N”, then the definition of the source energy spectrum and the keyword “N” may be absent, too (otherwise those specifications in the input file will be ignored). The default values are the following:

| | |
|---|---|
| E_UNIT | MeV |
| CUT_E | 1e-4 |
| CUT_P | 1e-5 |
| TASKS | 1 |
| PART | E |

```
All 19 switches related to the physical models are “on” (i.e., equal to 1)
TALLIES_PER_SOURCE      0
TALLIES_E_DEPOSITION    1
TALLIES_E_FLUX    1
TALLIES_OUTPUT    2
TALLIES_OUTPUT_INTERVAL     60
E_CONTIGUOUS      1
SEED            123
DIR             C:\ENDF\
NSTREAMS    64
COH_THR     0.0001
SIGMA_ERR   0.001
DIFFUSION_TOTAL_TO_INELASTIC_RATIO_THR      1e18
DIFFUSION_DIST_TO_RMSPATH_RATIO_THR  5
SMOOTH_CDF_ELASTIC    -1
SMOOTH_CDF_INELASTIC  1
ELASTIC_SMALL_ANGLE   0
ELASTIC_CDF_CUTOFF    1e-12
ELASTIC_CDF_STEP      0.01
ELASTIC_PDF_ERROR     -0.1
SIGMA_E_INDEXTYPE     -2
SIGMA_E_INDEXEXPONENT       17
ELASTIC_E_INDEXTYPE   -2
ELASTIC_E_INDEXEXPONENT     17
ELASTIC_CDF_INDEXTYPE       -2
ELASTIC_CDF_INDEXEXPONENT         18
EXC_E_INDEXTYPE       -2
EXC_E_INDEXEXPONENT   17
ION_E_INDEXTYPE       -2
ION_E_INDEXEXPONENT   17
ION_SEC_E_CDF_INDEXTYPE     -4
ION_SEC_E_CDF_INDEXEXPONENT     18
ELASTIC_CDF_EXCLUDEFIRST    1
ION_SEC_E_CDF_EXCLUDEFIRST        0
OUTPUT_INDEX_ARRAY_INFO   0
```

The default values of the geometry-related parameters are given in Section 4.2. The default values of the parameters related to output of track data are given in Section 4.6. The default values of the parameters related to interaction forcing are given in Section 4.8.

### *4.2. Keywords used for definition of geometry and materials*

MCNelectron can be run in two geometry modes: “simple geometry” and “complex geometry”. In simple geometry mode, the simulation setup consists of an infinite homogeneous layer of a material with arbitrary composition and a parallel beam of particles incident upon it at any angle (the initial point of the beam may be inside the layer; the beam radius is zero). In complex geometry mode, the simulation setup consists of up to 5000 “cells”, each of them defined as a region of space enclosed by any combination of up to 5000 bounding surfaces, where each surface can be one of the following:

(a) a plane,
(b) a sphere,
(c) a circular cylinder,
(d) a circular one-sheet cone.

The maximum allowed total number of surfaces is 10000 (a surface may be used in definitions of several cells). The definition of the source of radiation in complex geometry mode requires specifying the source position and the direction of radiation independently. The source position can be either a single point, or sampled randomly from one of those spatial distributions:

(a) a radially symmetric distribution on a plane with Gaussian fall-off,
(b) the uniform distribution inside an arbitrary cell.

The direction of source radiation can be either fixed (a parallel beam), or sampled randomly from one of those angular distributions:

(a) the uniform distribution of directions inside a cone of radiation,
(b) the isotropic distribution.

Any simulation performed in simple geometry mode can be re-done in complex geometry mode, with identical results. However, the processing time in complex geometry mode would be longer. Prior to v1.2.0, MCNelectron could be run only in simple geometry mode.

An example of an input file using the complex geometry mode is presented below.

```
S1 CY   0  0  0    0  0  1    2
S2 P   -4  0  0    1  0  0
S3 P   -2  0  0    1  0  0
S4 P   -3 -4  0    0  1  0
S5 P   -3 -1  0    0  1  0
S6 P  4   4   0    1  0  0
S7 P  4   4   0    0  1  0
S8 P  2.5 2.5 0    1  1  0
S9 P    0  0 -100  0  0  1

M1 "Argon" 18 1
M2 "Water" 1 2   8 1

C1 1 -0.001        -1          9
C2 2 -1             2 -3  4 -5 9
C3 0 "Empty space" -6 -7  8    9

SOURCE_POSITION 1  0 0 0
SOURCE_DIRECTION 3

COORD_X -4 8   4
COORD_Y -4 8   4
COORD_Z -100

PART E
0.1
N 1e6

E1   0.0029 0.00325 35
E2   0  0.1  100

PULSE_HEIGHT C2  E2
PCELL_ENTRY3 3 2  -8    E1
PTALLY  S8   LIM X -2.12132 2.12132   E1
ECELL_ENTRY2 2 1   E2
TALLIES_PER_SOURCE 1
```

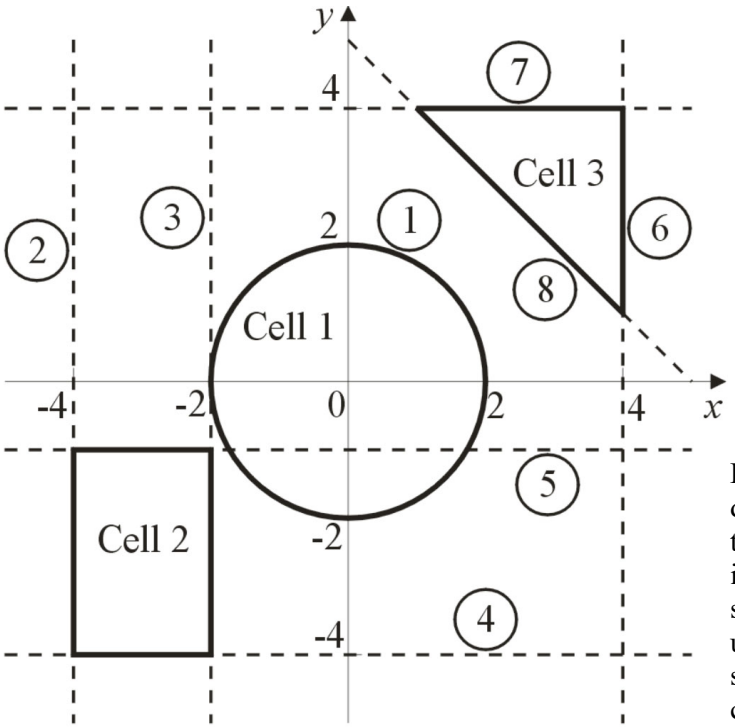


Fig. 4.1. The cross-section of the two-dimensional simulation setup corresponding to the sample MCNelectron input file given in the text. The dashed lines represent eight surfaces (one cylinder and seven planes) used in the definitions of the three cells. The surface identifiers are shown inside the small circles.

This example defines a system consisting of three cells and an isotropic point source of electrons with energy 100 keV. The cross-sectional view of the simulation setup is shown in Fig. 4.1. The input file starts with definitions of nine surfaces (lines starting with "S"): a cylinder used in the definition of cell 1, four planes used in the definition of cell 2, three planes used in the definition of cell 3, and a plane normal to the *Z* axis bounding all the cells from below. After that, there are definitions of two materials (lines starting with "M"), three cells (lines starting with "C"), source geometry (lines starting with "SOURCE") and three sets of coordinate planes (lines starting with "COORD"). The coordinate planes are needed to split all the system into rectangular volume elements, or "voxels". This makes it possible to decrease the time needed to track a particle through a complex geometry, because only the intersections of the particle's track with the surfaces that are inside the current voxel have to be checked (see Section 4.5 for details). The remaining part of this input file specifies the source particle type, energy, number of source particles, and four tallies.

The geometry mode (simple or complex) may be specified using the keyword COMPLEX_GEOMETRY. If the integer number following that keyword is non-zero, then the simulation will be performed in complex geometry mode, otherwise it will be performed in simple geometry mode. However, since the simple and complex geometries are defined using different sets of keywords, the geometry mode may be also selected by using keywords specific to the needed geometry mode. The keywords corresponding to the simple geometry mode cannot be mixed with the keywords corresponding to the complex geometry mode in a single input file (otherwise the program will quit with an error message). There is only one geometry-related keyword that is common to both geometry modes:

BEAM X Y Z A B C is used to specify the starting point and direction of the incident beam. X, Y and Z are the coordinates of the starting point (in cm). A, B and C are the components of the vector that has the same direction as the incident beam (that vector may be of any non-zero length). There are no restrictions on the values of X, Y and Z.

If the keyword BEAM is absent, then the following default specification is assumed:

BEAM 0 0 0 0 0 1

The remaining geometry-related keywords will be described separately for the simple geometry mode and for the complex geometry mode.

*Geometry-related keywords that are specific to the simple geometry mode*

MAT is used to define composition of the target material. This keyword must be followed by one or more pairs of space-delimited numbers. The first number in each pair is the atomic number of a chemical element, and the second number is its atomic fraction. If the fractions are negative, they will be interpreted as mass fractions. The atomic fractions are not required to be normalized to 1 (e.g., the numbers of atoms in the molecule of the material may be specified instead of the actual atomic fractions).

CONC is used to specify the total atomic concentration ($cm^{-3}$). If the number that follows CONC is negative, it is interpreted as total density ($g/cm^3$).

THICK is used to specify the thickness of the target layer (cm). If the number that follows THICK is negative, it is interpreted as mass thickness ($g/cm^2$).

SURFACE X Y Z A B C is used to specify one surface of the layer. X, Y and Z are coordinates of any one point of that surface (in cm). A, B and C are the components of the normal vector (that vector may be of any non-zero length). The other surface of the layer is in the direction of the normal vector relative to the surface defined by this keyword, at a distance defined using the keyword THICK (see above).

If the keyword SURFACE is absent, then the following default specification is assumed:

SURFACE 0 0 0 0 0 1

The other three mentioned keywords (MAT, CONC, THICK) have no default values, hence they must be specified in the input file. The keyword THICK may be absent when TRACKPOS is 0 and the source emits electrons or positrons, because in that case the geometrical parameters are not used during simulation. ***Note***: The directive "TRACKPOS 0" is allowed only in simple geometry mode.

*Geometry-related keywords that are specific to the complex geometry mode*

S<n>, where "<n>" is a positive integer number, is used to define a surface. The mentioned number is the surface identifier. The keywords following "S<n>" depend on the type of the surface:

(a) "S<n> P X Y Z A B C" is used to define a plane. X, Y and Z are coordinates of any one point of that plane (in cm). A, B and C are the components of the normal vector.

A plane can also be defined by specifying the four coefficients of the plane equation or the coordinates of three points on the plane:

"S<n> P A B C D" is used to specify the coefficients of the plane equation $A\,x + B\,y + C\,z - D = 0$;

"S<n> P X1 Y1 Z1 X2 Y2 Z2 X3 Y3 Z3" is used to specify the coordinates of three points on the plane. In this case, the direction of the plane's normal is such that the three points appear to be numbered counter-clockwise when observed from the half-space where the normal vector is directed to.

MCNelectron determines the method of defining the plane from the number of entries after "P" (this number must be 4, 6 or 9).

(b) "S<n> S X Y Z R" is used to define a sphere. X, Y and Z are the coordinates of the center of the sphere (cm), R is the radius of the sphere (cm).

(c) "S<n> CY X Y Z A B C R" is used to define a cylinder. X, Y and Z are the coordinates of a point on the symmetry axis of the cylinder (cm). A, B and C are the components of the vector parallel to the axis of symmetry of the cylinder. R is the radius of the cylinder (cm).

(d) "S<n> CO X Y Z A B C D" is used to define a one-sheet cone. X, Y and Z are the coordinates of the tip of the cone (cm). A, B and C are the components of the vector parallel to the axis of symmetry of the cone. D is the cone angle in degrees (it must be between 0 and 90). The angle is measured relative to the direction defined by A, B and C.

The direction vector defined by parameters "A", "B" and "C" may be of any non-zero length. The maximum allowed number of surfaces is 10000.

In addition to defining the mentioned four types of "regular" surfaces, the keyword "S<n>" can be used to define the so-called "macrobodies", i.e., the sets of two or more regular surfaces defining various 3D shapes, which can be used in definitions of cells and cell-entry tallies along with the regular surfaces. There are 8 "standard" macrobodies (right circular cylinder, truncated right-angle cone and 6 types of polyhedra) and a "general" macrobody (a shorthand notation for an arbitrary set of user-defined regular surfaces and their "senses"). It is also possible to reference individual facets (i.e., bounding surfaces) of each macrobody using the format "<macrobody No>.<facet No>". For a detailed description of all supported types of macrobodies and the format of their definitions, see Section 4.3.

M<n>, where "<n>" is a positive integer number, is used to define composition of a material. The mentioned number is the material identifier. This keyword must be followed by one or more pairs of space-delimited numbers. The first number in each pair is the atomic number of a chemical element, and the second number is its atomic fraction. If the fractions are negative, they will be interpreted as mass fractions. The atomic fractions are not required to be normalized to 1 (e.g., the numbers of atoms in the molecule of the material may be specified instead of the actual atomic fractions). The number of materials is not limited.

The definition of a material can include the directive "FORCE 1" or "FORCE 0", which "turns on" or "turns off" interaction forcing for a given material, if a set of forced interactions is defined in the same input file (see Section 4.8 for more information about interaction forcing and the keywords related to it).

MCNelectron checks if there are no identical materials in the simulation setup. If a material is found with the composition identical to the composition of one of the materials defined before, then a warning is displayed and all references to that material in definitions of cells and unions are replaced by the reference to the previously-defined material.

A material name may be optionally specified between double quotes after the material identifier (as in the example at the beginning of Section 4.2). The material names are ignored by MCNelectron, but they are displayed in the main window of the "shell" program MCNScript, which is designed for running Monte Carlo simulations with MCNelectron, as well as for organizing MCNelectron input and output files and creating graphics rendering scripts for visualization of the simulation geometry and particle tracks (MCNScript is described in a separate user manual).

C<n>, where "<n>" is a positive integer number, is used to define a cell. The mentioned number is the cell identifier. Each cell is defined as an intersection of "regions", with each region defined as the part of space that is on one side of a given surface. Thus, the definition of a cell requires specifying a set of surfaces and the "sense" of each region relative to the corresponding surface. The sense of a region indicates one of the two regions lying on each side of the corresponding surface. The region with a "positive" sense is the one where the normal vector of the surface is directed to. The normal vector of a sphere, cylinder or one-sheet cone is always directed to the outside of the surface, hence any region corresponding to any of those surfaces has always a uniquely defined sense: the inside of the surface has a negative sense, and the outside of the surface has a positive sense. The half-space formed by a plane is the only type of region whose sense relative to the surface is not uniquely defined: the same half-space may have positive or negative sense, depending on the user-specified direction of the plane's normal.

The sense of each region used in the definition of a cell is specified by the sign "+" or "–" before the surface identifier (the plus sign may be omitted). Before the list of regions, the cell material and its concentration or density must be specified. Thus, the definition of a cell has the following format:

C<n> Material_ID Conc ±Surface_ID1 ±Surface_ID2 ±Surface_ID3 …

If parameter "Conc" is positive, then it is interpreted as the total atomic concentration of the material ($cm^{-3}$), and if it is negative, then it is interpreted as the total density ($g/cm^3$). If the cell

is empty, i.e., not filled with any material, then the zero "0" or any negative integer number must be specified instead of the material identifier and the concentration:

C<n> 0 ±Surface_ID1 ±Surface_ID2 ±Surface_ID3 …

If the definition of a cell includes the identifier of a standard macrobody (see Section 4.3), then it must be preceded by the minus sign. In contrast, the identifier of a general macrobody must be specified without the minus sign. Individual facets of macrobodies are treated as "regular" surfaces, hence they may be specified either with the minus sign or without it. If an identifier of a facet of a macrobody is specified without the minus sign, then it means the same region that is used in the definition of the macrobody. Otherwise, it means the complement of that region.

Some surfaces may be removed from the definition of a cell during initial checking of the geometry. A surface is removed if the region corresponding to that surface entirely contains at least one other region specified in the same definition. If a bounding surface is removed, a warning message is displayed. It should be noted that a bounding surface may be redundant (i.e., effectively absent) even if it is not removed during the initial checking of the geometry. Such bounding surfaces will never contribute to cell-entry tallies (see Section 4.7.2).

The electron or photon cutoff energy may be specified separately for each cell. This is done by including the keyword CUT_E or CUT_P, followed by the electron or photon cutoff energy, anywhere after the specification of the cell material and its concentration, for example:

C1 2 -1 1 -2 -3 CUT_E 0.001 CUT_P 0.01

If a particle enters a cell whose cutoff energy is greater than the particle energy, then that particle will be removed immediately and all its energy will be added to the total energy absorbed in that cell. This fact may be used for simplified simulations involving ideal absorbers or diaphragms. If no cutoff energy is given in the cell definition, then the cell will be assigned the "global" electron and photon cutoff energies specified using the "global" keywords "CUT_E" and "CUT_P" (see Section 4.1).

The volume of a cell (in $cm^3$) may be specified in the cell definition using the keyword "VOL", for example,

C1 0 1 -2 VOL 15.3

The user-specified volume replaces the volume estimated by MCNelectron when calculating the mass of the cell and the average absorbed dose (if the relative difference of those two volumes is simultaneously greater than 1 % and greater than 10 standard deviations, then a warning is displayed). The volume may be specified even for infinite cells (in such a case, the calculated mass will be finite, and no warning will be displayed).

The definition of a cell may include the directive "FORCE 1" or "FORCE 0", which "turns on" or "turns off" interaction forcing for a given cell, if a set of forced interactions is defined in the same input file (see Section 4.8 for more information about interaction forcing and the keywords related to it).

In addition to the region identifiers, the so-called "complements" of previously defined cells may be listed in the definition of a cell. The complement of a cell is the region of space that is outside the cell. The complement of a cell is specified using the complement operator "#" followed by the cell identifier. The cell specified after the complement operator must be entirely inside the cell that is being defined (those two cells may share some of the bounding surfaces). Usage of the complement operator indicates that the region inside one of the previously defined cells must be excluded from the current cell. The cells specified after the complement operator may themselves be defined using the complement operator. However, one should keep in mind that the excluded region of space is defined by taking into account only the bounding surfaces that are listed in the definition of the cell indicated after the complement operator (that is to say, all the cells contained inside the latter cell are excluded, too). Definition of a cell must include at least one bounding surface (i.e., a cell cannot be defined in terms of the complements of other cells only).

The complement operator makes it possible to define cells that would otherwise have to be defined as a group of several adjacent cells, or as a union of cells (see below about the "UNION" keyword). For example, let us assume that the simulation setup includes a rectangular box with one side open. Without the complement operator, each of the five walls of the box would have to be defined separately as a rectangular cell with 6 bounding surfaces. Using the complement operator, it is sufficient to define two rectangular cells so that one of them is inside another one, and one face of the smaller cell is a part of one face of the larger cell (that face corresponds to the open side of the box). Definition of each cell includes a list of 6 planes. Since one of them is shared by both cells, there are 11 planes in total. Let us assume that their normals are directed towards the center of the box, and plane No. 1 is the one that is shared. The cells are not filled with any material. Then, they should be defined as follows:

```
C1   0    1  2  3  4  5  6
C2   0    #1  1  7  8  9  10  11
```

As demonstrated by this example, usage of the complement operator can simplify the specification of the system geometry significantly.

The empty cells may be optionally assigned names (similarly to the materials). In this case, the zero or a negative number after "C<n>" has the role of a "material identifier". I.e., all empty cells with the same number after "C<n>" will be assigned the same name. Consequently, it is sufficient to specify it only once for each value of the number after "C<n>" (and if it is specified more than once, then the last name will be used). The name of an empty cell must be written between double quotes after the zero or a negative number following "C<n>" (see the example of the input file at the beginning of Section 4.2). As in the case of materials, the names of empty cells are ignored by MCNelectron, but they are displayed in the main window of the "shell" program MCNScript, which is designed for running Monte Carlo simulations with MCNelectron, as well as for organizing MCNelectron input and output files and creating graphics rendering scripts for visualization of the simulation geometry and particle tracks (MCNScript is described in a separate user manual).

The maximum allowed number of cells is 5000. The maximum allowed number of regions in a definition of a cell is 5000.

SOURCE_POSITION is used to define the rule that must be used when sampling the point where a source particle originates. The first number following this keyword is the source "position type". It must be 1, 2, or 3:

1 – point source,
2 – radially symmetrical distribution on a plane with Gaussian fall-off;
3 – uniform distribution inside an arbitrary cell.

The source position type must be followed by additional space-delimited numbers, whose meaning depends on the position type. In the case of a point source (position type 1), there must be three additional numbers equal to *x*, *y* and *z* coordinates of the source (cm). In the case of a planar distribution (position type 2), there must be eight additional numbers: *x*, *y* and *z* coordinates of the distribution center (cm), *x*, *y* and *z* components of the plane's normal vector (that vector may be of any non-zero length), radius of the circle where the two-dimensional probability has a constant term (cm), and the root-mean-square width of the Gaussian fall-off (cm). In the case of a uniform distribution inside a cell (position type 3), there must be one additional number equal to the cell identifier.

In the case of a radially symmetrical distribution on a plane (position type 2), the position of a source point is obtained by adding two random displacement vectors to the position of the distribution center: a displacement corresponding to a uniform distribution inside a circle, and a two-dimensional Gaussian displacement (the latter is itself a sum of two random one-dimensional Gaussian displacements along two perpendicular axes on the plane). Consequently, the sampling density is not exactly uniform inside the mentioned circle: it

approaches a uniform distribution at values of the radial coordinate that are less than the circle radius at least by several RMS widths.

SOURCE_DIRECTION is used to define the rule that must be used when sampling the direction of motion of a source particle. The first number following this keyword is the source "direction type". It must be 1, 2, or 3:

1 – one direction (parallel beam),
2 – uniform distribution of directions inside a cone of radiation,
3 – isotropic distribution.

The source direction type must be followed by additional space-delimited numbers, whose meaning depends on the direction type. In the case of a parallel beam (direction type 1), there must be three additional numbers equal to *x*, *y* and *z* components of the particle's direction vector (that vector may be of any non-zero length). In the case of a uniform distribution of directions inside a cone of radiation (direction type 2), there must be four additional numbers: *x*, *y* and *z* components of the vector parallel to the distribution's symmetry axis (that vector may be of any non-zero length) and the cone angle in degrees (it must be between 0 and 180). The cone angle is measured from the direction defined by the mentioned vector. In the case of the isotropic distribution (direction type 3), no additional parameters are necessary.

COORD_X, COORD_Y, COORD_Z are used to define sets of equidistant coordinate planes normal to *X*, *Y* and *Z* axis, respectively. In general, each of those keywords must be followed by one or several pairs of numbers, where the first number is the coordinate of the first plane in a set (cm), and the second number is the number of planes in that set. The distance between planes in a set is determined from the starting coordinate of the next set. Consequently, the last "set" is always a single plane, so that the number of planes may be omitted for that set (and if that number is specified, then it must be 1). In the following example, there are three sets of planes normal to the *Y* axis: a set of 4 planes with *y* coordinates –8, –6, –4, –2, a set of 6 planes with *y* coordinates 0, 1, 2, 3, 4, 5, and the third set consisting of a single plane at $y = 6$:

```
COORD_Y  -8 4   0 6   6
```

The starting coordinates of all sets are sorted before calculating the inter-plane distances. Consequently, if there are two or more directives starting with the same "COORD_..." keyword (for example, two "COORD_Y" directives), then the "next" set may be defined in another "COORD" directive, and the set that is the last one in a particular "COORD" directive may be not the last one in the final sorted sequence of all sets.

The coordinate planes defined using the keywords COORD_X, COORD_Y, COORD_Z split all the volume of the simulated system into rectangular volume elements ("voxels"), which are used to speed up tracking of particles through complex geometries. Each voxel has an associated data structure with some information about the cells and surfaces that intersect with that voxel. As a result, the number of surfaces that have to be checked for intersection with the particle's track is reduced, because only the surfaces intersecting with the voxel that contains the particle have to be checked (see Section 4.5 for more information). Another benefit of defining the coordinate planes is that they facilitate checking the geometry setup for errors. In addition, the coordinate planes are required for sampling the source point position from the uniform distribution inside a cell (if the "source position type" is 3). The source point sampling efficiency may be increased by increasing the number of voxels in the source cell (see also descriptions of keywords N_RAND_CELL_VOLUME and N_RAND_SOURCE_VOLUME).

The maximum allowed number of sets for each of the three Cartesian coordinates is 10. The total number of parallel planes must not exceed $10^7$. There is an additional limitation: the total number of voxels formed by intersections of all coordinate planes must not exceed $10^7$.

UNION<n>, where "<n>" is a positive integer number, is used to define a union of cells. The format of this directive is similar to the format used for defining a cell, with the only difference that the numbers listed in the "UNION" directive are the cell identifiers, rather than the region

identifiers. For example, the following directive defines a union of cells No. 1, 4, 3 filled with material No. 5 with density 1 g/cm$^3$:

UNION1 5 -1 1 4 3

The cell complement operator "#" cannot be used in the definition of a union. The definitions of cells must precede the definition of their union (the materials and densities specified in the definitions of those cells are ignored). As the definition of a cell, the definition of a union may include values of the electron and photon cutoff energies (keywords CUT_E and CUT_P), the value of the volume of the union of cells (keyword VOL), and the directive "FORCE 1" or "FORCE 0", which "turns on" or "turns off" interaction forcing for a given union of cells.

The main difference between the cells used in the definition of a union and all other cells is that the former are allowed to intersect, whereas intersection of any two cells that are not parts of the same union is treated as a geometry error. A union may be interpreted as a composite cell. Although programmatically the cells composing the union maintain their identity, the cell-by-cell statistics in the MCNelectron output files pertain to the union as a whole. In all directives requiring the identifier of a cell (for example, in the definition of a cell-entry or pulse-height tally for a particular cell, or in the definition of the source cell), only the identifier of the first cell of the union may be specified (the other cells of the union will be taken into account automatically). The identifier of the union itself (i.e., the number after "UNION") cannot be used in any directives except in the definition of the union.

DISTANCE_PRECISION is used to define the "distance precision" (cm). If the distance between two points is less than this parameter, they are assumed to coincide, and if the distance between a point and a surface or a line is less than this parameter, then the point is assumed to be on the surface or the line. ***Note***: The distance precision must always exceed the round-off error of the particle's coordinates. The relative round-off error is slightly less than $10^{-15}$. Consequently, the ratio of the distance precision to the maximum possible value of the particle's coordinates must be greater than $10^{-15}$. Otherwise, geometry errors are possible. Those errors are usually signaled by a corresponding message, but it is also possible that in some situations a geometry error of this type could lead to incorrect results without any warning message.

ANGLE_PRECISION is used to define the "angle precision" (rad). If the angle between two straight lines or two planes is less than this parameter, they are assumed to be parallel.

MAXDISTANCE_RATIO is used to define the ratio of the maximum distance and the distance precision. Although the intersection points of lines and surfaces are usually calculated analytically, MCNelectron occasionally resorts to an iterative procedure using the simplex method to find the points where a surface intersects with the intersection line of other two surfaces. If the intersection line is infinite, then the maximum displacement of the test point during the iterative procedure must be limited in order to avoid the infinite loop. The number specified after this keyword is the mentioned maximum displacement divided by the distance precision. The product of DISTANCE_PRECISION and MAXDISTANCE_RATIO should be at least by an order of magnitude greater than the dimensions of the simulated system.

CHECK_FOR_INTERSECTIONS controls checking for intersections of cells that are not parts of the same union of cells. This keyword must be followed by the number 0, 1, 2 or 3, where "0" corresponds to absence of any checking, and the other numbers correspond to varying degrees of thoroughness of the checking:

1 – Analysis of the edge equations of the possible intersection region;

2 – Additional checking for intersection of each of the two cells with the rectangular box representing the intersection region of the two boxes bracketing each cell (if at least one of the cells does not intersect with the mentioned box, it means that the cells do not intersect with each other, otherwise no conclusion can be made). This method is applied if the first method fails to yield conclusive results;

3 – Additional checking for existence in the mentioned rectangular box of at least one voxel that either intersects with both cells or is entirely contained in one of them (if there is no such voxel, it means that the cells do not intersect, otherwise no conclusion can be made). This method is applied if the first two methods fail to yield conclusive results. ***Note***: When a voxel is designated as entirely contained in a particular cell, all information about the other cells that may intersect with that voxel is lost. That is why existence of such a voxel leads to inconclusive results when method No. 3 is applied.

Methods No. 2 and 3 rely on the presence of the "bracketing" coordinate planes defined using the mentioned keywords COORD_X, COORD_Y and COORD_Z. If there are no coordinate planes defined, then methods No. 2 and 3 are not applied. In addition, method No. 3 is applied only if the intersection of the two bracketing boxes contains more than one voxel (because, if only one voxel is contained in it, then method No. 3 would be identical to method No. 2). If it is determined that the cells intersect, then the program quits with a message about the geometry error. If the final results of the checking are inconclusive, then a corresponding warning is displayed, but execution of the program is not interrupted.

N_RAND_CELL_VOLUME is used to define the minimum number of random points that should be generated for approximate estimation of the cell volume. The cell volumes are estimated by randomly generating the specified number of points inside all voxels that are partly inside the given cell and then determining the fraction of points that are inside the cell. The cell volumes can be estimated only for the cells that are "bracketed" between *X*-, *Y*- and *Z*- coordinate planes. Consequently, all three mentioned sets of coordinates planes must be defined (they are defined using the mentioned keywords "COORD_X", "COORD_X" and "COORD_Z"). If a cell is not bracketed by coordinate planes, then its volume will not be calculated. In general, this is not treated as an error, because the cell volumes are calculated only as an additional piece of information for the user. There is one exception, however: if the source is uniformly distributed inside a cell (source position type 3), then that cell must be bracketed between coordinate planes, because this is necessary for random sampling of the source points.

N_RAND_SOURCE_VOLUME is used to define the minimum number of random points that should be generated for preliminary estimation of the source point sampling efficiency. This parameter is used only when the source is uniformly distributed inside a cell (source position type 3). The source point sampling efficiency is estimated by generating the specified number of random points and calculating the fraction of points that are inside the source cell. The only purpose of this calculation is informing the user about the sampling efficiency before starting the simulation (the user can then interrupt the program by pressing the keys "Ctrl+C" and modify the simulation parameters if the efficiency is too low). The value of this parameter must be a multiple of the number of random points that should be generated for approximate estimation of volumes of non-source cells (defined using the keyword "N_RAND_CELL_VOLUME").

N_RAND_TOTAL_VOLUME is used to define the total number of random points for estimation of the cell volume and the source point sampling efficiency. The number of random points per cell is obtained by dividing the value of this parameter by the number of cells. If the result is less than the number defined using N_RAND_CELL_VOLUME, then the latter is used instead.

RAND_BLOCK_SIZE is used to define the size of a "block" of random points for determining the cell volumes. The number of random points generated for approximate estimation of volumes of non-source cells (defined using the keyword "N_RAND_CELL_VOLUME") must be a multiple of this parameter. ***Note***: This parameter is included mainly for programming convenience, in order to make it easier to ensure that estimates of cell volumes do not depend on the number of CPU threads used for geometry processing (see the description of the "NTHREADS_GEOM" keyword below).

NTHREADS_GEOM is used to define the number of concurrent CPU threads to be used in two stages of initial geometry processing: definition of the data associated with each voxel (see also Section 4.5) and calculation of cell volumes.

OUTPUT_CELL_INTERACTION_STATS is a "switch" that "turns on" or "turns off" output of the overall cell-by-cell interaction statistics, such as the total number of electrons and photons that have entered and exited each cell, or the total number of collisions and the average energy loss per collision in each cell (see Section 6 for information about the output file format). If the integer number following this keyword is non-zero, then the statistics will be calculated; otherwise, they will not be calculated. The data structure with overall interaction statistics requires more than 1 kB of computer memory. In each thread, the memory amount allocated for those statistics is equal to the number of cells times the memory amount needed to store the statistics for one cell. Consequently, if, for example, there are 1000 cells and 1000 threads, then the total memory needed to store the overall cell-by-cell interaction statistics would be greater than 1 GB. In such cases, it might be preferable to turn off output of those statistics, because then the mentioned amount of memory would not be reserved for the overall cell-by-cell interaction statistics and could be used for other purposes.

USE_FREE_MOTION_RADIUS controls an option to use the "free-motion sphere", i.e., the spherical region whose intersection with the current voxel is completely contained inside the current cell. While the particle is inside this sphere, the particle is guaranteed to be in the current cell, hence there is no need to check for intersection of the particle's track with the bounding surfaces of the current cell. The integer number following this keyword must be 0, 1 or 2. If it is zero, then the free-motion sphere will not be used, potentially increasing the simulation time, because the checking for crossing of interfaces between cells will be done more often. If the number is 1, then the position of the center of that sphere and its radius are not stored in memory; instead, they are calculated after starting or resuming the particle track, or after exit from the previous such sphere, or after crossing the voxel boundary (in those cases, the current position of the particle becomes the center of the new free-motion sphere). If the number is 2, then the center position and the radius of that sphere are stored in memory and loaded after resuming the particle track (when loading the banked particle data from memory), decreasing the amount of calculations that have to be done but increasing memory usage (storing the values of the three coordinates of the center and the radius of the sphere requires 32 additional bytes of memory for each banked particle). In the latter case, the center position and the radius of the free-motion sphere would be re-calculated only after exit from the previous free-motion sphere or after crossing the voxel boundary.

The default values of the geometry-related parameters specific to the complex geometry mode are given below:

```
COMPLEX_GEOMETRY              1
SOURCE_POSITION               1   0  0  0
SOURCE_DIRECTION              1   0  0  1
DISTANCE_PRECISION            1e-12
ANGLE_PRECISION               1e-12
MAXDISTANCE_RATIO             1e20
CHECK_FOR_INTERSECTIONS       3
N_RAND_CELL_VOLUME            10000
N_RAND_SOURCE_VOLUME          10000
N_RAND_TOTAL_VOLUME           20000000
RAND_BLOCK_SIZE               1000
NTHREADS_GEOM                 8
OUTPUT_CELL_INTERACTION_STATS   1
USE_FREE_MOTION_RADIUS        1
```

### *4.3. Surfaces defined by macrobodies*

In addition to defining the mentioned four types of "regular" surfaces, the keyword "S<n>" can be used to define the so-called "macrobodies", i.e., the sets of two or more regular surfaces defining various 3D shapes, which can be used in definitions of cells and cell-entry tallies along with the regular surfaces. There are 8 "standard" macrobodies (right circular cylinder, truncated right-angle cone and 6 types of polyhedra) and a "general" macrobody (a shorthand notation for an arbitrary set of user-defined regular surfaces and their "senses"). The type of a macrobody is specified after the keyword "S<n>" using one of the following keywords:

| | |
|---|---|
| RCC | Right circular cylinder |
| TRC | Truncated right-angle cone |
| BOX | General paralellepiped |
| RPP | Rectangular parallelepiped, surfaces normal to major axes |
| RHP or HEX | Right hexagonal prism |
| WED | Wedge |
| ARB4 | Four-sided polyhedron defined in terms of vertex coordinates |
| ARB | Polyhedron with 4 – 8 corners and 4 – 6 sides defined in terms of triplets of corners |
| MB | A general macrobody defined in terms of surface identifiers and their "senses" |

These keywords, excluding "ARB4" and "MB", and their meanings are the same as in MCNP. The standard macrobodies "BOX", "RPP" and "RHP" in MCNelectron must be finite in all dimensions, unlike in MCNP, where they can be infinite in a dimension. Another minor difference from MCNP is that the standard macrobody "BOX" in MCNelectron is not limited to orthogonal boxes (it can be used to define a general parallelepiped). Similarly, the standard macrobody "WED" in MCNelectron is not limited to right-angle wedges.

It is possible to reference individual facets (i.e., bounding surfaces) of each macrobody by specifying the macrobody identifier (i.e., the number after "S" in its definition) and the facet number separated by the decimal point (i.e., by using the format "<macrobody ID>.<facet No>").

Each of the mentioned keywords must be followed by numeric entries whose number and meaning depends on the type of the macrobody (a detailed description of those entries is presented further in this Section). 7 of the 8 standard macrobodies (RCC, TRC, BOX, RPP, RHP, WED and ARB) are defined using the same format as in MCNP. The numbering of the facets of these seven types of macrobodies is also the same as in MCNP.

If the identifier of a standard macrobody is specified in the definition of a cell (or a general macrobody), then it always means the inside of the macrobody. Consequently, it must be preceded by the minus sign. This minus sign is used in order to improve compatibility with MCNP and to be consistent with the practice of assigning the negative sense to the inside of a closed surface (specifying the outside of the macrobody just by removing the minus sign is not allowed). In contrast, the identifier of a general macrobody in the definition of a cell (or another general macrobody) must always be specified without the minus sign, because it is simply a shorthand notation of a set of its bounding surfaces with their associated "senses". Individual facets of macrobodies are treated as "regular" surfaces, hence they may be specified either with the minus sign or without it. If the identifier of a facet of a standard macrobody is specified in the definition of a cell (or a general macrobody) with the minus sign, then it means the same region that is used in the definition of the macrobody (i.e., the region intersecting with the inside of the macrobody). Otherwise, it means the complement of that region. For example, if the MCNelectron input file contains the directive "S10 RCC", then "-10.2" means the half-space intersecting with the inside of the RCC, and "10.2" means the other half-space. The opposite sign convention is applied to facet identifiers of general macrobodies: the minus sign before the notation "<macrobody ID>.<facet No>" means the complement of the region specified in the definition of the general macrobody.

The complete descriptions of all types of macrobodies and of the format of their definitions are given below (some of those descriptions have been copied from the MCNP user's guide [6]).

RCC: Right circular cylinder (can).

S<n> RCC *Vx Vy Vz Hx Hy Hz R*

where *Vx Vy Vz* = center of base
*Hx Hy Hz* = cylinder axis vector
*R* = radius

Example: S1 RCC 0 -5 0 0 10 0 4
a 10-cm high can about the *y*-axis, base plane at *y* = –5 with radius of 4 cm.

TRC: Truncated right-angle cone.

S<n> TRC *Vx Vy Vz Hx Hy Hz R1 R2*

where *Vx Vy Vz* = x,y,z coordinates of bottom of truncated cone
*Hx Hy Hz* = cone axis height vector
*R1* = radius of lower cone base
*R2* = radius of upper cone base

Example: S2 TRC -5 0 0 10 0 0 4 2
A 10-cm high truncated cone about the x-axis with the center of the 4-cm radius base at x,y,z = –5,0,0 and with the 2-cm radius top at x,y,z = 5,0,0.

BOX: General paralellepiped.

S<n> BOX *Vx Vy Vz A1x A1y A1z A2x A2y A2z A3x A3y A3z*

where *Vx Vy Vz* = *x,y,z* coordinates of corner
*A1x A1y A1z* = vector of first edge starting at *Vx Vy Vz*
*A2x A2y A2z* = vector of second edge starting at *Vx Vy Vz*
*A3x A3y A3z* = vector of third edge starting at *Vx Vy Vz*

Example: S3 BOX -1 -1 -1 2 0 0 0 2 0 0 0 2
a cube centered at the origin, 2 cm on a side, sides parallel to the major axes.

RPP: Rectangular parallelepiped, surfaces normal to major axes, *x,y,z* values relative to origin.

S<n> RPP Xmin Xmax Ymin Ymax Zmin Zmax

Example: S4 RPP -1 1 -1 1 -1 1
equivalent to BOX above.

RHP or HEX: Right hexagonal prism.

S<n> RHP *v1 v2 v3 h1 h2 h3 r1 r2 r3*

where *v1 v2 v3* = *x,y,z* coordinates of the bottom of the hex
*h1 h2 h3*= vector from the bottom to the top
for a *z*-hex with height h, *h*1,*h*2,*h*3 = *0 0 h*
*r1 r2 r3* = vector from the axis to the middle of the first facet
for a pitch 2*p* facet normal to *y*-axis, *r1,r2,r3 = 0 p 0*

The remaining five facets are obtained by rotating the first facet around the symmetry axis in 60-degree increments.

Example: S5 RHP 0 0 -4 0 0 8 0 2 0
a hexagonal prism about the *z*-axis whose base plane is at *z* = –4 with a height of 8 cm and whose first facet is normal to the *y*-axis at *y* = 2.

WED: Wedge (a 5-sided polyhedron with two identical parallel triangular bases).

S<n> WED *Vx Vy Vz A1x A1y A1z A2x A2y A2z A3x A3y A3z*

where *Vx Vy Vz* = vertex

*A1x A1y A1z* = vector of first side of triangular base starting at *Vx Vy Vz*
*A2x A2y A2z* = vector of second side of triangular base starting at *Vx Vy Vz*
*A3x A3y A3z* = vector of third edge starting at *Vx Vy Vz* (not required to be normal to the base)

Example: S6 WED 0 0 -6 4 0 0 0 3 0 0 0 12
A 12-cm high right-angle wedge with vertex at x,y,z = 0,0,-6. The triangular base and top are a right triangle with sides of length 4 (*x*-direction) and 3 (*y*-direction) and hypotenuse of length 5.

ARB4: ARBitrary four-sided polyhedron defined in terms of vertex coordinates.

S<n> ARB4 *V1x V1y V1z V2x V2y V2z V3x V3y V3z V4x V4y V4z*
where *V1x V1y V1z* = *x,y,z* coordinates of vertex No. 1
*V2x V2y V2z* = *x,y,z* coordinates of vertex No. 2
*V3x V3y V3z* = *x,y,z* coordinates of vertex No. 3
*V4x V4y V4z* = *x,y,z* coordinates of vertex No. 4

Example: S7 ARB4 1 1 1 1 -1 -1 -1 1 -1 -1 -1 1
a regular tetrahedron centered at the origin, with edge length $2\sqrt{2}$ cm.

ARB: Polyhedron with 4 – 8 corners and 4 – 6 sides defined in terms of triplets of corners.

S<n> ARB *ax ay az bx by bz cx cy cz ... hx by hz N1 N2 N3 N4 N5 N6*

There must be eight triplets of entries input for the ARB to describe the (x,y,z) of the corners, although some may not be used (just use zero triplets of entries). These are followed by six more entries, *N*, which follow the prescription: each entry is a four-digit integer that defines a side of the ARB in terms of the corners for the side. For example, the entry 1278 would define this plane surface to be bounded by the first, second, seventh, and eighth triplets (corners). Since three points are sufficient to determine the plane, only the first, second, and seventh corners would be used in this example to determine the plane. The distance from the plane to the fourth corner (corner 8 in the example) is determined by MCNelectron. If the absolute value of this distance is greater than the "distance precision" parameter (defined using the keyword DISTANCE_PRECISION), a corresponding warning is displayed, but execution of the program is not interrupted. If the fourth digit is zero, the fourth point is ignored. For a four-sided ARB, four nonzero four-digit integers (last digit is zero for four-sided since there are only three corners for each side) are required to define the sides. For a five-sided ARB, five nonzero four-digit integers are required, and six nonzero four-digit integers are required for a six-sided ARB. Since there must be 30 entries altogether for an ARB (or MCNelectron gives an error message), there must be two zero integers for the four-sided ARB and one zero integer for a five-sided ARB.

Example: S8 ARB -5 -10 -5 -5 -10 5 5 -10 -5 5 -10 5 0 12 0 0 0 0 &
0 0 0 0 0 0 1234 1250 1350 2450 3450 0
A five-sided polyhedron with corners at x,y,z = (-5,-10,-5), (-5,-10,6), (5,-10,-5), (5,-10,5), (0,12,0), and planar facets are constructed from corners 1234, etc.

MB: General macrobody defined in terms of surface identifiers and their "senses".

S<n> MB ±Surface_ID1 ±Surface_ID2 ±Surface_ID3 …

The surface identifiers specified in the definition of a general macrobody are interpreted by MCNelectron exactly as in the definition of a cell (see description of the keyword "C<n>" in

Section 4.2). Identifiers of other macrobodies (standard or general) or of their facets can be used in the definition of a general macrobody along with identifiers of "regular" surfaces. As the definitions of cells, the definitions of general macrobodies are checked for consistency, and if it is determined that the volume of the region of space enclosed by a given macrobody is zero, the program quits with a corresponding error message.

Example: S9 MB 3 -5 -2.3
A general macrobody defined as the set of three regions (the third of those regions is the complement of region No. 3 of the definition of macrobody No. 2).

The order of the facet numbering is given below for each type of the standard macrobodies.

| | | |
|---|---|---|
| RCC: | 1 | Cylindrical surface of radius *R* |
| | 2 | Plane normal to end of *Hx Hy Hz* |
| | 3 | Plane normal to beginning of *Hx Hy Hz* |
| TRC: | 1 | Conical surface |
| | 2 | Plane normal to end of *Hx Hy Hz* |
| | 3 | Plane normal to beginning of *Hx Hy Hz* |
| BOX: | 1 | Plane that is not parallel to edge *A1x A1y A1z* and passes through its end |
| | 2 | Plane that is not parallel to edge *A1x A1y A1z* and passes through its beginning |
| | 3 | Plane that is not parallel to edge *A2x A2y A2z* and passes through its end |
| | 4 | Plane that is not parallel to edge *A2x A2y A2z* and passes through its beginning |
| | 5 | Plane that is not parallel to edge *A3x A3y A3z* and passes through its end |
| | 6 | Plane that is not parallel to edge *A3x A3y A3z* and passes through its beginning |
| RPP: | 1 | Plane *Xmax* |
| | 2 | Plane *Xmin* |
| | 3 | Plane *Ymax* |
| | 4 | Plane *Ymin* |
| | 5 | Plane *Zmax* |
| | 6 | Plane *Zmin* |
| RHP or HEX: | 1 | Plane normal to end of *r1 r2 r3* |
| | 2 | Plane opposite facet 1 |
| | 3 | Plane obtained by rotating facet 1 by 60 degrees counterclockwise |
| | 4 | Plane opposite facet 3 |
| | 5 | Plane obtained by rotating facet 3 by 60 degrees counterclockwise |
| | 6 | Plane opposite facet 5 |
| | 7 | Plane normal to end of *h1 h2 h3* |
| | 8 | Plane normal to beginning of *h1 h2 h3* |
| WED: | 1 | Slant plane including top and bottom hypotenuses |
| | 2 | Plane including vectors *V2* and *V3* |
| | 3 | Plane including vectors *V1* and *V3* |
| | 4 | Plane including vectors *V1* and *V2* at end of *V3* (top triangle) |
| | 5 | Plane including vectors *V1* and *V2* at beginning of *V3* (bottom triangle, including vertex point) |
| ARB4: | 1 | Plane defined by vertices *V1*, *V3* and *V4* |
| | 2 | Plane defined by vertices *V1*, *V4* and *V2* |
| | 3 | Plane defined by vertices *V1*, *V2* and *V3* |
| | 4 | Plane defined by vertices *V2*, *V3* and *V4* |

ARB: 1 Plane defined by corners *N1*
2 Plane defined by corners *N2*
3 Plane defined by corners *N3*
4 Plane defined by corners *N4*
5 Plane defined by corners *N5*
6 Plane defined by corners *N6*

The facets of a general macrobody are numbered sequentially, in the order of their appearance in the definition of the macrobody. For example, if the general macrobody is defined as follows:

S3 MB 4 5 2 1

then specifying "-3.2" in the definition of a cell or a general macrobody would have the same effect as specifying "-5".

A sequence of facets of the same macrobody can be written by specifying the macrobody identifier only once and then listing the facet numbers between brackets after the decimal point (if necessary, the minus sign may be specified before the facet number). For example, "3.[2 -4 3]" is equivalent to "3.2 -3.4 3.3". The minus sign may also be specified before the macrobody identifier. The final sign corresponding to an individual facet is the "product" of the signs before the macrobody identifier and before the facet number. For example, "-3.[-2 4 -3]" is equivalent to "3.[2 -4 3]". This notation may be used in definitions of cells and general macrobodies, as well as in definitions of cell-entry tallies (see Section 4.7.2).

The redundant bounding surfaces of a general macrobody are removed during the initial checking of the geometry (similarly to the redundant bounding surfaces of a cell, as mentioned in Section 4.2). The removal of a surface from the definition of a general macrobody affects the sequence numbers of its remaining facets that were listed after the removed surface in the original definition of the macrobody. If a bounding surface is removed, a warning message is displayed.

## *4.4. Using coordinate transformations in MCNelectron*

### *4.4.1. Definition of a coordinate transformation in the MCNelectron input file*

The coordinate transformations can be used in MCNelectron in two ways:

(a) as a method to create a new cell or surface by rotating and translating an existing cell or surface;

(b) as a method to define an auxiliary right-hand Cartesian coordinate system for a subsequent definition of a new surface in this auxiliary coordinate system (this is useful if the definition of the surface in the auxiliary system is much simpler than its definition in the primary system).

Accordingly, the parameters specified in the definition of each coordinate transformation can be interpreted either as the translation vector and the rotation angles applied to the original cell or surface, or, equivalently, as the position of the origin and the directions of the axes of the auxiliary coordinate system. Each coordinate transformation is defined independently of its intended use, and it can be subsequently used in any of the above-mentioned ways as many times as needed. However, for the purpose of describing the definition format, the second interpretation is more convenient: the definition of the coordinate transformation specifies the position of the origin of the auxiliary coordinate system and the directions of its axes. The latter directions are completely defined by the *x* and *y* direction vectors (the *z* direction vector is obtained as the vector product of the *x* and *y* direction vectors). It is possible to omit the *y* direction vector from the definition (then it will be calculated automatically so that it is normal to the user-specified *x* direction vector, and the missing rotation angle will be chosen on the basis of arbitrary criteria). The *x* and *y* direction vectors may be defined either in terms of their Cartesian components, or in terms of the corresponding angles between the direction vector and the axes of the primary coordinate system, or in terms of the azimuth and elevation angles of the direction vector. All those parameters may be optionally defined not in the main (original) coordinate system, but in the auxiliary coordinate system

corresponding to a previously-defined coordinate transformation (this makes it possible to "chain" multiple coordinate transformations).

Each coordinate transformation is defined using the following format:

<*>TR<n> <TR<m>> $O_x$ $O_y$ $O_z$ $X_x$ $X_y$ $X_z$ $Y_x$ $Y_y$ $Y_z$

where the following notations are used:

- the optional asterisk before the initial "TR" is used to indicate that the Cartesian components of the direction vectors are replaced by the corresponding angles (in degrees),
- <n> is a positive integer number (the identifier of the coordinate transformation),
- "TR<m>" is optional and indicates the auxiliary coordinate system corresponding to a previously-defined coordinate transformation (for "chaining" multiple coordinate transformations),
- $O_x$, $O_y$ and $O_z$ are the coordinates of the origin of the auxiliary coordinate system (in centimeters),
- $X_x$, $X_y$ and $X_z$ are the components of the $x$ direction vector of the auxiliary coordinate system,
- $Y_x$, $Y_y$ and $Y_z$ are the components of the $y$ direction vector of the auxiliary coordinate system.

The direction vectors can be of any non-zero length. If the user-specified $y$ direction vector is not normal to the $x$ direction vector, then the $y$ direction vector will be automatically rotated in the plane parallel to the specified $x$ and $y$ vectors to make it normal to the $x$ direction vector (in this case, a warning is displayed).

For example, the following two equivalent directives define the auxiliary coordinate system with the origin at the point (1, 2, 3) and the $x$ and $y$ direction vectors obtained by rotating the corresponding vectors of the primary system around the $Z$ axis of the primary system by 45 degrees counter-clockwise:

```
TR10    1 2 3    1 1 0    -1 1 0
*TR10    1 2 3    45 45 90    135 45 90
```

As illustrated by the second definition in the above example, the angles must be specified in degrees. The allowed values of each angle are from −720° to +720°. As mentioned, the last three numbers (corresponding to the $y$ direction vector) may be omitted.

The three components of the $x$ and $y$ direction vectors may be replaced with the azimuth and elevation angles. Their meaning is illustrated in Fig. 4.2. The azimuth angle ($\alpha$) is the angle between the $X$ axis of the primary system and the projection of the direction vector of the auxiliary system to the $XY$ plane of the primary system. The elevation angle ($\beta$) is the angle between the $XY$ plane of the primary system and the direction vector of the auxiliary system. The values of $\alpha$ and $\beta$ must be given in degrees. The allowed values of $\alpha$ are from −720° to +720°. The allowed values of $\beta$ are from −90° to +90°. The two definitions below are equivalent to the two definitions above:

```
TR10    1 2 3    45 0    -1 1 0
*TR10    1 2 3    45 0    135 45 90
```

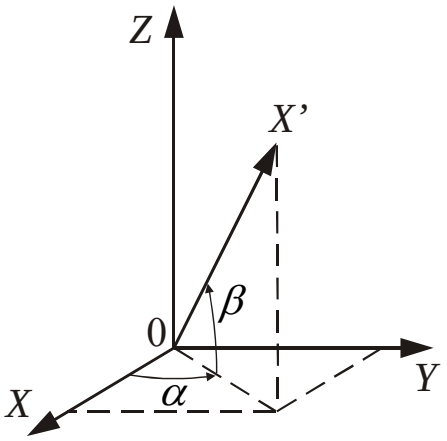


Fig. 4.2. The azimuth ($\alpha$) and elevation ($\beta$) angles that can be optionally specified instead of the components of the $x$ and $y$ direction vectors of the auxiliary coordinate system.

In this case, the initial asterisk has no effect on the interpretation of numerical entries No. 4 and 5 of the definition (those entries will be interpreted as the azimuth and elevation angles in degrees, regardless of whether the asterisk is present or not). If the $x$ direction vector is defined in terms of its azimuth and elevation angles, then it is possible to define the $y$ direction vector similarly. For example, the following definition is equivalent to the previous definitions:

```
TR10    1 2 3    45 0    135 0
```

In this case, the numerical entries No. 6 and 7 have the meaning of the azimuth and elevation angles of the $y$ direction vector of the auxiliary coordinate system (in degrees), and the asterisk before "TR" would have no effect.

The correct interpretation of the definition of a coordinate transformation is determined by MCNelectron from the number of numerical entries in the definition: if that number is 6 or 9, then entries No. 4, 5 and 6 will be interpreted as the Cartesian components of the *x* direction vector or the corresponding angles, and if there are 5, 7 or 8 numerical entries, then entries No. 4 and 5 will be interpreted as the azimuth and elevation angles of the *x* direction vector. If there are 8 or 9 numerical entries, then the last three of them be interpreted as the Cartesian components of the *y* direction vector or the corresponding angles, and if there are 7 numerical entries, then the last two of them will be interpreted as the azimuth and elevation angles of the *y* direction vector. It is also possible to define a "pure" translation (without rotation) by specifying only the three initial numerical entries. Thus, the total number of numerical entries in the definition of a coordinate transformation must be 3, 5, 6, 7, 8 or 9.

When a coordinate transformation is used for creating a copy of an existing cell or surface, it is more convenient to interpret it as a sequence of one rotation around the origin and one translation (in that order). This interpretation can be also applied for the case of defining an auxiliary coordinate system: the latter system is a "copy" of the primary coordinate system obtained by rotation around the origin and translation. In this case, it is important to keep in mind that the rotation is always around the origin of the primary coordinate system and that it always precedes the translation. If it is necessary to rotate an object around the point that is not at the origin, then the translation vector must be calculated using trigonometric formulas with the position of the rotation center, the rotation angle and the rotation axis as input. For example, the following directive defines counter-clockwise rotation by 45° in the *XY* plane around the point (0, 1, 0):

```
TR2    0.70710678  0.29289322  0    1 1 0    -1 1 0
```

In this example, the origin of the auxiliary coordinate system is at the point $\left(1/\sqrt{2},\ 1-1/\sqrt{2},\ 0\right)$. It is sometimes possible to avoid the trigonometric calculations of the translation vector by combining two or more coordinate transformations, whose translation vectors do not require any calculations. In the last example, the coordinate transformation "TR2" can be replaced by the sequence of these two transformations:

1) rotation around the origin by 45° and translation by (0, 1, 0);

2) pure translation in the transformed coordinate system by (0, –1, 0).

This sequence of two transformations is defined by the following two directives:

```
TR21    0 1 0    1 1 0    -1 1 0
TR22    TR21     0 -1 0
```

The last directive includes an additional entry "TR21", which indicates that the subsequent numerical entries in this directive are defined not in the main (original) coordinate system, but in the coordinate system obtained by applying transformation "TR21" to the main coordinate system. Such "chained" transformations can be read in another way, which does not require keeping in mind the axis directions of the auxiliary coordinate systems (changing with each transformation in the chain): all coordinate transformations in the chain can be assumed to be defined in the *main* coordinate system, if the chain is read in reverse order (bottom to top). In the same example, such an approach would yield the following sequence:

1) translation by (0, –1, 0) in the main coordinate system,

2) rotation by 45° around the origin of the main coordinate system and translation by (0, 1, 0).

In this method of reading a chain of coordinate transformations, all translation vectors are defined in terms of the Cartesian coordinates of the main system, and all rotations are around the origin of the main system. However, the center of rotation (for all rotations that are done in a given chain of coordinate transformations) can be easily changed: it is sufficient to add its coordinates to the translation vector of the first transformation of the chain and add the opposite pure translation at the end of the chain (see the last example).

*4.4.2. Specifying the identifiers of coordinate transformations in definitions of surfaces and cells*

The auxiliary coordinate system corresponding to a defined coordinate transformation (see Section 4.4.1) can be specified in the definition of a surface or a macrobody by inserting its identifier immediately after the "S<n>" keyword, before the specification of the surface type (see also Sections 4.2 and 4.3). For example, the following directive means that the right circular cylinder is defined in terms of the coordinate axes of the auxiliary coordinate system defined by the "TR4" directive:

```
S1   4   RCC   0.5 2 3   1 0 0   0.2
```

This method of defining new surfaces cannot be applied to general macrobodies.

A new surface or a macrobody (standard or general) can be defined as a rotated and translated copy of a previously-defined surface or a macrobody. In this case, there are only two entries after the "S<n>" keyword: the identifier of the coordinate transform and the identifier of the original surface (or a macrobody). For example, the following directive specifies that surface "S4" is obtained by applying transformation "TR10" to the previously-defined surface with identifier 3:

```
S4   10   3
```

A macrobody is transformed by creating a copy of each facet (i.e., the constituent surface) of the original macrobody. The constituent surfaces of a transformed general macrobody are assigned the ID of zero and consequently those surfaces cannot be referenced in definitions of cells, macrobodies and cell-entry tallies by specifying a single number (unlike the constituent surfaces of the original general macrobody). The constituent surfaces of a transformed general macrobody can be referenced only by using the ID of the transformed macrobody and the facet sequence number separated by the decimal point. For example, if "3" in the last example is the ID of a general macrobody defined as a set of three surfaces, then the individual facets of the transformed macrobody "4" can be specified as "4.1", "4.2" and "4.3".

A new cell can be defined as a rotated and translated copy of a previously-defined cell. In such a case, the set of all surface identifiers (excluding the ones specified after the complement operator "#") in the definition of the cell must be replaced by two entries, which have the format "TR<n> CellID", where "<n>" is the identifier of the coordinate transformation, and "CellID" is the identifier of the original cell. For example, the following directive defines a cell with ID 6, which is obtained by applying the coordinate transformation "TR2" to the cell with ID 3:

```
C6   1   -11.34   TR2 3
```

As in the case of transformations of macrobodies, a cell is transformed by creating a copy of each facet (i.e., the bounding surface) of the original cell. If the definition of the original cell includes the complement operator "#", then the cells specified after this operator will be ignored. I.e., the transformed cell will not have any excluded regions corresponding to the contained cells. In order to add them, the contained cells must be transformed and copied separately (using the transformation ID specified in the definition of the "host" cell) and then listed after the complement operator in the definition of the host cell, e.g.,

```
C6   1   -11.34   TR2 3 #4 #5
```

In this example, cell No. 6 is constructed by creating a copy of each surface listed in the definition of cell No. 3 (excluding the surfaces specified after the complement operator "#"). If the definition of cell No. 3 includes the complement operator and if cell No. 6 must be an exact copy of cell No. 3, then all cells whose complements are included in the definition of cell No. 3 must be transformed separately and then their copies must be listed in the definition of cell No. 6 after the complement operator. In the above example, there are two such cells (with IDs 4 and 5).

The bounding surfaces of a cell obtained by transforming another cell cannot be referenced in definitions of cells and macrobodies (unless they coincide with other surfaces, which were defined separately). However, those bounding surfaces can be referenced in cell-entry tallies by specifying the identifier of the original surface (see Section 4.7.2 for details).

If the simulation setup contains many copies of the same geometrical object, then the number of coordinate transformations can become too large to enter all of them manually. For

example, the facets of the polyhedron shown in Fig. 4.3 are created by changing the azimuth angle of the plane's normal vector from 0° to 330° in 30-degree increments, and the elevation angle from –75° to 75° (also in 30-degree increments). Since the number of the azimuth angle values is 12 and the number of the elevation angle values is 6, the total number of facets is 12 · 6 = 72. The facets of this polyhedron can be defined as follows: first, define 72 auxiliary coordinate systems by rotating the *X* axis of the main system as described above, then define a plane normal to the *X* axis of each auxiliary coordinate system at a fixed distance from the origin. Thus, 72 "TR<n>" directives and 72 "S<n>" directives are needed. Those directives are given below:

```
TR1    0 0 0   0   -75
TR2    0 0 0   30   -75
…
TR12   0 0 0   330   -75

TR13   0 0 0   0   -45
TR14   0 0 0   30   -45
…
TR24   0 0 0   330   -45
…
…
…
TR61   0 0 0   0   75
TR62   0 0 0   30   75
…
TR72   0 0 0   330   75

S1   1   P   1 0 0   -1 0 0
S2   2   P   1 0 0   -1 0 0
…
S72   72   P   1 0 0   -1 0 0
```

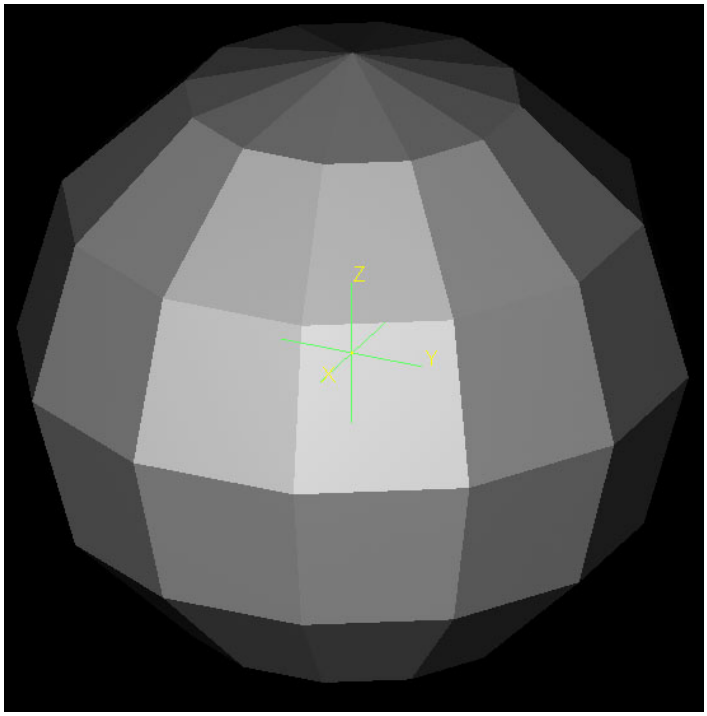


Fig. 4.3. An example of a 3D object requiring many repetitive coordinate transformations (this polyhedron has 72 facets).

In such cases, the number of lines in the MCNelectron input file can be decreased significantly by replacing the input directives with a program written in the MCNEcode programming language, whose syntax is similar to the syntax of the C language. Sequences of repetitive directives can be replaced by a "for" loop. For example, the code that is equivalent to the sequence of the "TR" and "S" directives given above looks like this:

```
program Planes
k = 1
for (j = 0; j < 6; j = j + 1) {
        for (i = 0; i < 12; i = i + 1) {
                dir("TR", k, "   0 0 0   ", i * 30, "   ", -75 + j * 30)
                k = k + 1
        }
}
for (i = 1; i <= 72; i = i + 1) dir("S", i, "   ", i, "   P   1 0 0   -1 0 0")
end program
```

MCNelectron interprets such blocks of code embedded in the input file as sequences of input directives, where each directive corresponds to a call to the built-in function "dir". There may be multiple code blocks, with "regular" input directives in between them. For example, the cell shown in Fig. 4.3 can be defined by adding the following lines after the code of the previous example:

```
M1 "Z = 82"    82        1
program C1
dir("C1  1  -11.34  &")
for (i = 1; i < 72; i = i + 1) dir(i, " &")
dir(72)
end program
```

MCNEcode has many other features, such as various built-in functions (there are 70 built-in functions in total, including mathematical functions, data output functions and others) and the support for user-defined data arrays and subroutines. A short overview of procedural generation of MCNelectron input directives is presented in Section 4.12. A detailed description of all syntax elements of the MCNEcode programming language and the built-in functions is in the Appendix.

### *4.5. Usage of voxels in MCNelectron*

Further on, "voxels" will be defined as rectangular volume elements with edges formed by intersections of coordinate planes, i.e., three sets of parallel planes that are normal to the three Cartesian coordinate axes (some of those three sets of planes may be empty). Voxels fill entire space: the limiting voxels are infinite along one, two or three axes (if there are no coordinate planes defined, then there is a single "voxel" filling entire space). If there is a sufficiently large number of voxels in the regions of space with a large number of bounding surfaces, then the number of surfaces that have to be checked for intersection with the particle's track may be reduced significantly, because only the surfaces intersecting with the current voxel have to be checked (see the end of this Section for detailed information about the data associated with each voxel).

The total number of voxels is obtained by adding 1 to the total numbers of planes normal to each of the three axes and then multiplying the three numbers obtained. For example, in the case of the sample simulation setup corresponding to the MCNelectron input file given in Section 4.2 and illustrated in Fig. 4.1, there are 200 voxels: (10 *X*-intervals) · (10 *Y*-intervals) · (2 *Z*-intervals) = 200. If there are adjacent voxels that do not intersect with any cells or that are entirely contained in the same cell (the "host cell"), then they are merged into larger rectangular voxels, in an attempt to minimize the number of voxels by maximizing their volume. This reduces the memory usage and the number of intersections with voxel faces that have to be calculated when tracking the particles.

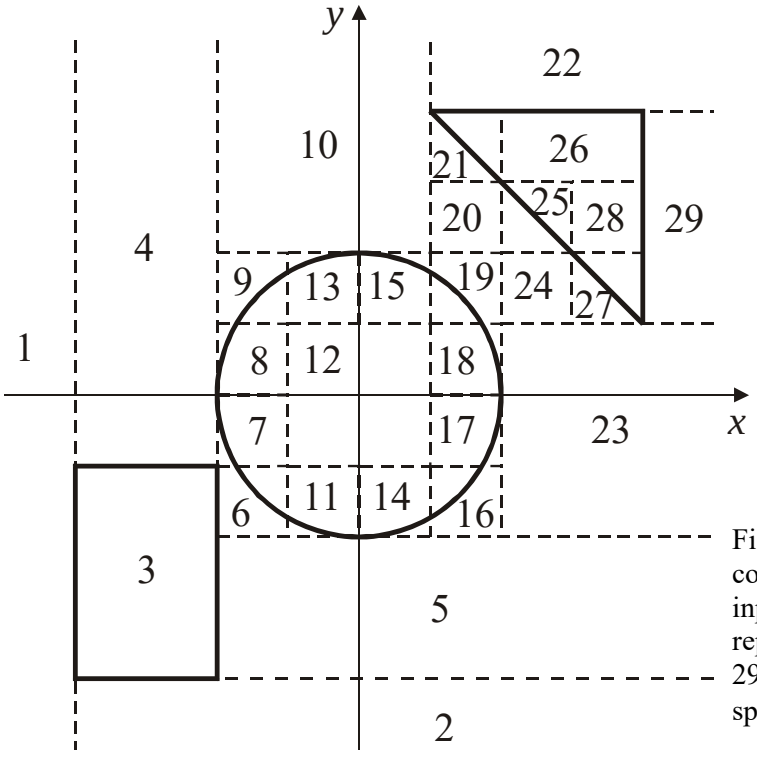


Fig. 4.4. The voxel configuration of the system corresponding to the sample MCNelectron input file given in Section 4.2. The dashed lines represent the bounding planes of voxels No. 1 – 29 (voxel No. 0, which occupies the entire half-space $z < -100$, is not shown).

In the same example, the final number of voxels after merging the initial voxels is 30. One of those voxels (voxel No. 0) is obtained by merging 100 voxels with $z$ coordinates less than −100; it occupies the entire half-space $z < -100$. The remaining 29 voxels (No. 1 – 29) are shown in Fig. 4.4 by dashed lines. Obviously, voxel No. 3 is obtained by merging six initial voxels and occupies the entire volume of the cell with rectangular cross-section (this cell is the host cell of that voxel). Voxels No. 12 and No. 26 are obtained by merging some of the internal voxels of the other two cells. Voxels No. 1, 2, 4, 5, 10, 22, 23, 29 are obtained by merging the "empty" voxels, which do not intersect with any cells. The remaining voxels cannot be merged, because they do not belong to any of those two categories (i.e., they intersect with some of the bounding surfaces), or because it is not possible to form a larger rectangular region by merging those voxels with adjacent voxels that are contained in the same cell or are empty.

Each voxel has a data structure associated with it (type VOXEL1 as defined in the MCNelectron source code files). The fields of that data structure have the following meaning:

1) the sequence number of a cell that contains the given voxel in its entirety (the "host cell"), or a negative number, if there is no such cell (if such a cell exists, then the following entries are undefined, except in the case when at least one face of the voxel coincides with one of the bounding surfaces of that cell),
2) the list of surfaces that simultaneously intersect with the given voxel (or are entirely contained in it) and belong to cells intersecting with that voxel or entirely contained in it (surfaces that coincide with a face of the voxel and belong to cells intersecting with that voxel or entirely contained in it are treated as intersecting with it),
3) for each of the surfaces mentioned in "2" – two lists of cells using that surface in the cell definition and intersecting with that voxel or contained in it (one list for each of the two regions associated with that surface). The value of each element in those lists is equal to the sequence number of the corresponding cell in the overall list (defined in "4" below),
4) the list of cells that simultaneously intersect with the given voxel (or are entirely contained in it) and have surfaces that also intersect with that voxel or are entirely contained in it,
5) for each of the cells mentioned in "4" – the list of its surfaces intersecting with that voxel or contained in it, and the corresponding list containing the values of a sense of each region relative to the bounding surface (±1),
6) for each of the cells mentioned in "4" – two arrays of unsigned 16-bit integers with each bit position coinciding with the sequence number of the corresponding surface of the list defined in "2" (the number of elements in each array is such that the total number of bits is greater than or equal to the number of surfaces defined in "2"). In the first array (the "cell mask array"), the bits corresponding to the used surfaces are set to 1, and the bits corresponding to the unused surfaces are set to 0. In the second array (the "cell position array"), the bits corresponding to the regions with positive sense are set to 1, and all other bits (i.e., those corresponding to the regions with negative sense and the unused surfaces) are set to 0.

After a particle escapes a cell, the sequence of actions is the following (for greater simplicity, the description presented below assumes that the cell definitions do not use the complement operator):

a) Determine the voxel containing the escape point.
b) Determine if a cell defined in "1" exists (such a cell may exist if the current cell is used in a definition of a union of two or more intersecting cells). If such a cell exists, then it is designated as the current one, and transport continues in the same material as before. Otherwise, go to "c".
c) Determine the position of the current point relative to each of the surfaces defined in "2" (if the point happens to be exactly on the surface, then the position is determined from the angle between the particle's momentum and the normal of the surface). The result of this step is stored in an array of unsigned 16-bit integers having the same format as described in "6" (the

"point position array"). If the current point is in the positive direction relative to a particular surface, then the corresponding bit is set to 1; otherwise it is set to 0.

d) For each of those surfaces, all cells defined in "3" and on the same side of that surface as the current point are checked for the presence of the current point inside them. This is done for each of those cells as follows. First, the corresponding cell mask array is bitwise multiplied by the point position array (obtained in "c"). Next, the array obtained as a result of that multiplication is compared with the cell position array. If the latter two arrays are identical, this means that the current point is inside the corresponding cell. If such a cell is found, then it is designated as the current one, and transport continues in that cell.

e) Otherwise, find the nearest (or next) point of intersection of the particle's track with the surfaces defined in "2" or with the voxel face.

f) If the point found in "e" is an interior point of the voxel (i.e., lies on the surface of one of the cells defined in "3"), then update the particle position relative to that surface (i.e., "flip" the corresponding bit of the point position array defined in "c") and go to Step "d" (however, this time check not all the cells, but only the cells corresponding to the crossed surface and on the same side of it as the particle, i.e., only the cells included in one of the lists defined in "3").

g) If the point found in "e" is on the voxel boundary, update the current voxel number and go to "c".

h) If there is no cell or voxel boundary in the direction where the particle is moving to (this is possible only when the particle is in one of the limiting voxels, which are infinite along one, two or all three Cartesian axes), this means that the particle escaped from the system.

### *4.6. Output of particle track data*

MCNelectron can output coordinates and energies of particles after each collision. A set of collision points defines a so-called "track" of a particle. Each track is represented by a piecewise-linear curve whose initial point coincides with the point of emission of a source particle (if that point is inside a cell) or with the point of entry of a source particle into a cell (if the emission point is not inside a cell), or with the point of creation of a secondary particle, and each subsequent vertex is a point of a collision. If a collision involves emission of a secondary particle, then the current track is continued by following the primary particle, and an additional track corresponding to the emitted secondary particle is started. The primary particle is defined as follows:

1. In the case of ionization by an electron or positron impact, the primary electron (or positron) is the one with the higher energy.
2. In the case of emission of a bremsstrahlung photon, the primary particle is the electron or the positron.
3. In the case of incoherent scattering of a photon, the primary particle is the scattered photon.

The track is terminated when the particle is absorbed or when it escapes from the system. An absorption event can be of the following types:

1) an energy-loss event resulting in a primary particle with energy less than the cutoff energy,
2) a photoelectric absorption of a photon,
3) a pair production event,
4) a positron annihilation.

Output of particle tracks is controlled by the following keywords:

NTRACKS is used to define the maximum number of tracks to output (if the number following this keyword is zero, then no tracks will be output). Here, the term "track" means the entire electron-photon "shower" created by a single source particle. Thus, the actual number of tracks (including the tracks of all secondary particles) in the track data file can be greater than the number specified after "NTRACKS" by several orders of magnitude. Further on, the term "history" will be used when referring to a single electron-photon shower, and the term "track" will be used when referring to the trajectory of a single tracked particle;

NCOLLISIONS_TRACK is used to define the maximum number of collisions per history. If that number is exceeded, then all unfinished tracks in that history will be "truncated";

SKIP_TRACKS_WITHOUT_COLLISIONS is a "switch" that is used to suppress output of track data for histories without collisions (i.e., the histories corresponding to the source particles that did not interact with any atom). If the integer number following this keyword is non-zero, then such tracks will not be output, otherwise they will be output. In the latter case, the track data for that history will contain only two points: the initial point and the escape point;

MAXGEN_TRACKS controls the sequence number of the last generation of secondary particles to be included in the created track dataset. The source particles belong to generation No. 0. Every secondary particle created by a source particle belongs to generation No. 1. Every secondary particle created by a first-generation particle belongs to generation No. 2, etc. If the number specified after this keyword is -1, then the generation number will not be limited;

CHUNKSIZE_TRACKS controls the computer memory usage when creating a track dataset. The track data are split into smaller arrays ("chunks"): a new "chunk" is allocated when needed. The number of vertices in a single chunk must be specified after this keyword.

The track data are placed into a file that is in the subfolder created in the same folder as the main output file (the subfolder name format is "Files_<output_file_name>"). The name of the file with track data is obtained by appending "_tracks" to the name of the main output file. In the track data file, all tracks are arranged by the history number, and all tracks of the same history are arranged by the generation number. The first track for each history is the source particle track. All first-generation tracks of each history are in chronological order relative to each other. However, the higher-generation tracks are in chronological order only relative to the tracks of the particles that were created by the same previous-generation particle. Those tracks may be interspersed with tracks of particles created by other previous-generation particles, and may be not in chronological order relative to the latter tracks.

The set of data for each track consists of the following components:

1) A line with the overall sequence number of the track and the information about the type of the track. This line starts with the track sequence number, which is followed by a colon. If the track corresponds to a source particle (i.e., the start of a new history), then the sequence number of the history is given after the colon using the format " History No. <n>" (for example, "3312: History No. 2"). The mentioned history number does not take into account the source particles that may have been skipped due to absence of collisions (if some of the source particles were skipped, then the history number may be less than the original sequence number of the source particle). If the track corresponds to a secondary particle, then the particle's generation number and the type of the track (indicating both the type of the particle and the type of the event where it was created) are specified after the colon using the format "<generation No.>/<type No.>" (for example, "3313:1/7"). The type number is from 1 to 11, and it has the following meaning:
    - 1 – bremsstrahlung photon,
    - 2 – positron annihilation photon,
    - 3 – X-ray photon emitted after ionization by electron or positron impact,
    - 4 – X-ray photon emitted during atomic relaxation after photoelectric absorption (X-ray fluorescence) or after incoherent scattering (Compton fluorescence),
    - 5 – electron from pair production,
    - 6 – positron from pair production,
    - 7 – Compton recoil electron,
    - 8 – photoelectron,
    - 9 – Auger electron emitted after photoelectric absorption or incoherent scattering,
    - 10 – Auger electron emitted after ionization by electron or positron impact,
    - 11 – knock-on electron;

2) For source particle tracks – a line with the original sequence number of the source particle (as mentioned above, it may be greater than the sequence number of the history). For secondary particles, this line is absent;
3) A line with the number of vertices of that track. It is equal to the number of lines defining the current track. Thus, it includes not only the internal nodes, but also the initial and final points of each track;
4) The lines with vertex coordinates and energies (one line per vertex). Each line contains four numbers: the *x*, *y* and *z* coordinates of the vertex (in centimeters), and the particle's energy after the corresponding event (in electronvolts).

The initial point of a track of a source particle is the point where that particle entered a cell of the geometry (this point may be different from the source point, if the latter is not inside any cell). The initial point of a track of a secondary particle is the point where the particle was created. This point always coincides with one of the vertices of one of the previous-generation tracks of the same history.

For absorption tracks, the final line contains the coordinates of the point where the particle's energy dropped below the low-energy cutoff. The distinguishing feature of absorption tracks is the zero final energy. The distinguishing feature of truncated (i.e., unfinished) tracks is a positive final energy. For escape tracks, the final line contains the coordinates of the escape point (i.e., the point where the particle crossed the boundary of a cell for the last time on its way out of the system), and the final energy is output with the minus sign (it is the distinguishing feature of escape tracks).

When a history is truncated, a warning message is displayed. There are in general more than one truncated tracks in that history. Some of them may have only one vertex (the initial point).

If there are histories without collisions and if such histories are required to be excluded from the track dataset, then the set of included histories may depend on the number of CPU threads used during the simulation, because in the case of multiple threads some of the histories with collisions may be excluded, too. In order to determine whether a track dataset corresponding to the current history should be created, MCNelectron uses only the number of histories that were excluded (due to absence of collisions) *in the same thread only*. This lack of communication between threads makes it difficult to take into account the histories that were excluded in other threads.

Track data cannot be output if interaction forcing is used (see Section 4.8). Track data are not stored in CUDA devices. Consequently, in order to output track data in a simulation that uses CUDA devices, a part of the simulation must be performed by the CPU (the track data will be created only in the simulation threads running on the CPU). If some of the CUDA devices are emulated, then the last few tracks that are written to a file may be different from the tracks written when those devices are not emulated ("emulation" of CUDA devices is explained in Section 4.10).

The default values of the parameters related to output of track data are given below:

```
NTRACKS       0
NCOLLISIONS_TRACK       100000
SKIP_TRACKS_WITHOUT_COLLISIONS  1
MAXGEN_TRACKS   -1
CHUNKSIZE_TRACKS 10000
```

### *4.7. Usage of tallies in MCNelectron*

MCNelectron can calculate three types of tallies. Each of them is described below, together with the corresponding keywords.

#### *4.7.1. Plane-crossing tallies*

MCNelectron can calculate tallies of electrons or photons that have crossed a user-defined "tallying plane" or a set of parallel equidistant tallying planes during the simulation. The tally parameter (i.e., the variable whose value determines the sequence number of the incremented bin of the tally) can be any one of those seven quantities:

1) the particle energy,
2) the angle between the plane's normal and the particle's momentum vector,
3) the cosine of the angle between the plane's normal and the particle's momentum vector,
4) the radial coordinate of the crossing point on the tallying plane,
5) the azimuth angle of the crossing point on the tallying plane,
6) the $X'$ coordinate of the crossing point on the tallying plane,
7) the $Y'$ coordinate of the crossing point on the tallying plane.

The latter four variables are defined in the tallying plane's coordinate system, whose axes are in general different from the axes of the system that is used for specifying positions of the cells and the source geometry. That is why there is the prime symbol in notations $X'$ and $Y'$. It is also possible to calculate "two-dimensional" tallies, where each bin corresponds to a pair of intervals of any two variables (for example, the particle energy and the radial coordinate). All bins are of equal width.

The specification of a plane-crossing tally contains notations of ranges of the variables that must be used for calculating the "scoring" bin. Each of those ranges must be defined before the specification of the corresponding tally in the input file. Definition of a range must begin with any one of those seven keywords:

"E", "THETA", "MU", "R", "PHI", "X", or "Y",

or the same keyword with an appended positive integer number (for example, "E1", "MU10", etc). This keyword defines the meaning of the variable (the energy, the incident angle, the cosine of the incident angle, the radial coordinate, the azimuth angle, the $X'$ coordinate, or the $Y'$ coordinate, respectively). After that keyword, there must be three space-delimited numbers: the lower bound of the range, the upper bound of the range, and the number of bins. For example, the definition of an energy range consisting of 10 equidistant bins from 1 MeV to 2 MeV might look like this:

```
E1   1 2 10
```

In this example, the first bin corresponds to energies from 1.0 MeV to 1.1 MeV, and the last bin corresponds to energies from 1.9 MeV to 2.0 MeV. The unit of energy is defined using the keyword "E_UNIT" (see Section 4.1). The values of the incident angle "THETA" and the azimuth angle "PHI" must be given in degrees. The maximum allowed number of ranges of each type is 100.

The specification of a plane-crossing tally for electrons or positrons must begin with the keyword "ETALLY", or the same keyword with an appended positive integer number (for example, "ETALLY2"). The specification of a plane-crossing tally for photons must begin with the keyword "PTALLY", or the same keyword with an appended positive integer number. The specification of a "mixed" (electron-photon) plane-crossing tally must begin with the keyword "TALLY", or the same keyword with an appended positive integer number (for example, "TALLY2"). The maximum allowed number of directives of each of those three types is 100. The mentioned keyword must be followed by six space-delimited numbers: three coordinates of a point on the tallying plane (in cm) and three components of the normal vector of the tallying plane (that vector may be of any non-zero length). Those numbers must be followed by space-delimited notations of ranges that have been previously defined in the input file. An example of the simplest possible plane-crossing tally:

```
ETALLY1   0 0 0   0 0 1   E1
```

The above example defines an energy tally of electrons or positrons crossing the $XY$ plane. During the simulation, each bin count will be incremented when an electron or a positron crosses that plane and it is determined that the value of the control variable (energy in the above example) is less than or equal to the upper limit of that bin and greater than the lower limit of that bin. If the control variable is not the incident angle or its cosine ("THETA" or "MU", respectively), the count is increased by 1 when the tallying plane is crossed in the direction of its normal vector, and decreased by 1 when the tallying plane is crossed in the opposite direction. If the control variable is the incident angle or its cosine, then the count is always increased by 1.

It is also possible to specify the surface identifier instead of the six mentioned numbers in the definition of a plane-crossing tally. The surface must be a plane, and its identifier must be appended to 'S'. For example, the following definition of a plane-crossing tally

```
ETALLY1   S2   E1
```

is equivalent to the previous definition, if surface No. 2 has been defined as follows:

```
S2   P   0 0 0   0 0 1
```

In order to change the direction of the normal vector of a previously-defined plane, the minus sign must be inserted before "S", for example:

```
ETALLY1   -S2   E1
```

In order to specify a set of parallel equidistant tallying planes, two additional space-delimited numbers must be inserted before the list of ranges: the number of planes and the inter-plane distance (in cm). For example, the following line defines 100 tallies, each corresponding to one of 100 planes, which are at a distance of 0.1 cm from each other:

```
ETALLY1   0 0 0   0 0 1   100 0.1   E1
```

In this case, the first three numbers define a point that is on the *first* plane in the set. All other planes in the set are "stacked" in the direction of the normal vector.

It is possible to group any two ranges in the specification of a plane-crossing tally by enclosing their notations with parentheses, for example:

```
ETALLY1   0 0 0   0 0 1   100 0.1   E1   (X1 Y1)   (E1 MU2)
```

Each such pair of ranges corresponds to a two-dimensional tally, which is represented by a table with rows numbered by the first (or "primary") variable of the pair, and with columns numbered by the second (or "secondary") variable of the pair. There can be up to 20 one-dimensional tallies and up to 20 two-dimensional tallies defined in a single specification of a plane-crossing tally. Thus, the maximum possible number of tallies defined in a single specification of a plane-crossing tally is 40 (multiplied by the number of planes in the set of equidistant planes). The maximum number of equidistant planes that can be specified in a single directive is 1000.

Any of the mentioned 7 variables can be limited by specifying its minimum and maximum values in the definition of a plane-crossing tally. This is achieved using the keyword "LIM", followed by notation of the variable ("E", "THETA", "MU", "R", "PHI", "X" or "Y"), followed by two space-delimited values of the two bounds. Those four components may be inserted anywhere after the geometric parameters (i.e., after the definition of the plane and the two optional numbers defining a set of equidistant planes). For example, in the case of the following tally, "scoring" will be possible only for electrons and positrons that cross a ring-shaped area with inner and outer radii equal to 10 cm and 12 cm, respectively, and with energies between 0.05 MeV and 0.1 MeV:

```
ETALLY1   0 0 0   0 0 1   100 0.1   LIM R 10 12   (MU1 PHI1)   LIM E 0.05 0.1
```

Those limits apply to all tallies defined on the same line of the input file. The lower bound is excluded from allowed values, whereas the upper bound is included in allowed values.

Usage of the variables "R", "PHI", "X", and "Y" requires defining the auxiliary coordinate system on the tallying plane (i.e., position of the origin and directions of the coordinate axes $X'$ and $Y'$). If the tallying plane is defined explicitly (by six number, as in the last example), then the origin of the auxiliary coordinate system is the point specified by the first three numbers in the definition of a plane-crossing tally (or the projection of that point for other planes in the set). If a tallying plane is defined by specifying the surface identifier, then the origin depends on the method used for defining the plane in the directive that defines the surface (see Section 4.2). If the plane has been defined by specifying the point on the plane and the direction of the normal vector, then the mentioned point is the origin of the auxiliary coordinate system. If the plane has been defined by the four coefficients of the plane equation, then the origin of the auxiliary coordinate system is the projection of the origin of the main coordinate system to the plane. If the tallying plane has been defined by three points on the plane, then the origin of the auxiliary coordinate system is the first one of them. If the tallying plane is not normal to the $X$ axis of the "main" coordinate system, then the direction of the $X'$ axis is the same as the direction of the projection of the "main" $X$ axis to the tallying plane. This means that if the tallying plane is not parallel to the $X$ axis, then the direction of the $X'$ axis is the same as the direction of the vector with components ($A'$, $B$, $C$), where $A' = A - (1 / A)$ and $A$, $B$, $C$ are the components of the tallying plane's normal vector with length 1. If the tallying plane is normal to the $X$ axis of the main coordinate system, then the $X'$ axis has either the same direction as the $Y$ axis of the main coordinate system (if the normal is directed in the

positive *X* direction), or opposite to the *Y* axis (if the normal is directed in the negative *X* direction). In any case, the direction of the *Y'* axis on the tallying plane is such that the *X'* axis, the *Y'* axis and the normal vector of the tallying plane are oriented relative to each other according to the right-hand rule (for example, if the tallying plane is normal to the *X* axis, then, according to the previously-defined direction of the *X'* axis, the direction of the *Y'* axis is the same as the direction of the *Z* axis of the "main" coordinate system). Having defined the coordinate system on the tallying plane, the radial coordinate "R" can be defined as the distance from the point of crossing to the origin of that coordinate system, and the azimuth angle "PHI" can be defined as the angle between the *X'* axis and the line joining the origin with the point of crossing. The positive direction of measuring the azimuth angle "PHI" is counterclockwise when observed from the half-space where the normal vector is pointing to.

The particle counts of the plane-crossing tallies are updated only after the source particle has entered a cell (or the target layer in simple geometry mode). If the source particle crosses any of the tallying planes before entering a cell, that event will not cause an update of the particle counts of the corresponding plane-crossing tallies.

Each tally data are written to a separate text file. Those files are in a subfolder created in the same folder as the main output file (the subfolder name format is "Files_<output_file_name>"). For single-plane tallies, the file name consists of the tally and range identifiers separated by the underscore, e.g., "ETALLY1_E1.txt" in the case of a 1D tally, or "ETALLY1_X1_Y1.txt" in the case of a 2D tally. For multi-plane tallies, the file name also includes the sequence number of the plane in the set of parallel planes and the distance between that plane and the first plane of the set (in cm), which is written in parentheses immediately after the plane number, e.g., "ETALLY1_30(2.9)_E1.txt". In each file, the first line contains column headers. All other lines contain upper bounds of the bins of the primary variable in the first column and the corresponding particle counts, energy transfer (optional) and relative standard deviations in the other columns. If the input file contains the option "TALLIES_PER_SOURCE 1", then the particle counts and the values of energy transfer will be per source particle (i.e., they will be divided by the total number of source particles). For 1D tallies, the header of the second column is "N", the header of the column with energy transfer is "E_flux", and the header of the last column is "RelSDev". For 2D tallies, headers of all columns with particle counts, energy transfer and relative standard deviations have the format "N_<value>", "F_<value>" and "D_<value>", respectively, where "<value>" is the upper bound of the corresponding bin of the secondary variable. The column headers and all numerical values in the tally data files are tab-delimited.

The values of the energy transfer will be absent in the plane-crossing tally data files if the directive "TALLIES_E_FLUX 0" is included in the input file. The setting controlled by the global switch TALLIES_E_FLUX can be overridden for individual plane-crossing tallies by including the directive "E_FLUX 1" or "E_FLUX 0" in the tally definition.

If no interaction forcing is used (see Section 4.8), then the (absolute) standard deviation of the particle count in each bin is equal to the square root of the number of particles that have crossed the given plane in any direction, subject to the constraints of the given bin. This is equivalent to assuming that all particles counted in a given bin are independent and have weights equal to 1. The relative standard deviation is obtained by dividing the mentioned absolute standard deviation by the absolute value of the particle count corresponding to the same bin (note that the latter particle count may have been calculated with different signs assigned to particles crossing the plane in different directions). If interaction forcing is used, then particles that are counted in each bin have in general different weights. In this case, the absolute standard error is calculated as the square root of the sum of squared weights of all particles that have been counted in the given bin, regardless of their direction of crossing.

### *4.7.2. Cell-entry tallies*

A cell-entry tally is an energy tally of electrons or photons that have crossed a user-defined subset of faces of a particular cell. An cell-entry tally for electrons and positrons is defined using the keyword ECELL_ENTRY with an appended positive integer number (the tally identifier). A

photon cell-entry tally is defined using the keyword PCELL_ENTRY (its usage is identical to usage of the keyword ECELL_ENTRY). A "mixed" (electron-photon) cell-entry tally is defined using the keyword CELL_ENTRY (its usage is also identical to usage of the keyword ECELL_ENTRY). The format of the definition of a cell-entry tally is the following:

ECELL_ENTRY<n> cellID tallyType E<i1> E<i2> E<i3> … ±surfID1 ±surfID2 ±surfID3 …

This definition has the following components:

1) "cellID" is the cell identifier. In the case of a union of cells, only the first cell of the union may be specified (the remaining cells of the union will be taken into account automatically).

2) "tallyType" is the tally type, which can be –1, 0, 1 or 2. Those numbers mean the following:
   (a) if the tally type is –1, then only the particles escaping the specified cell will be tallied;
   (b) if the tally type is 0, then both the entering particles and the escaping particles will be tallied, and the direction of crossing will determine the sign of the count increment: an entry into the cell will cause an increase of the particle count by 1, and an escape from the cell will cause a decrease of the particle count by 1;
   (c) if the tally type is 1, then only the particles entering the specified cell will be tallied;
   (d) if the tally type is 2, then both the entering particles and the escaping particles will be tallied, and the direction of crossing will have no effect on the sign of the count increment.

3) "E<i1>", "E<i2>", etc. are identifiers of energy ranges, which define the number of energy tallies and their bins. Each of those energy ranges must be defined before the cell-entry tallies that use them (the rules of defining an energy range are explained in Section 4.7.1).

4) "±surfID1", "±surfID2", etc. are the identifiers of the bounding surfaces used in the definition of the specified cell. If the identifiers are entered with the minus sign, then those surfaces will be excluded, i.e., the set of faces used for the tally will include all faces of the cell excluding the specified surfaces. If the surface identifiers are entered without the minus sign, then only they will be included. Positive surface identifiers cannot be mixed with negative ones (otherwise the program will quit with an error message). If there are no surface identifiers listed in the definition of a cell-entry tally, then all faces of the cell will be included in the tally.

   If the definition of the specified cell includes the cell complement operator "#", then the zero surface identifier is allowed in the definition of the cell-entry tally. It denotes the interface between the bulk of the specified cell and all cells specified after the complement operator, i.e., all cells contained inside the specified cell. If the zero surface identifier is specified, then the cell-entry tally will be incremented (or decremented, as explained above) whenever the particle enters one of the contained cells from the specified "host cell", or when a particle enters the host cell from one of the contained cells. If there is the minus sign before the zero ("–0"), then the mentioned interface will be excluded from the cell-entry tally (it is included by default).

Facets of macrobodies may be specified in the definition of a cell-entry tally in the same way as in the definition of a cell, i.e., by specifying the macrobody identifier and the facet number separated by the decimal point. In addition, brackets may be used as a shorthand notation for a sequence of facets of a single macrobody (see the end of Section 4.3 for details).

Individual bounding surfaces of a cell obtained by transforming another cell may be specified in the definition of a cell-entry tally by the same identifiers as in the original cell. In the case of a union of two or more cells that are related to each other by a coordinate transformation, specification of a single number would be interpreted as a set of surfaces obtained by transforming the indicated original surface in all those cells. In order to use only the surfaces that are specified in the definition of a particular cell of the union, the following format should be used for specifying the surface: <cellID>/<surfaceID>. For example, "2/5" means surface No. 5 specified in the definition of cell No. 2 (if that cell was defined explicitly, by listing all bounding surfaces), or the surface obtained by transforming the surface No. 5 (if cell No. 2 was obtained

by transforming another cell whose definition includes surface No. 5). As in the case of a sequence of facets of a single macrobody, brackets may be used to avoid repetition of the cell identifier. For example, "2/[5 6 8.2]" means surfaces 5, 6 and facet No. 2 of the macrobody No. 8 used in the definition of cell No. 2 (or obtained by transforming the indicated surfaces used in the definition of the original cell). If some of the surfaces inside the brackets are facets of a single macrobody, then the "second-level" brackets may be used, for example, "2/[1 2 5.[1 2 4] 3]". The asterisk after the cell identifier (or after the slash following the cell identifier) means all bounding surfaces of the indicated cell (for example, "2*" or "2/*").

If a bounding surface of a union of cells is shared by two or more constituent cells of that union, then it is not possible to define a cell-entry tally so that only the crossings of that bounding surface into or out of a particular constituent cell are counted: if that bounding surface is included in the cell-entry tally for that union of cells, then any crossing of that surface by a particle entering or escaping that union will be counted, and if that bounding surface is excluded, then any crossing of that surface by a particle entering or escaping that union will not be counted.

The identifiers of energy ranges and bounding surfaces may be listed in any order.

If the source point is inside a cell, entry of the source particle into that cell at the source point will not cause an update of cell-entry tallies (only surface crossings are taken into account).

There is a separate ASCII output file for each cell-entry tally and each energy range used in its definition. The names of those files include the identifiers of the cell-entry tally and the energy range. The format of those files is the same as the format of the files with the data of one-dimensional plane-crossing tallies (see Section 4.7.1), and they are placed into the same subfolder. The relative standard deviations of the particle counts are calculated in the same way as for plane-crossing tallies (see Section 4.7.1).

The sign of the energy transfer corresponding to particles that escape the cell is determined by the type of the cell-entry tally in the same way as the sign of the corresponding particle count (see above). As in the case of plane-crossing tallies, the values of the energy transfer will be absent in the cell-entry tally data files if the directive "TALLIES_E_FLUX 0" is included in the input file. The setting controlled by the global switch TALLIES_E_FLUX can be overridden for individual cell-entry tallies by including the directive "E_FLUX 1" or "E_FLUX 0" in the tally definition.

In the sample simulation corresponding to the input file given in Section 4.2, plane-crossing tally "PTALLY" and cell-entry tally for cell No. 3 "PCELL_ENTRY3" would be identical, because the directive "LIM X –2.12132 2.12132" in the definition of the plane-crossing tally ensures that only the particles entering cell No. 3 through surface No. 8 are counted, and because the particles escaping cell No. 3 through surfaces No. 6 and No. 7 are the same particles that entered cell No. 3 through surface No. 8. In addition, particles cannot enter cell 3 through surfaces 6 and 7 (they can only escape through those surfaces). Consequently, if surface identifiers "6 7" or "–8" are specified in a cell-entry tally for cell No. 3, then the tally type 2 is equivalent to the tally type –1, and the tally type 0 would differ from the tally types 2 or –1 only by the sign of the particle count.

### *4.7.3. Pulse-height tallies*

A pulse-height tally is the tally of absorbed energy. It can be calculated either for the entire simulated system or for a particular cell. It is defined using the keyword PULSE_HEIGHT. In order to calculate a pulse-height tally for an individual cell, that keyword should be followed by "C<n>", where <n> is the cell identifier. The remaining entries in the definition of a pulse-height tally are three space-delimited numbers: the lower bound of the energy range, the upper bound of the energy range, and the number of bins. Then MCNelectron will divide the specified energy interval into the specified number of equidistant bins and count the number of histories when the energy absorbed in the target material belonged to each of those bins. For example, the following directive defines a pulse-height tally with 100 bins for cell 4:

```
PULSE_HEIGHT  C4  0  1  100
```

In this example, the tallied energy range is from 0 to 1 MeV (assuming the default unit of energy, i.e., "E_UNIT MeV"), and the bin width is 10 keV. Each bin count is incremented when the absorbed energy is less than or equal to the high-energy limit of the bin and greater than the low-energy limit of the bin. The histories when the absorbed energy was less than or equal to the low-energy bound of the first bin, or greater than the high-energy limit of the last bin, are not counted.

It is also possible to specify the identifier of an energy range instead of the three mentioned numbers in the definition of a pulse-height tally. For example, if the energy range is defined as follows:

```
E1   0 1 100
```

then the following directive would define the same pulse-height tally as the previous example:

```
PULSE_HEIGHT  C4  E1
```

Only one energy range may be specified in the definition of a pulse-height tally (unlike in the definitions of plane-crossing and cell-entry tallies).

The pulse-height tally data are placed into the text file "PulseHeight.txt", which is in a subfolder created in the same folder as the main output file (the subfolder name format is "Files_<output_file_name>"). In the case of pulse-height tallies for individual cells, the cell identifier is appended to the file name (for example, "PulseHeight_C4.txt"). If there are two or more pulse-height tallies for the entire system or for a particular cell, then the sequence number of the tally will be appended to the file name. The pulse-height tally data file contains three to five columns of numbers: the high-energy limits of all bins, the corresponding numbers of counts, values of the total absorbed energy and the average absorbed dose (in Gy) for each bin (those two values are optional), and the relative standard deviation of each number of counts (the relative standard deviation is equal to the square root of the difference between the inverse number of counts and the inverse number of histories). If the input file contains the option "TALLIES_PER_SOURCE 1", then the particle counts and the values of the total absorbed energy and the average absorbed dose for each bin will be per source particle (i.e., they will be divided by the number of source particles). The first line of the file contains the column headers. The numbers and column headers are tab-delimited. If the total mass of all cells (or the mass of a given cell) is zero, infinite or unknown, then the absorbed dose will be absent (i.e., there will be four columns instead of five).

Both the absorbed energy and the absorbed dose will be absent in the pulse-height tally data files if the directive "TALLIES_E_DEPOSITION 0" is included in the input file. The setting controlled by the global switch TALLIES_E_DEPOSITION can be overridden for individual pulse-height tallies by including the directive "E_DEPOSITION 1" or "E_DEPOSITION 0" at the end of the tally definition.

### *4.7.4. Specifying the maximum error for tallies*

For each tally, it is possible to specify the maximum relative standard deviation (or "target relative error"). If it is determined during the simulation that the relative standard deviation for each of those tallies is less than the corresponding target error, the simulation will be terminated immediately. Thus, the target errors are used as alternative criteria for determining if a simulation may be stopped. The tallies with specified target errors will be therefore called "control tallies". The simulation is stopped when the target error is reached for each of the control tallies, or when all source particles are processed (the number of source particles is specified in the input file or on the command line after the keyword "N"). The target error is specified using one of these keywords:

```
AVGERR
MINERR
MAXERR
```

This keyword must be followed by a value of the target error. In order to determine if the target is met, the program compares the average, minimum or maximum relative standard deviation, calculated over all bins of all tallies defined on the same line, with the corresponding target error once every second. In the case of a plane-crossing tally, that keyword may be inserted anywhere

after the geometric parameters (i.e., after the definition of the plane and the two optional numbers defining a set of equidistant planes), for example:

ETALLY1 0 0 0 0 0 1 100 0.1 E1 AvgErr 0.05

In this example, the simulation will be stopped when the standard deviation averaged over all bins in all 100 tallies of the set becomes less than 5 % (assuming that there are no other control tallies defined). In the case of a pulse-height tally, one of the mentioned three keywords must be added at the end of the tally specification, for example:

PULSE_HEIGHT 0 1 100 MaxErr 0.02

In this example, the simulation will be stopped when the maximum standard deviation over all 100 bins of this pulse-height tally becomes less than 2 %. In the case of a cell-entry tally, one of the mentioned three keywords may be inserted anywhere after the cell identifier and the tally type.

During the simulation, the program displays the current value of the relative error for the control tally and the corresponding target error. If there are two or more control tallies, then the displayed error is the one that is furthest from the target error (in relative terms), i.e., that has the largest ratio to the target error. For example, in the case of the two tallies defined above, the message that is appended to the line of text with second-by-second statistics shown in the console window might look like this:

```
AvgErr (ET1) = 9.4% / 5%
```

or:

```
MaxErr (PH1) = 2.82% / 2%
```

In those messages, the characters in parentheses indicate the type of the tally and its sequence number among all tallies of the same type (in the above examples, those are the plane-crossing tally No. 1 for electrons, i.e., "ETALLY1", and the pulse-height tally No. 1).

### *4.7.5. Specifying the particle type for tallies*

In a plane-crossing tally or a cell-entry tally, it is possible to take into account only the particles of specific origin (for example, only the source particles, or only the characteristic X-ray photons). This allows eliminating the "background" from the particles that are not important for the problem that is being solved (for example, eliminating the contribution of bremsstrahlung photons, if the goal is estimation of characteristic X-ray generation efficiency). The particle origin is specified by a set of comma-delimited numbers from 0 to 11 after the keyword "ORG" in the definition of the tally, where the zero corresponds to the source particles, and the meanings of the other numbers are the same as of the numbers used to categorize the secondary particles in the track data files (see Section 4.6). There must be no spaces between the commas and the numbers (otherwise, in the case of a cell-entry tally, the type numbers could be misinterpreted as the surface identifiers, resulting in incorrect tally data without any indication of the error). For example, the following directive defines a plane-crossing tally that takes into account only the characteristic X-ray photons:

PTALLY1 0 0 0 0 0 1 E2 ORG 3,4

If the first particle type listed after "ORG" is preceded by the minus sign, then all listed types of particles will be excluded, and all other types of particles will be included. For example, the following directive is equivalent to the previous one:

PTALLY1 0 0 0 0 0 1 E2 ORG -0,1,2

By default, if the keyword "ORG" is not used, then all 12 particle types are taken into account, excluding the photon types for "ETALLY<n>" and "ECELL_ENTRY<n>", and electron types for "PTALLY<n>" and "PCELL_ENTRY<n>". The photon types are 1 to 4 (and 0 if the source particles are photons). The electron types are 5 to 11 (and 0 if the source particles are electrons or positrons). If any of the photon types is listed after the keyword "ORG" in a definition of "ETALLY<n>" or "ECELL_ENTRY<n>", or an electron type is listed after the keyword "ORG" in a definition of "PTALLY<n>" or "PCELL_ENTRY<n>", this type will still be excluded. Using both electron and photon types is possible only for "TALLY<n>" and "CELL_ENTRY<n>".

*4.7.6. Specifying the time period of saving the tally data to files during the simulation*

MCNelectron can write the tally data to files not only after ending the simulation, but also during the simulation at regular intervals of time specified by the user. This setting can be controlled either globally (for all tallies at once), or for individual tallies by inserting the corresponding keywords into the definition of the tally. The two global keywords controlling output of tally data during the simulation (TALLIES_OUTPUT and TALLIES_OUTPUT_INTERVAL) were described in Section 4.1. The global setting controlled by TALLIES_OUTPUT and TALLIES_OUTPUT_INTERVAL may be overridden for individual tallies using the corresponding keywords OUTPUT and INTERVAL in the definition of the tally. The keyword OUTPUT must be followed by 0 or 1, and the keyword INTERVAL must be followed by the length of the time interval in seconds (the smallest allowed interval is 1 s). For example, the following directive specifies that the data of the plane-crossing tally defined in the example of Section 4.7.5 should be saved every 10 seconds:

```
PTALLY1   0 0 0   0 0 1   E2   ORG 3,4   OUTPUT 1   INTERVAL 10
```

In multithreading mode, thread No. 1 is used for writing the tally data (during that time, the simulation is paused in that thread), whereas all other threads continue the simulation. Consequently, some of the particle counts may be updated during the data output, causing the apparently missing or extra counts in some of the tallies. However, since the tally output usually takes only a fraction of a second, those discrepancies have negligible influence on the accuracy of the tallies, and this influence decreases with increasing simulation time. There can be no such discrepancies in the final tally data, which are written to files after ending the simulation.

### *4.8. Interaction forcing*

Interaction forcing is a variance reduction technique that is based on an artificial increase of frequencies of certain types of particle interactions. The bias that could be caused by this method is avoided by assigning weights to particles such that the average sum of the weights of all particles produced in a certain type of interactions is equal to the average number of particles that would be produced in the same interactions during an analog simulation (the "analog" simulation is the one that does not apply variance reduction techniques).

In principle, the following approach could be applied in order to take into account all possible interactions (even the ones with extremely small probability) without the need to simulate a large number of histories (one "history" is the sequence of all interaction events started by a single source particle):

1. Whenever a collision occurs, split the particle into as many particles ("replicas") as there are interaction types and assign the weight to each "replica" equal to the weight of the original particle ($W$) multiplied by the relative frequency of the corresponding interaction type. Since the sum of all relative frequencies is equal to 1, the sum of all those weights is equal to $W$.
2. Make all those "replicas" identical to the original particle in all other respects (i.e., the same energy, the same direction of motion and the same position).
3. Simulate each type of interaction (one type of interaction for each "replica") in the usual way, i.e., sample energies and directions of the primary and secondary particles from the relevant probability distributions.
4. Assign weights to the secondary particles equal to the weight of the corresponding "replica" of the original particle.

After that, each of the particles produced by each type of interaction (possibly including the primary particle with a changed energy or direction) is tracked by repeating the same steps. In order to obtain an unbiased value of the tallied quantity (e.g., the total energy loss, or the total number of particles produced in a certain type of interactions, or the total count in a particular bin of a particular tally), the contribution of each particle to that sum must be multiplied by the particle weight. However, this method has two obvious drawbacks:

a) Some of the interaction types may be already frequent enough, or those interactions may be not important for the simulated phenomenon (for example, elastic scattering of electrons is not important when simulating electron-induced characteristic X-ray emission). In those cases, no significant improvement in precision would be achieved by forcing those interactions.
b) Since each interaction is accompanied by particle "splitting", such an approach would cause an exponential growth of the total number of particles that have to be tracked, resulting in prohibitive memory requirements. In addition, the simulation time per history may become so long that it would offset any gains caused by a decrease of the number of histories that have to be simulated.

The first of the mentioned drawbacks can be eliminated by forcing only the types of interactions that are both relatively infrequent and important for the investigated phenomenon. For example, the type of interaction that is most important for simulations of electron-induced characteristic X-ray emission is inner-shell impact ionization, which is much less frequent than outer-shell impact ionization. All other types of interactions are sampled in the usual way, i.e., by selecting one of them randomly. This decreases the number of particles that have to be tracked, because all non-forced interactions are represented by just one "replica" per collision. However, the second drawback is not eliminated, because there are still at least two "replicas" per collision, and their energies are equal to energy of the original particle. The latter equality means that each of the "replicas", if tracked in an analog manner, would create, on the average, the same number of secondary particles as the original particle (although with a smaller weight). Thus, a single collision may double the total number of particles that have to be tracked. If this "doubling" is applied to all subsequent collisions, too, then the total number of tracked particles will grow in a geometric progression, slowing down the simulation and eventually causing memory overflow.

One of the ways to deal with this multiplication of tracked particles is the so-called "Russian roulette" technique: if the particle weight $W$ is less than a pre-defined minimum weight $W_{min}$, the particle is "killed" (i.e., removed) with probability $1 - W / W_{min}$, and if the particle "survives" this game, then it is assigned the weight $W_{min}$. This ensures that the total weight of the surviving particles is statistically the same as the total weight of the particles just before "rouletting" them (because the survival probability is equal to $W / W_{min}$). Since the surviving particle is in all other respects identical to the initial particle, all statistics calculated during the Monte Carlo simulation remain unbiased.

The implementation of interaction forcing in MCNelectron is based on applying the "Russian roulette" (see the previous paragraph) to the entire set of the "replicas" participating in the forced interactions in a given collision. That is to say, all those replicas either are "killed" or "survive" with probability $W_f / W_{min}$, where $W_f$ is the total weight of all replicas participating in the forced interactions. The weights of the surviving replicas are increased by a factor of $W_{min} / W_f$, so that their sum is exactly equal to $W_{min}$. If $W_f > W_{min}$, then the set of forced interactions is reduced by changing the status of the interactions from "forced" to "non-forced" one by one, starting from the interaction with the highest relative frequency, until the total weight of all replicas participating in the remaining forced interactions becomes less than $W_{min}$ (its default value is 0.001). In this case, a corresponding warning is displayed in the console window. This removal of forced interactions is temporary; all those steps are repeated before each collision (however, the mentioned warning is displayed only once for each type of forced interactions in each CPU thread and in each used CUDA device).

The above-described variant of interaction forcing allows in some situations to reduce the simulation time needed to achieve the required precision in the tallies. For example, interaction forcing causes a significant decrease of the simulation time (sometimes more than by two orders of magnitude) when simulating electron-induced characteristic X-ray emission. In MCNelectron v1.2.6, interaction forcing can be applied only to electron and positron interactions. There are six types of electron and positron interactions that can be forced:

1) elastic scattering,
2) bremsstrahlung,
3) atomic excitation,
4) K-shell impact ionization,
5) L-shell impact ionization,
6) M-shell impact ionization.

L-shell impact ionization is actually a set of three interactions (one for each of the three subshells of the L shell), and M-shell impact ionization is a set of five interactions (one for each of the five subshells). Consequently, if M-shell impact ionization is the only type of forced interaction, then the primary electron will be split into six replicas (five replicas corresponding to the five forced interactions and one replica for the single randomly selected non-forced interaction).

Below are descriptions of all keywords that control interaction forcing.

FORCE_WT is the total weight of the replicas that "survived" the game of Russian roulette ($W_{min}$). An increase of this parameter causes an increase of the number of histories that have to be simulated in order to achieve the required precision of the statistics corresponding to the forced interactions, and consequently improves sampling of all interactions (including the non-forced ones). Its default value is 0.001, and the maximum allowed value is 0.1.

The remaining keywords related to interaction forcing, which are listed below, are used as "switches", which "turn on" or "turn off" forcing of certain types of interactions (a non-zero value of the integer number that follows each of those keywords means that the corresponding interaction type will be forced, and the zero value means that it will not be forced):

FORCE_ELASTIC – forcing of elastic scattering from all types of atoms,
FORCE_BREMS – forcing of bremsstrahlung from all types of atoms,
FORCE_EXC – forcing of excitation of all types of atoms,
FORCE_ION_K – forcing of K-shell impact ionization of all types of atoms,
FORCE_ION_L – forcing of L-shell impact ionization of all types of atoms,
FORCE_ION_M – forcing of M-shell impact ionization of all types of atoms.

***Note***: The specified interaction will be forced only if its cross section is non-zero.

In addition, it is possible to specify interaction forcing only for a particular chemical element. This is done by appending the atomic number to any of the previous six keywords. For example, the keyword FORCE_ION_M92 should be used to turn on or off forcing of M-shell impact ionization for uranium. If both the keyword with the atomic number and the keyword without it are present, priority is given to the keyword with the atomic number. For example, if the input file contains the directives "FORCE_ION_L 1" and "FORCE_ION_L8 0", then the program will force L-shell impact ionization of all atoms excluding oxygen.

By default, interaction forcing is not applied, i.e., the mentioned switches are "off".

If a set of forced interactions has been defined, it is possible to apply electron interaction forcing only in particular cells of the geometry, or only in particular materials. This is controlled by the directive "FORCE 1" or "FORCE 0" in the definition of the material or the cell (or the union of cells). The global switch "FORCE_MATERIALS" is used to tell the program if interaction forcing should be applied by default. In any case, interaction forcing will be applied only after defining the set of forced electron interaction types. Be default, the switch FORCE_MATERIALS is "on", i.e., interaction forcing will be applied in all materials where the forced interaction types are possible, unless overridden by the directive "FORCE 0" in the definition of the material or the cell. If the directive "FORCE_MATERIALS 0" is used, then interaction forcing will be applied only after enabling it explicitly with the directive "FORCE 1" in the definition of the material or the cell. If the numbers after "FORCE" in the definitions of a material and a cell using that material are different (i.e., if one definition contains "FORCE 0", whereas the other one contains "FORCE 1"), then priority is given to the setting specified in the definition of the cell.

### *4.9. Format of the file with alternative cross sections information*

An example of the file with alternative cross sections information:

```
EXC_FN 18 c:\Plasma Data Exchange Project\e-Ar excitation - SIGLO.txt
EXC_INTERP_END 18 5e5
EXC_INTERP_RULE 18 5
EXC_INTERP_COEFS 18 12.3276 2.08644 -0.273936 0.00864765
ION_FN 18 c:\Plasma Data Exchange Project\e-Ar ionization - SIGLO.txt
ION_INTERP_END 18 1e5
ION_INTERP_RULE 18 5
ION_INTERP_COEFS 18 26.0849 -0.962273 -0.035216 0.00279351
EXC_FN 54 c:\Plasma Data Exchange Project\e-Xe excitation - SIGLO.txt
EXC_INTERP_END 54 5e5
EXC_INTERP_RULE 54 5
EXC_INTERP_COEFS 54 31.0399 -3.8101 0.32736 -0.0103518
ION_FN 54 c:\Plasma Data Exchange Project\e-Xe ionization - SIGLO.txt
ION_INTERP_END 54 1e5
ION_INTERP_RULE 54 5
ION_INTERP_COEFS 54 15.3521 2.44828 -0.370784 0.0135908
70
80
90
100
110
120
130
```

Each line of the file with alternative cross sections information, excluding the lines with additional energies, starts with a keyword, which is followed by the atomic number ($Z$) of a chemical element. Explanation of the keywords used in that file:

EXC_FN is used to specify the file name with alternative low-energy excitation cross sections. In the above example, that file name is specified for Ar and Xe ($Z$ = 18 and 54).

EXC_INTERP_END is used to specify the high-energy endpoint of the interpolation range (eV) for excitation cross sections. In the above example, that endpoint is 500 keV. ***Note***: The low-energy endpoint of the interpolation range coincides with the largest value of the electron energy in the file with alternative cross sections. If the specified high-energy interpolation endpoint is less than the just-mentioned energy, then there will be no interpolation.

EXC_INTERP_RULE is used to specify interpolation rule for excitation cross sections. Four interpolation rules are supported: linear-linear, linear-log, log-linear and log-log. They are denoted by integer numbers from 2 to 5 (by analogy with ENDF format):

2 – linear-linear, 3 – linear-log, 4 – log-linear, 5 – log-log.

EXC_INTERP_COEFS is used to specify four coefficients of the third-degree polynomial that is used for interpolation of excitation cross sections. Those coefficients must be listed in the order of increasing degree: the first coefficient corresponds to the term of degree zero, and the last coefficient corresponds to the term of degree 3. In the above example, the natural logarithm of the excitation cross section for argon in the interpolation range is calculated as follows:

$$\ln(\sigma) = 12.3276 + 2.08644 \cdot \ln(E) - 0.273936 \cdot [\ln(E)]^2 + 0.00864765 \cdot [\ln(E)]^3,$$

where the energy $E$ is expressed in eV, and the cross section $\sigma$ is expressed in barns.

Keywords ION_FN, ION_INTERP_END, ION_INTERP_RULE, ION_INTERP_COEFS are used to specify the files with alternative ionization cross sections and the corresponding interpolation information. Their usage is the same as for excitation cross sections (see above).

If no interpolation is needed, the lines starting with "EXC_ INTERP_" or "ION_INTERP_" may be omitted. Any non-empty line that starts not with one of the mentioned keywords is interpreted as a part of the list of additional energies for calculation of ionization cross sections. Each such line must contain only one energy value (eV), and all those energies must be less than the largest energy in all files with alternative ionization cross sections. If there are no additional energies, then the values of alternative ionization cross sections will be calculated for the same energies that are given in the ENDF tables of ionization cross sections for each electronic subshell, using linear interpolation. The set of additional energies is the same for all chemical elements that compose the target material. ***Note***: The additional energies should be used when the maximum of the ionization cross section is not represented well enough in ENDF tables of ionization cross sections (because of too large intervals between energy values in the ENDF electroionizaton data).

### *4.10. CUDA-specific keywords*

There are 23 keywords that control usage of computer resources by the GPUs and the division of workload between the CPU and the GPUs. Before describing those keywords, several terms must be defined. The term "task" will be used to mean the part of the code that simulates a particle history. During the simulation, each task generates source particles and follows each source particle together with all secondary particles until all of them either escape from the system or are lost during various interactions. Then that task generates the next source particle, etc. On a lower level (i.e., the level of hardware and the operating system), simulation is run by independent "threads", whose number is limited by the available computing resources. In the case of Nvidia GPUs, the maximum possible number of threads per one CUDA device is 1024. As explained in Section 1.3, the CUDA version of MCNelectron does not assign a fixed set of tasks to each thread. Instead, each task is done in steps by various threads, i.e., the tasks are frequently "redistributed" among the threads. For maximum performance, the number of tasks should be much larger than the number of active threads, because then there is a greater freedom for choosing the optimum set of tasks to run at each given moment of time (the mentioned "optimum" set of tasks is the one that minimizes warp divergence, as explained in Section 1.3). In contrast, each thread on the CPU follows a particle history in its entirety, i.e., each CPU task is executed by a single thread. Hence, in the case of a CPU-only simulation, the terms "task" and "thread" may be used interchangeably. The maximum possible total number of tasks (including the tasks on all used GPUs and on the CPU) is $2^{16} = 65536$.

For each history, a task uses a stream of random numbers generated from a fixed "seed". The total number of streams of random numbers is defined using the mentioned keyword "NSTREAMS" (see Section 4.1). This number includes both the streams that are used on the CPU and the streams that are used by the CUDA devices. Ideally, all streams should not be correlated with each other and should not repeat themselves. Each task is assigned a fixed set of streams and then cycles between them for different histories (thus, the streams of random numbers are assigned to tasks, not to threads). The maximum possible total number of streams is $2^{16} = 65536$.

The part of an application that runs on a CUDA device is called a "kernel". On each CUDA device, one kernel is executed at a time, but many threads execute the kernel simultaneously. On the application level, those threads are grouped into "blocks", which are not synonymous with "warps" mentioned in Section 1.3. A block may span several warps, or it may be only a part of a single warp. In absence of warp divergence, the optimum size of a block is a multiple of the warp size, i.e., 32. If warp divergence is significant, better performance may be sometimes achieved using block size less than 32 (however, the block size should still be a multiple of 8). The results of testing indicate that the best performance on desktop computers with GeForce GTX 780 Ti and GTX 580 cards is usually achieved when the block size is 8, and in the case of a notebook computer with a GeForce GTX 960M card the best performance has been achieved using block size 32.

All CUDA-specific keywords end with "_CUDA". As all other keywords, they must be followed by the value of the corresponding parameter. 19 of the 23 CUDA-specific keywords may be specified either on the command line or in the input file. The remaining 4 keywords may be

specified only on the command line. If a keyword is specified both on the command line and in the input file, then the parameter value specified on the command line will be used. Below is the list of the mentioned 19 keywords:

**rand_CUDA** is used to specify the type of the random number generator to be used on the CUDA devices. The number 1 indicates the Fibonacci series random number generator that is used on the CPU (in this case, the parameter SEED in the input file will also be used to initialize the instances of that generator that run on the CUDA devices). The number 2 indicates the XORWOW generator from the cuRAND library. Simulations using the latter generator are usually faster by up to 10 %. This random number generator can be used only with the 64-bit version of MCNelectron_CUDA;

**seed_CUDA** is used to specify the seed of the XORWOW generator from the cuRAND library (this parameter is used only with the option "rand_CUDA 2", otherwise it is ignored). This seed must be positive and less than $2^{64}$;

**tasks_CUDA** is used to specify the number of tasks per one CUDA device. If that number is zero, then CUDA devices will not be used and all CUDA-specific keywords will be ignored;

**threads_CUDA** is used to specify the number of threads per one CUDA device (it has been noticed that the best performance is achieved when this parameter is slightly less than the maximum possible number 1024, e.g., 960);

**streams_CUDA** is used to specify the number of random-number streams per one task on a CUDA device. The total number of streams per one device is obtained by multiplying streams_CUDA and tasks_CUDA. The program calculates the number of streams on the CPU by subtracting the numbers of streams on all used CUDA devices from the total number of streams (defined by the keyword NSTREAMS). If the resulting number is less than the number of CPU threads (defined by the keyword TASKS), then a corresponding error message is displayed;

**blockSize_CUDA** is the number of CUDA threads per one block;

**time_CUDA** is used to specify the duration of a single invocation of the CUDA kernel (in milliseconds). The recommended value is 100 or less. If this number is too large, the computer may become unresponsive. If this happens, then after a few seconds the Nvidia video driver will usually recover (terminating MCNelectron_CUDA in the process), however, occasionally the recovery fails and the computer needs to be restarted. In addition, if the video card that is connected with the computer monitor is among the CUDA devices used for the simulation, then a too large value of that parameter may cause "stuttering" of the video signal, which may interfere with other work that is being done on the same computer at the same time. On the other hand, if this parameter is too small (e.g., less than 1), then the number of kernel calls may be become so large that the "overhead" caused by multiple kernel calls may become important and performance may drop;

**ratio_CUDA** is used to specify the fraction of particle histories that should be simulated by CUDA devices. If more than one CUDA device is used, then those histories are distributed equally among all used CUDA devices. During the simulation, the CPU and GPU workloads may be dynamically adjusted (see descriptions of the keywords "heap_CUDA" and "balance_CUDA" below);

**heap_CUDA** is used to specify how CUDA tasks start a new history. If that parameter is zero, then each task is assigned a fixed number of source particles (i.e., a fixed number of histories). This option precludes workload balancing between CUDA devices and the CPU; it should be used if exact repeatability of simulation results is needed. If that parameter is 1, then, after finishing a history, each task starts a new history if the total number of started histories on the CUDA device is less than the assigned total number of histories that have to be simulated; otherwise, the task becomes inactive. Thus, if histories that were initially assigned to a particular task were short, then that task may have enough time left for simulating extra histories and thus decreasing the workload of the tasks that are "lagging". Since the processing times of different histories are affected by various random factors, and since different tasks use different streams of random numbers, this

option introduces a random component into the sequence of random numbers generated in each task, i.e., it is no longer exactly repeatable even when the same starting seed is used (however, this is usually not a problem, because the aim of Monte Carlo simulations is estimation of statistical averages). If that parameter is 2, then each task does not follow a history from start to finish; instead, a "global" buffer of banked particles (consisting of secondary particles in all unfinished histories) is available to all tasks on a given CUDA device. After a particle is lost (e.g., due to escape from the system or due to decrease of its energy below the cutoff value), the task first attempts to read the banked particle data from the mentioned buffer. If the read attempt is successful, then that particle is tracked (even if it belongs to a history that was started by another task). If the read attempt fails (i.e., if the mentioned buffer is empty) and if the total number of started histories on the CUDA device is less than the assigned total number of histories that have to be simulated, then the task generates a new source particle and starts a new history. The option "heap_CUDA 2" has an even stronger randomizing influence on the sequence of random numbers than the option "heap_CUDA 1" does. In the case "heap_CUDA 2", the number of source particles is allowed to be less than the number of tasks. ***Notes***: **1)** Pulse-height tallies can be calculated only with "heap_CUDA 0" or "heap_CUDA 1". **2)** Workload balancing between CUDA devices and the CPU is possible only with "heap_CUDA 1" or "heap_CUDA 2" (see the description of the keyword "balance_CUDA" below);

**balance_CUDA** is a "switch" that "turns on" workload balancing between CUDA devices and the CPU. When this feature is "turned on", then during the simulation the number of histories assigned to the CPU and to each CUDA device is dynamically adjusted on the basis of performance of the CPU and of each CUDA device, in an attempt to minimize differences between total processing times of the CPU and of each CUDA device. The workload balancing in conjunction with option "heap_CUDA 1" or "heap_CUDA 2" eliminates the need to determine the optimal value of the parameter "ratio_CUDA" beforehand (the latter parameter defines only the initial distribution of workload, which is automatically adjusted during the simulation). ***Notes***: **1)** Workload balancing cannot be used together with the option "heap_CUDA 0". **2)** When workload balancing is turned on, MCNelectron_CUDA may occasionally simulate up to four "extra" histories;

**avgBanked_CUDA** is used to specify the memory amount that must be allocated on a CUDA device for storing the banked particle data (the "banked particles buffer"). If this parameter is positive, then it is interpreted as the average number of banked particles per one task when the mentioned buffer is full. If this parameter is negative, then it is interpreted as the fraction of the global memory on each CUDA device that must be allocated for banked particles. A cautionary note regarding SLI, quoted from "CUDA C Programming Guide" [2]: "an allocation in one CUDA device on one GPU will consume memory on other GPUs that are part of the SLI configuration of the Direct3D or OpenGL device. Because of this, allocations may fail earlier than otherwise expected." Thus, if there are two Nvidia video cards in SLI configuration, then values of the allocated memory fraction close to 0.5 or greater (for example, "avgBanked_CUDA -0.6") will cause a CUDA memory error (except for video cards with more than 4 GB of video memory). If there are more than two cards in SLI configuration, then the memory error may occur at an even lower fraction of the memory specified. For this reason, if a large number of banked particles are expected, it is recommended to disable SLI before the simulation.

**maxBankedRatio_CUDA** is used to specify the size of the "index buffer", i.e., the memory amount that must be allocated per one task for storing positions of banked particles in the above-mentioned banked particles buffer. The "index buffer" is used only with "heap_CUDA 0" and "heap_CUDA 1", because in those cases each task tracks only those secondary particles that were created by the same task in the same history. The number that follows this keyword is interpreted as the ratio of the number of elements of "index buffer" to the average number of banked particles per one task (defined by the keyword "avgBanked_CUDA");

**bankModeThr_CUDA** controls the "bank mode" when option "heap_CUDA 2" is used. The term "bank mode" refers to the choice of an electron that has to be banked after an impact ionization

event: the banked electron may be either the lower-energy one (the so-called “secondary” or “knock-on” electron), or the higher-energy one (the “primary” electron). Normally, when option “heap_CUDA 2” is used, the knock-on electron is the one that is banked. However, this may cause an overflow of the banked particles buffer in the case of high energies of source particles and thick targets. Consequently, the bank mode is changed when the filling fraction of the banked particles buffer exceeds a certain value. That “threshold” value of the filling fraction is specified after this keyword. Its default value is 0.8 (i.e., while the banked particles buffer is less than 80 % full, the knock-on electron will be banked, and when the banked particles buffer is more than 80 % full, the primary electron will be banked). ***Note***: In the case “heap_CUDA 0” or “heap_CUDA 1”, this keyword is ignored (in those cases, the primary electron is always the one that is banked);

**first_CUDA** is used to specify the sequence number of the first CUDA device in the range of CUDA devices used for the simulation (CUDA devices are numbered sequentially starting from 1);

**last_CUDA** is used to specify the sequence number of the last CUDA device in the range of CUDA devices used for the simulation (CUDA devices are numbered sequentially starting from 1). If firstCUDA is 1 and last_CUDA is 0, then all available CUDA devices will be used;

**use_CUDA** is used to specify the sequence numbers of the CUDA devices that must be used for the simulation. When specified on the command line, those numbers must be comma-delimited. For example, “use_CUDA 1,3” means that CUDA devices No. 1 and No. 3 have to be used. ***Notes***: **1)** When specified in the input file, the numbers may be either comma-delimited or space-delimited. **2)** The keyword “use_CUDA” cannot be used together with keywords “first_CUDA”, “last_CUDA”, or “skip_CUDA”. **3)** If the keyword “use_CUDA” is missing, then the range of used CUDA devices will be determined according to the numbers entered after the above-mentioned keywords “first_CUDA” and “last_CUDA”.

**r_CUDA** is used to specify the workload ratios for each of the CUDA devices that must be used for the simulation. I.e., the program divides the total CUDA workload ratio (specified using the mentioned keyword “ratio_CUDA”) among the CUDA devices proportionally to the numbers specified after this keyword. When specified on the command line, those numbers must be comma-delimited. This keyword can be used only together with the mentioned keyword “use_CUDA”, and the number of entries after “r_CUDA” must be the same as the number of entries after “use_CUDA”. For example, in the case “use_CUDA 1,3 r_CUDA 2,1 ratio_CUDA 0.6” the workload of the CUDA device No. 1 will be twice larger than the workload of the CUDA device No. 3. Since in this example the total workload of the CUDA devices is 60 %, the CUDA device No. 1 will simulate 40 % of all histories, and the CUDA device No. 3 will simulate 20 % of all histories. ***Notes***: **1)** When specified in the input file, the numbers may be either comma-delimited or space-delimited. **2)** If any of the workload ratios specified after “r_CUDA” is zero, the corresponding CUDA device will not be used. **3)** If the keyword “r_CUDA” is missing, the workload will be divided equally among all used CUDA devices.

**skip_CUDA** is used to specify the sequence numbers of the CUDA devices that must be skipped, i.e., not used for the simulation. When specified on the command line, those numbers must be comma-delimited. For example, “skip_CUDA 2,4 first_CUDA 1 last_CUDA 6” is equivalent to “use_CUDA 1,3,5,6”. The keyword “skip_CUDA” cannot be used together with “use_CUDA”. ***Note***: **1)** When specified in the input file, the device numbers listed after “skip_CUDA” may be either comma-delimited or space-delimited. **2)** If the keyword “skip_CUDA” is missing, then there will be no skipped CUDA devices, i.e., the sequence numbers of the used CUDA devices will be entirely defined using the above-mentioned keywords “first_CUDA” and “last_CUDA”, or using the list of devices specified after the above-mentioned keyword “use_CUDA”.

**stats_CUDA** controls output of CUDA device statistics to files. The allowed values are:

0 – no CUDA device statistics will be written to files,

1 – the summary statistics will be included in the output file,

2 – second-by-second statistics will be written to files in the same subfolder where the tally data are (the subfolder name format is “Files_<output_file_name>”). The names of files with second-by-second statistics have format “d<n>.txt”, where “<n>” is the sequence number of a CUDA device.

3 – both the summary statistics and second-by-second statistics will be written.

[For more information about CUDA device statistics, see Section 4.11.]

The remaining 4 CUDA-specific keywords control “emulation” of CUDA devices by the CPU (those keywords may be used only on the command line). Here, the term “emulation” means that the particle histories are simulated by the CPU using exactly the same random number streams that would be used by the CUDA device. I.e., the only difference between a normal CPU-only computation and a computation done in emulation mode is that the random number streams are assigned to particle histories differently. In emulation mode, assignment of random number streams to particle histories is controlled by the mentioned keywords “NSTREAMS”, “tasks_CUDA”, “streams_CUDA”, “ratio_CUDA” and “r_CUDA”. This makes it possible to reproduce the results of a hybrid CPU/GPU simulation using only the CPU or using only a part of available CUDA devices. For exact reproduction of the results in emulation mode, the options “rand_CUDA 1” and “heap_CUDA 0” are required. Below are descriptions of the four emulation-related keywords:

**em_CUDA** is used to specify the sequence numbers of the CUDA devices that must be emulated. Those numbers must be comma-delimited. For example, “em_CUDA 1,3” means that CUDA devices No. 1 and No. 3 have to be emulated. ***Note***: The keyword “em_CUDA” cannot be used together with keywords “emFirst_CUDA”, “emLast_CUDA”, or “emulate_CUDA”, which are described below;

**emFirst_CUDA** is used to specify the sequence number of the first emulated CUDA device in the range of emulated CUDA devices (CUDA devices are numbered sequentially starting from 1);

**emLast_CUDA** is used to specify the sequence number of the last emulated CUDA device in the range of emulated CUDA devices (CUDA devices are numbered sequentially starting from 1);

**emulate_CUDA** is a “switch” that “turns on” emulation of all devices specified after the keyword “use_CUDA” or belonging to the range of devices defined using the keywords “first_CUDA” and “last_CUDA”. This option cannot be used together with “em_CUDA”, “emFirst_CUDA” or “emLast_CUDA”.

The default values of the CUDA-specific parameters are the following:

| | |
|---|---|
| rand_CUDA | 2 |
| seed_CUDA | 123 |
| tasks_CUDA | 0 |
| threads_CUDA | 960 |
| streams_CUDA | 1 |
| blockSize_CUDA | 8 |
| time_CUDA | 100 |
| ratio_CUDA | 0.5 |
| heap_CUDA | 1 or 2 (depending on whether pulse-height tallies are used or not) |
| balance_CUDA | 1 |
| avgBanked_CUDA | -0.2 |
| maxBankedRatio_CUDA | 3 |
| bankModeThr_CUDA | 0.8 |
| first_CUDA | 1 |
| last_CUDA | 0 |
| stats_CUDA | 0 |
| emFirst_CUDA | 0 |
| emLast_CUDA | 0 |
| emulate_CUDA | 0 |

### *4.11. CUDA device statistics*

MCNelectron_CUDA outputs CUDA device statistics in three ways:

1) during the simulation, the second-by-second statistics as well as current values of the summary statistics are displayed on one line in the console window (that line is updated every second);
2) during the simulation, second-by-second statistics may be optionally written to a file (as explained in Section 4.10, this is achieved using the option "stats_CUDA 2" or "stats_CUDA 3");
3) after the simulation is finished, the summary statistics may be optionally written to the output file (as explained in Section 4.10, this is achieved using the option "stats_CUDA 1" or "stats_CUDA 3").

The formats of each of those three outputs are described below.

#### *4.11.1. One-line statistics in the console window*

Since all statistics cannot fit on a single line, MCNelectron_CUDA displays only a portion of the total set of statistics at a time. The displayed set of statistics is selected by repeatedly pressing the "+" or "-" key on the keyboard during the simulation (the key "+" cycles the statistics forward, and the key "-" cycles the statistics backwards). There are three lines of statistics for each CUDA device used and one line with some overall statistics. The latter line also contains the statistics pertaining to the part of the simulation that is done by the CPU. If two or more CUDA devices are used, then it is possible to cycle between statistics of the same kind for different devices (the CUDA devices are numbered sequentially starting from 1, and the CPU is the device No. 0). This is achieved by repeatedly pressing the "]" or "[" key on the keyboard during the simulation. After pressing the key "]", the same type of statistics for the next device will be displayed, and after pressing the key "[", the same type of statistics for the previous device will be displayed (if the next or previous device is the CPU, then after pressing "]" or "[" the line with overall statistics will be displayed).

During a hybrid CPU/GPU simulation, the line with overall statistics could look like this:

```
3.160 s: 8883/30000 (29.61%), CPU - 423/19500 (2.17%), CUDA - 8460/10500
(80.57%), threads 1920/1920, time 99.02%, 99.94%.
```

This line has the following components:

- The time elapsed since the start of the simulation (in seconds).
- The total number of finished histories, the total number of source particles and their ratio. ***Note***: In the case "heap_CUDA 2", the number of finished histories on CUDA devices is replaced by the number of finished source tracks (see the description of output parameter "src" below).
- The number of finished histories on the CPU, the total number of histories assigned to the CPU, and their ratio.
- The number of finished histories on CUDA devices, the total number of histories assigned to the CUDA devices, and their ratio (the same note as above applies).
- The number of active CUDA threads and the total number of CUDA threads.
- Ratio of the total kernel execution time to the total elapsed time (both those times are calculated using the system timer). This ratio indicates the amount of "overhead" caused by multiple invocations of the kernel (because of that overhead, this ratio is less than 1).
- Ratio of the total useful time spent by all CUDA threads to the total execution time of all CUDA threads (both those times are calculated using the CUDA native high-performance timer). The "useful" time is the time when a thread is active (i.e., the time interval between starting the thread and terminating it). This ratio indicates differences between workloads of different CUDA threads on a CUDA device, as well as between workloads of different CUDA devices (because of those differences, that ratio is less than 1).

If the simulation is performed using two or more CUDA devices and not using the CPU, then in addition to the above-mentioned overall value of the useful-to-total time ratio, its values for each of the devices are given between parentheses (this component is not present in the above example).

During a CPU-only simulation, this line contains some statistics for the first four simulation threads, for example:

```
2.001 s: 2780/40000 (6.95%), active threads - 1:2220:2220 (4), 2:2844:2844 (4),
3:2845:2845 (1), 4:3035:3035 (1), ...
```

In this case, the following three colon-delimited numbers are given for each of the first four active threads: the thread number, the sequence number of the currently used random-number stream (if this number is equal to the thread number, then it is not shown), and the sequence number of the current history (in the total number of source particles). After those three numbers, the current number of banked particles in that thread is given in parentheses.

If control tallies are used (see Section 4.7.4), then the largest current relative error for those tallies is included at the end of the same line (the format of this message is explained in Section 4.7.4). In this case, some of the previous information is omitted (in order to keep this line sufficiently short).

The three mentioned lines with CUDA device statistics could look, for example, like this:

```
2.049 s: dev. 1-1 - src 912 / 1960 = 46.53% (199) rem 70939 (48943) tb 100.16
(100.16) tcy 164.74 (163.50) tcl 92.68 (97.94)

4.218 s: dev. 1-2 - nk 42 (11) brk 0.000% (0.000%) cyc 617.922 (633.486) nTh
5.544 (5.221) maxLoad 10.285% (6.117%) nw 0.00% (0.00%)

5.219 s: dev. 1-3 - nB 105555, 0.310% (402, 0.001%) nThR 191.774 (924.000) nThW
608.208 (294.727) nThN 1120.019 (701.273)
```

Each of those three lines starts with the time elapsed since the start of the simulation (in seconds), followed by the device number and the sequence number of the set of statistics for that device. After that, there are several numbers (or pairs of numbers) which have the meaning of certain sums or averages, calculated by summing or averaging certain quantities over all kernel calls for the current device (there is one exception: "nB" has the meaning of a maximum value, rather than a sum). Each of those numbers (or a pair of numbers) is preceded by its notation and followed by a number (or two numbers) in parentheses. The numbers in parentheses indicate the change of the corresponding sum, or the value of the corresponding average, during the last second. The first set of CUDA statistics consists of the following components:

**src** – the number of finished histories, the total number of histories (i.e., source particles) assigned to that device, and their ratio. In the case "heap_CUDA 2", the number of finished histories is replaced by the number of finished source tracks. A "source track" is a connected series of line segments, each one representing a free flight of a particle. A source track starts at the point of entry of the source particle and ends at the point where the last particle of a chosen "path" in a given history is lost. If there is a "fork" in this "path" due to creation of a secondary particle, then the "branch" that is assigned to the source track is chosen according to arbitrary criteria (for example, if the source particle is a high-energy photon, then after a pair production event the source track will be continued by the electron created in that event, whereas the positron will be banked). Thus, a single source track may represent several particles. In a task that started a new history, the first attempt to read banked particle data after generating the source particle occurs immediately after the source track is finished. Since each source track starts with a source particle, the number of finished source tracks is less than or equal to the number of started histories and greater than or equal to the number of finished histories. The number of histories or source tracks finished during the last second is given in parentheses;

**rem** – the number of removed or replaced particles (or the sum of their weights if interaction forcing is used);

**tb** – average duration of a single block of CUDA threads during one kernel call (in milliseconds);

**tcy** – average duration of a single loop cycle in the kernel (in microseconds). A single loop cycle in one CUDA thread corresponds to a single call to a "state handler" mentioned in Section 1.3. In addition, a single loop cycle includes re-distribution of tasks among threads (see Section 1.3);

**tcl** – average duration of a single call to a state handler (in nanoseconds).

The second set of CUDA statistics consists of the following components:

**nk** – number of kernel calls;

**brk** – fraction of block calls ended by breaking the loop (as opposed to ending the loop because the assigned time is exceeded). This occurs when it is determined that all threads of the current block are inactive;

**cyc** – average number of loop cycles per one block call;

**nTh** – average number of active CUDA threads per one loop cycle;

**maxLoad** – fraction of loop cycles with maximum load (i.e., with all threads in the block active);

**nw** – fraction of loop cycles when a task, which was in the "read" state at the start of a cycle, remained in the "read" state at the end of that loop cycle (i.e., an attempt to read particle data from the banked particles buffer was unsuccessful and no more particle histories remained to be started). This can happen only with the option "heap_CUDA 2". Consequently, in the case "heap_CUDA 0" or "heap_CUDA 1" this fraction is always 0.

The third set of CUDA statistics consists of the following components:

**nB** – the maximum number of banked particles and its ratio to the maximum allowed number of banked particles. In the case "heap_CUDA 0" or "heap_CUDA 1", the maximum is per one task, and in the case "heap_CUDA 2", it is the maximum number of banked particles in the global buffer (the numbers in parentheses are calculated for the last kernel call).

**nThR** – average number of tasks that were in the "read" state at the start of a kernel call (a task is in the "read" state just before attempting to read banked particle data, or when it is inactive after finishing simulations of all histories that were assigned to it);

**nThW** – average number of tasks that were in the "write" state at the start of a kernel call (a task is in the "write" state just before simulating an interaction event characterized by creation of new particles, which may need to be banked);

**nThN** – average number of tasks that were in the "neutral" state at the start of a kernel call. A state is called "neutral" if it does not involve either reading from the banked particles buffer or writing to it (for example, calculation of interaction cross sections).

#### *4.11.2. Second-by-second CUDA device statistics written to files*

The first line in the file with second-by-second CUDA device statistics contains the column headers, and the other lines contain the values of various statistics for each second of the simulation. The first column of that file contains values of the time elapsed since the start of the simulation (in seconds), rounded to the nearest second. The next 16 columns contain 15 previously-described statistics, calculated for each one-second interval and listed in the same order as in one-line statistics (see above), and one additional statistic "tw" (column No. 13), whose meaning is similar to the mentioned quantity "nw", with the only difference that "tw" is the fraction of time rather than the fraction of loop cycles. I.e., "tw" is the fraction of total time spent for unsuccessful reads from the banked particles buffer by a task that was in the "read" state at the start of a loop cycle, with a condition that no more source particles had to be generated. The headers of those columns are the same as previously-mentioned notations. The numbers in columns No. 2 – 12 ("src" to "nw") and

No. 15 – 17 (“nThR”, “nThW” and “nThN”) are the same ones that would be displayed in parentheses in the one-line statistics during the simulation at the corresponding moment of time. In the case “heap_CUDA 0” or “heap_CUDA 1”, the numbers in column No. 14 (“nB”) have the meaning of the maximum number of banked particles per one history observed up to that moment in time, and in the case “heap_CUDA 2” they have the meaning of the maximum total number of banked particles observed during the last kernel call.

The next 18 columns (i.e., columns 18 to 35) contain fractions of the numbers of calls to various state handlers in the total number of calls. In other words, those are fractions of the total number of loop cycles spent for processing different stages of the simulation. The meanings of those 18 columns are explained below:

nPsig – calculation of photon interaction cross sections;
nPdist – calculation of the distance to photon collision;
nPinter – sampling of chemical element and interaction type in the case of a photon collision;
nCoh – simulation of coherent scattering of photons;
nIncoh – simulation of incoherent scattering of photons;
nPE – simulation of photoelectric absorption;
nPP – simulation of pair production;
nEsig – calculation of electron interaction cross sections;
nEdist – calculation of the distance to electron collision;
nEinter – sampling of chemical element and interaction type in the case of an electron collision;
nElast – simulation of elastic scattering of electrons;
nBrems – simulation of bremsstrahlung;
nExc – simulation of atomic excitation by electron collisions;
nIon – simulation of electron impact ionization;
nRelax – simulation of atomic relaxation;
nAnnih – simulation of positron annihilation;
nSource – generation of source particles;
nRead – reading banked particle data when a task in the “read” state at the start of a loop cycle.

The next 18 columns (i.e., columns 36 to 53) contain the time fractions corresponding to fractions of the number of calls given in the previous 18 columns. I.e., the only difference is that they give the fraction of simulation time spent for various states, instead of the fraction of the number of calls to various state handlers. The headers of those columns start with “t” instead of “n” (i.e., “tPsig”, “tPdist”, etc.).

The last two columns (No. 54 and No. 55) contain the time fractions spent for selecting the tasks to be processed in the current loop cycle (the corresponding column title is “tCheck1”) and for updating numbers of tasks corresponding to various states after exiting from a state handler (the corresponding column title is “tCheck2”). The former of the two mentioned steps is done at the start of a loop cycle by the first thread of the block, and the latter step is done at the end of a loop cycle by each thread of the block (in this step, multiple threads modify a number stored in one memory location, and such memory accesses are done serially).

If one of the options “stats_CUDA 2” or “stats_CUDA 3” has been specified in the input file or on the command line, then output of second-by-second CUDA device statistics to files can be stopped or resumed at any time during the simulation by pressing the ‘S’ or ‘F’ key on the keyboard, respectively.

#### *4.11.3. Summary CUDA device statistics in the output file*

If one of the options “stats_CUDA 1” or “stats_CUDA 3” has been used, then average values of most of the time fractions mentioned above will be written to the output file. Several other statistics will be output as well, e.g., the maximum number of banked particles for each CUDA device and for the CPU. ***Note***: In the case “heap_CUDA 2”, the maximum value of the *total* number

of banked particles on each CUDA device will be output, whereas in the case “heap_CUDA 0” and “heap_CUDA 1” the maximum number of banked particles *in one history* will be output.

Since most of the CUDA statistics written to the output file have descriptive labels explaining their meaning, a separate explanation is not needed here. For an explanation of other information included in the output file, see Section 6.

### *4.12. Procedural generation of MCNelectron input directives*

As mentioned in Section 4.4.2, the MCNelectron input file may contain blocks of the user’s code written in the MCNEcode programming language. This section provides a short overview of such code embedded in the input file, with emphasis on the features of MCNEcode that are most relevant for the original purpose of MCNelectron (the detailed specification of the MCNEcode programming language is in the Appendix). In the context of MCNelectron input directives, probably the most useful feature of MCNEcode is the loop operator “for”, which can significantly shorten the specification of multiple directives having similar format. Examples of using the “for” loop for defining complex geometries involving multiple transformations of the same object are given in Section 4.4.2.

#### *4.12.1. Embedding user programs in an MCNelectron input file*

When MCNelectron reads the input file, each code block is replaced internally by a set of input directives, where each directive corresponds to a call to the “dir” function. The “dir” function is invoked as follows:

dir(<"string1">, expression1, <"string2">, expression2, …)

Each string of characters must be written between double quotes. If the string itself includes the double quote, it must be written twice (""). The MCNEcode compiler/interpreter, which is included in the MCNelectron code, evaluates all expressions, formats their results as text, and concatenates all strings and the results of all expressions into a single directive, which is then passed for further checking and processing. The expressions used in the list of arguments of the function “dir” are not required to be separated from each other by strings, and the strings are not required to be separated from each other by expressions (as in the above specification of the argument list). If there are consecutive strings, they will be concatenated. The default format used for writing the numbers is defined by the C format specification “%.14g”, which means that the character string representing the number will have up to 14 significant digits, the trailing zeros after the decimal point will not be displayed, and the decimal exponent will be used if the number is less than $10^{-4}$ or greater than or equal to $10^{14}$. Any other C format specification for output of floating-point numbers may be used by calling the built-in function “SetNumberFormat” with the specification of the required format written in double quotes between parentheses, e.g.,

SetNumberFormat("%8.3f")

The numbers written after this call to “SetNumberFormat” will have three decimal places, the trailing zeros will be displayed, the decimal exponent will not be used and the number will be represented by at least 8 characters (spaces will be inserted before the number if its total length is less than 8). All calls to “dir” use the same number format, defined by the last call to “SetNumberFormat”, or the default format if there was no such call. If SetNumberFormat() is called without any arguments, then the default format will be restored. If an invalid number format is passed to “SetNumberFormat”, then the current format will not be changed.

The function “dir” automatically appends the newline character to the created line of text, which means that “dir” should be used to create either a complete directive or the last part of the current directive. The function “dir2” does not append the newline character (this is the only difference between “dir” and “dir2”). After a call to “dir2”, the created directive is not passed for further processing, and it is possible to append additional text to it by calling “dir” or “dir2” repeatedly. The created directive is processed only after a call to “dir” or after determining that it was the last directive of the current program.

If a string specified in the list of arguments of the function “dir” is followed by an expression, then the comma between the string and the expression may be omitted. For example, the following two statements are equivalent to each other:

```
dir("TR", k, "   0 0 0   ", i * 30, "    ", 15 + j * 30)
dir("TR"k, "   0 0 0   "i * 30, "   "15 + j * 30)
```

Another built-in function that is closely related to the “dir” function is the “print” (or “print2”) function. This function displays a line of text on the screen. It is used in the same way as the “dir” (or “dir2”) function. Another related function “write” will be described in Section 4.12.2.

By default, the directives created by the calls to the function “dir” are stored by MCNelectron internally as temporary strings of text and are not written to files. However, it is possible to create the so-called “log file”, which contains the entire set of processed MCNelectron input directives, including the procedurally generated ones. The log file is functionally equivalent to the MCNelectron input file used for creating it (i.e., if the original input file is replaced by the log file, the results of the simulation will be the same). In order to create the log file, the MCNelectron command-line argument “writeLog” must be used. This keyword must be followed by “1” if the log file must be created, or by “0” if it must not be created. The default name of the log file is obtained by appending “_log” to the name of the MCNelectron input file. Any other name of the log file may be specified after the MCNelectron command-line keyword “log”, or after the same keyword in the MCNelectron input file (see Section 4.1).

In addition to the variables defined in the user’s code, there are 10 “system variables”, which cannot be modified in the user’s code and which can be passed to MCNelectron as command-line arguments. The names of those system variables are @0, @1, @2, …, @9. In order to set the value of a system variable, the MCNelectron command line must specify the name of the system variable, followed by its value. This feature makes it possible to redo a simulation corresponding to a different set of conditions just by replacing a value of a system variable (or values of several system variables), without the need to change the input file. For example, the following file can be used to run the simulation of electron backscattering from any elemental material (with the atomic number defined by the system variable @0 and density defined by @1) at any energy of incident electrons (defined by the system variable @2):

*Example 1* (file “Backscattering.txt”)

```
parameters
Z = @0
E = @2
SetNumberFormat("%g")
dir("LOG Backscattering_Z="Z,"_E="E,"_log.txt")
dir("OUT Backscattering_Z="Z,"_E="E,"_out.txt")
end parameters

program Mat_Conc_E
SetNumberFormat()
dir("MAT  "Z,"  1")
dir("CONC "-@1)
dir(E)
end program

THICK 10000
CUT_E 5e-5
CUT_P 5e-5
PART E
N 1000
TASKS 8
```

The code block “PARAMETERS” in the last example differs from a “regular” program in only one respect: all variables appearing on the left of the assignment operator in the “PARAMETERS” block are treated as “global” parameters, i.e., they are visible in all other programs defined in the same file. If a variable was not defined as a global parameter, then it will be visible only in the program where it was assigned a value. The “PARAMETERS” block must precede all other code blocks of the same file, excluding the “ARRAYS” block (see below).

*4.12.2. Data arrays and using MCNelectron for batch processing*

The built-in function “system” executes a specified system command. The command line that is passed to the command-line interpreter of the operating system is formatted in the same way as in the “dir” function. The “system” function makes it possible to use the MCNelectron input file as a “batch” file for multiple simulations. For example, the following program runs the simulation of electron backscattering corresponding to the previous example at four values of the atomic number ($Z$ = 1, 11, 21, 31) and five values of energy ($E$ = 0.1, 0.316228, 1, 3.16228, 10 MeV):

*Example 2*:

```
program batch
for (Z = 1; Z <= 31; Z = Z + 10) {
      for (i = 0; i < 5; i = i + 1) {
            E = 0.1 * 10 ^ (i * 0.5)
            system("MCNelectron.exe in Backscattering.txt writeLog 1 @0 "Z," @1 1 @2 "E)
      }
}
end program
```

In this example, the physical quantitities that are being varied ($Z$ and $E$) can be easily calculated inside the program, because $Z$ and log($E$) are assigned equidistant values. In the case of an arbitrary set of values, it may be more convenient to first assign them to a data array, and then loop over the elements of that array. Arrays are defined in the code block “ARRAYS”, which must precede all other code blocks defined in the same file. Arrays are defined by specifying their data type, name and lengths of the dimensions. Eight types of array data are allowed. Their names and meanings are listed below:

double – 8-byte floating point,
float – 4-byte floating point,
int – 4-byte signed integer,
uint – 4-byte unsigned integer,
int2 – 2-byte signed integer,
uint2 – 2-byte unsigned integer,
int1 – 1-byte signed integer,
uint1 – 1-byte unsigned integer.

The lengths of the array dimensions must be listed between brackets after the array name; they are comma-delimited. If no initial values are given, all array elements will be initially set to zero. The initial values of array elements may be specified between braces after the equality character following the array definition (as in C). The number of array initializers may be less than the total number of array elements (then only the first elements of the array will be initialized). In the case of a multi-dimensional array, the array layout in computer memory is such that the adjacent elements of the array differ by the value of the last index (as in C). The arrays are global objects (i.e., they are visible in all programs defined in the same file). By default, the array indices in MCNelectron are one-based, which means that the smallest allowed value of an array index is 1, and the largest allowed value is equal to the length of the array dimension (specified in the definition of the array).

The array indices can be made zero-based (as in C) by inserting the directive “INDEX_BASE 0” at the beginning of the code block “ARRAYS”. Then the smallest allowed value of an array index would be 0, and the largest allowed value would be one less than the length of the array dimension.

Array elements are referenced in programs by specifying their indices between brackets after the array name (for multi-dimensional arrays, the indices must be comma-delimited). The following example extends the previous one by adding the “ARRAYS” block. In this example, the atomic number is assigned the values 1, 2, 4, 14, 22, 26, 29, whereas the energy is assigned the values 0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1, 2, 5, 10 MeV.

*Example 3*:

```
arrays
    int Z[7] = {1, 2, 4, 14, 22, 26, 29}
    double E[10] = {0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1, 2, 5, 10}
end arrays
program batch
for (j = 1; j <= 7; j = j + 1) {
    for (i = 1; i <= 10; i = i + 1) {
        system("MCNelectron.exe in Backscattering.txt writeLog 1 @0 "Z[j]," @1 1 @2 "E[i])
    }
}
end program
```

The system variables can be assigned values of array addresses. Then MCNelectron can access the elements of those arrays. This makes it possible to exchange any amount of data between the current instance of MCNelectron and the higher-level instance of MCNelectron (which defined the mentioned arrays and started the current instance of MCNelectron with a call to the built-in function “system”). The address of the first element of an array is returned by the built-in function “loc”, which must be called as follows:

```
address = loc(ArrayName).
```

The copying of data from a known address to an array is done using the built-in function “memcpy”, which must be called as follows:

```
memcpy(loc(destinationArray), sourceAddress, numberOfBytesToCopy).
```

The copying of data from an array to a known address is done similarly:

```
memcpy(destinationAddress, loc(sourceArray), numberOfBytesToCopy).
```

The “memcpy” function is not capable of checking if the specified number of bytes does not exceed the sizes of the source and destination arrays. It is the user’s responsibility to do so. If the specified number of bytes exceeds the size of the destination array, then the anomalous condition called “buffer overflow” will occur. Such an anomaly can make the code potentially insecure, because exploiting the buffer overflow is one of the security exploits.

The log file created by MCNelectron can be used as an input file for any other program. For example, the following MCNelectron input file can be used to create an MCNP6 input file for performing for the same simulation as in Example 1:

*Example 4* (file "MCNP6.txt")

```
Electron backscattering
program CELL_1 dir("1 1 "-@1,"   1 -2 -4 IMP:P=1 IMP:E=1")
      end program
2 0      3 -1 IMP:P=1 IMP:E=1
3 0      -3:2:(1 4) IMP:P=0 IMP:E=0

1 pz 0
2 pz 10000
3 pz -1
4 cz 10000

mode p e
PHYS:E 100 0 0 0 0 1 1 1 1 0 0 J J 0.917 100 $ use the single-event algorithm
program SDEF dir("sdef sur=3 pos=0 0 -1 par=3 DIR=1 erg="@2," rad=0")
      end program
RAND SEED=123
CUT:p 1e20 5e-5
CUT:e 1e20 5e-5
program M1 dir("m1 PLIB=12p "@0*1000," 1")
      end program
NPS  1000
```

In this example, the order of input directives required by MCNP is preserved by writing a separate one-line program for each of the three directives that use the three physical parameters passed to MCNelectron as command-line arguments: atomic number (variable @0), density (variable @1) and energy (variable @2). In order to create the corresponding MCNP input file, MCNelectron must be started with the command-line arguments "writeLog 1" and "check 0". The former argument ensures that the log file will be created (by default, MCNelectron does not create it), and the latter argument ensures that MCNelectron will not attempt to interpret either the lines of code generated by the calls to the "dir" function or the lines that are not inside user programs as MCNelectron input directives and that MCNelectron will quit immediately after reading the input file and running the user programs. The following example illustrates how the MCNP input files created by MCNelectron can be used to do the same set of simulations as in Example 3:

*Example 5:*

```
arrays
      int Z[7] = {1, 2, 4, 14, 22, 26, 29}
      double E[10] = {0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1, 2, 5, 10}
end arrays
program batch
for (j = 1; j <= 7; j = j + 1) {
      for (i = 1; i <= 10; i = i + 1) {
            system("MCNelectron.exe in MCNP6.txt check 0 writeLog 1 log
            MCNP6__Z_"Z[j],"__E_"E[i],"_in.txt @0 "Z[j]," @1 1 @2 "E[i])
            system("del MCNP6__Z_"Z[j],"__E_"E[i],"_out.txt")
            system("mcnp611 tasks 8 i=MCNP6__Z_"Z[j],"__E_"E[i],"_in.txt
            o=MCNP6__Z_"Z[j],"__E_"E[i],"_out.txt")
            system("del runtpe")
      }
}
end program
```

The element values of a one-dimensional array can be read from a user-specified ASCII file using the built-in function “ReadArray”. Before calling that function, the file must be opened for reading using the built-in function “fopenr”. The name of the file must be specified as an argument of the latter function between double quotes, e.g., “fopenr("Array data.txt")”. The string of text specified between parentheses after “fopenr” is interpreted in the same way as in the “dir” function, i.e., it may contain expressions. Only one file may be open for reading at a time. A repeated call to “fopenr” will close the previously-opened file and open a new one. If the file is opened successfully, then “fopenr” returns 1. If the attempt to open a file fails, then “fopenr” does not close the previously-opened file and returns 0 (if there is no file open for reading) or −1 (if another file was opened for reading earlier). In order to check if a file is open for reading, “fopenr” may be called without arguments (e.g., “rc = fopenr()”): if a file is open, then the returned value will be −1, otherwise it will be 0. The file that is currently open for reading can be closed by calling “fcloser()”.

The “ReadArray” function is called as follows: ReadArray(loc(ArrayName)), where “ArrayName” is the name of the one-dimensional array, and “loc” is the built-in function that returns the address of the first element of the array (all other information about the array, such as the array size, is also available to “ReadArray”). The “ReadArray” function treats the contents of the file that is open for reading as a set of “words” separated by spaces, tabs and line breaks, and converts each word to a number. If the word does not start with a number, then it will be converted to a zero. The “ReadArray” function stops when the number of processed words becomes equal to the number of the array elements, or when the end of the file is reached, whichever happens first. Another similar built-in function “ReadArray2” reads only the specified number of array elements. It is called as follows: ReadArray(loc(ArrayName), N), where “N” is the number of array elements to read. The function “ReadArray” (or “ReadArray2”) reads the file line-by-line, i.e., it reads entire lines and then parses them into words. Only the first 1000 characters are read from each line. Each new call to “ReadArray” starts with reading a new non-empty line (except when the current line is longer than 1000 characters). Consequently, if the previous non-empty line had unprocessed words, they are not accessible. It is possible to skip lines of the file by calling the built-in function “SkipLines(nLines)”, where “nLines” is the number of lines to skip. Both the “ReadArray” and “ReadArray2” functions return the number of values that were actually read from the file.

As mentioned, the built-in function “dir” can be used to output text to the MCNelectron log file. By default, the log file is not created and the text that is created by the “dir” function is stored only temporarily in the random-access memory (RAM) of the computer. In order to create the log file, the command-line option “writeLog 1” must be specified. It is also possible to output text to a user-specified file. This is achieved by first opening the file for writing and then using the built-in function “write” or “write2”. In order to open a file for writing, the built-in function “fopenw” must be called. It is used similarly to the previously-described function “fopenr”. If a file with the specified name already exists, then its contents will be destroyed. After the file is opened for writing, all calls to “write” will output text to that file. The argument list of the function “write” or “write2” is interpreted in exactly the same way as the argument list of the function “dir” or “dir2”. The function “write” appends the newline character to the output string, and the function “write2” does not append the newline character. There can be only one file open for writing at a time. The built-in function “fclosew” is used to close the file opened for writing. If there is no file open for writing, then the built-in functions “write” and “write2” are equivalent to “dir” and “dir2”, i.e., their output is interpreted as MCNelectron input directives (which are written to the log file if the mentioned command-line option “writeLog 1” was specified). It is possible to check if a file is open for writing by calling “fopenw” without any arguments (for example, “rc = fopenw()”): if a file is open for writing, then the returned value will be −1, otherwise it will be 0.

The input file for the program invoked by a call to the built-in function “system” can be created “from scratch”, using the “write” function as described in the previous paragraph. This method would eliminate the need for the intermediate call to MCNelectron for creating the log file in a batch run (in Example 5, this is the first call to “system” inside the loop), if the program that is used for the simulations is not MCNelectron. However, this would require calling “write” for every

input directive, whereas the method involving the MCNelectron log file requires calling "dir" only for the directives whose parameters are controlled programmatically (see Example 4).

The first line of the MCNelectron log file always contains a directive. That is to say, the log file never includes the initial empty lines or the initial lines with comments. I.e., if the input file starts with an empty line or with the line containing only a comment, or if it starts with a program whose output starts with an empty line or a comment line, then that line will be absent in the log file. However, the empty lines and comment lines that are after the initial directive are not excluded.

#### *4.12.3. Similarities and differences between MCNEcode and C*

As illustrated by the above examples, the syntax of the MCNEcode programming language is in some respects similar to the syntax of the C language. The common syntax elements of the two languages include the headers of the "for", "while" and "if/else" operators, usage of braces in the "bodies" of the mentioned operators, usage of the semicolons (in MCNEcode, the semicolons are optional, except when there are multiple statements on a single line), comparison operators, definitions of one-dimensional arrays and references to them. The main differences between MCNEcode and C are the following:

1. All variables in MCNEcode are of the 8-byte floating-point type (equivalent to the type "double" in C). Although the array elements can be of other types (see Section 4.12.2), they are converted to 8-byte floating point numbers "on the fly" in all arithmetic operations.
2. New variables in MCNEcode do not require declarations: a new variable is defined by assigning a value to it.
3. MCNEcode does not have unary operators, except for the unary minus. For example, the C expression "i++", which increases the value of the variable "i" by 1, must be replaced in MCNEcode with "i = i + 1".
4. The C logical operators "&&"and "||" (meaning "AND" and "OR", respectively) are replaced in MCNEcode by the operators "&" and "|" (which are used in C for bitwise operations).
5. In MCNEcode, the array indices are separated by commas (in C, they are separated by "][").
6. As mentioned, the array indices in MCNEcode are one-based by default (however, there is an option to make them zero-based as in C).
7. The identifiers of variables, arrays and functions in MCNEcode are case-insensitive, whereas in C they are case-sensitive.
8. The identifiers in MCNEcode cannot be longer than 50 characters (in C, there is no such limitation).
9. Unlike in C, there are no string variables in MCNEcode, whereas string constants can be used only in the list of arguments of the above-mentioned built-in functions "dir", "dir2", "write", "write2", "print", "print2", "system", "fopenw", "fopenr" and "SetNumberFormat".
10. MCNEcode is a much simpler language than C and does not have many other capabilities and keywords that C has (such as the keywords "break", "continue" and "goto").

A detailed description of all features of the MCNEcode language (excluding the built-in functions for data output and input, which were described in this Section) is presented in the Appendix.

## 5. Simulation of physical effects that are not included in specification of the ENDF/B library

In addition to various types of interactions and effects whose cross sections and probabilities are stored in the ENDF/B data library [7], MCNelectron simulates seven physical effects, for which the ENDF/B-VII.1 library provides neither tabular probabilities nor an analytic prescription:

(a) directions of motion of the two electrons existing after an impact ionization event,
(b) non-isotropic angular distribution of bremsstrahlung photons,
(c) partial suppression of positron bremsstrahlung in comparison with electron bremsstrahlung,
(d) non-isotropic angular distribution of photoelectrons,
(e) non-uniform distribution of positron and electron energies during pair production,
(f) angular distribution of electrons and positrons created in pair production events,
(g) Doppler broadening of energy distribution of incoherently scattered photons.

For six of those effects (b – g), there is a keyword in the MCNelectron input file, which allows "turning on" or "turning off" simulation of the effect. By default, all those effects are simulated. If any of those effects is "turned off", then it will be replaced by a simplified model (e.g., isotropic bremsstrahlung) or neglected completely. The mentioned keywords, listed in the same order as the corresponding effects, are the following:

(b) BREMS_ANGULAR_DISTR,
(c) BREMS_POSITRON_CORRECTION,
(d) PE_ANGULAR_DISTR,
(e) PP_ENERGY_DISTR,
(f) PP_ANGULAR_DISTR,
(g) INCOH_DOPPLER.

For some of those effects, there are other keywords controlling some details of the simulation. Those keywords are explained in the descriptions of the methods used for simulating each of the mentioned effects, which are provided below.

(a) Directions of motion of the two electrons existing after an impact ionization event

As in MCNP6 [1], the angles between the direction of motion of the incident electron and the directions of motion of the primary and secondary electrons after an impact ionization event are uniquely determined by conservation of momentum. There is also the same minor departure from the otherwise analog simulation as in MCNP6, i.e., the azimuth angles of the two electrons existing after an impact ionization event are sampled independently, so that their direction vectors do not lie in the same plane as the direction vector of the incident electron [1]. Further on, the electron with the lower kinetic energy will be called the "secondary" or "knock-on" electron, and the other electron existing after the impact ionization event will be called the "primary" electron. The cosine of the scattering angle (i.e., the angle between the direction vectors of the primary and incident electrons) is calculated as follows:

$$\cos\theta_1 = \sqrt{\frac{(E_0 - W)\cdot(E_0 + 2mc^2)}{E_0\cdot(E_0 - W + 2mc^2)}}\,, \tag{5.1}$$

where $E_0$ is the kinetic energy of the incident electron, $W$ is the energy transfer (the sum of the kinetic energy of the secondary electron and the binding energy of the electronic subshell where a vacancy has been created), $m$ is the electron rest mass, and $c$ is the speed of light. The cosine of the recoil angle (i.e., the angle between the direction vectors of the secondary and incident electrons) is calculated as follows:

$$\cos\theta_2 = \frac{p_0'^2 - p_1^2 + p_2^2}{2\,p_0' p_2}\,, \tag{5.2}$$

where $p_1$ and $p_2$ are the momenta of the primary and secondary electrons, respectively, and $p'_0$ is the "reduced" momentum of the incident electron, corresponding to the kinetic energy $E_0$ reduced by the binding energy ($E_0 - B$). The relativistic relation between momentum $p$ and kinetic energy $E$ is

$$p = \frac{1}{c}\sqrt{E(E + 2mc^2)}\,. \tag{5.3}$$

(b) Non-isotropic angular distribution of bremsstrahlung photons

If the integer number that follows the keyword "BREMS_ANGULAR_DISTR" is non-zero, then bremsstrahlung photons will be emitted non-isotropically, with angular probability density depending both on incident electron energy and on emitted photon energy (this is the default behavior). Otherwise, the bremsstrahlung photons will be emitted isotropically. In any case, the electron's direction is unchanged by the bremsstrahlung event. Simulation of bremsstrahlung angular distribution requires pre-computed values of angular probability density of bremsstrahlung. Those values are in the file "Brems_angular_prob_dens.dat", which is in subfolder "Data" of the

MCNelectron distribution package. The probability density values that are stored in that file were calculated by integrating analytically over the triply differential cross-sections derived by Bethe and Heitler. Those calculations were done using a code that is a supplement to the article [8] (that code was downloaded from the website of the journal where that article was published). The data in the file "Brems_angular_prob_dens.dat" are in binary format. Below is a description of that format (nested sequences are indicated by indentations; the lines starting with the words "Sequence" and "End of sequence" are only included for clarity and are not represented in the data):

First 400 bytes of the file contain the starting positions in this file for each of 100 chemical elements,
Sequence 1, which is repeated 100 times (once for each chemical element):
    4 bytes contain the number of incident electron energies ("nInc"),
    Sequence 2, which is repeated nInc times:
        8 bytes contain the incident electron energy (eV),
        4 bytes contain the number of bremsstrahlung photon energies ("nBrem"),
        Sequence 3, which is repeated nBrem times:
            8 bytes contain the bremsstrahlung photon energy (eV),
            4 bytes contain the number of probability density values ("nProb"),
            Sequence 4, which is repeated nProb times:
                8 bytes contain the value of the cosine of the photon emission angle,
                8 bytes contain the corresponding value of the probability density.
            End of sequence 4
        End of sequence 3
    End of sequence 2
End of sequence 1

The MCNelectron package includes the executable "Data\ExtractBremsAngularDistr.exe", which extracts ASCII data from the mentioned binary file and creates a subfolder "Brems_angular_prob_dens" with 100 human-readable files (a separate file for each chemical element). The meaning and order of numbers in each of those text files is the same as in a single "Sequence 1" in the above description of the binary format. Those text files can be used as input data for the simulation, too, instead of the mentioned binary file. This is controlled by the keyword "BREMS_BINARY_DATA". If the integer number that follows that keyword is non-zero, then the data will be loaded from the mentioned binary file "Data\Brems_angular_prob_dens.dat" (this is the default behavior). Otherwise, the data will be loaded from the mentioned ASCII files. In the latter case, the user has an option to specify the folder with those text files. The folder name must be specified after the keyword "BREMS_ANGULAR_DISTR_DIR" in the input file. The default name of the folder with the ASCII files containing the angular probability densities of bremsstrahlung is "C:\Brems_angular_prob_dens\".

In the case of simulation of non-isotropic bremsstrahlung, the mentioned additional probability density data are used only when the incident electron energy ($E$) belongs to a pre-defined energy interval [$E_{min}$, $E_{max}$]. When $E < E_{min}$ or $E > E_{max}$, then a simple analytical probability distribution

$$p(\mu)\,\mathrm{d}\mu = \tfrac{1}{2}\,(1-\beta^2)\,/\,(1-\beta\,\mu)^2\,\mathrm{d}\mu \qquad (5.4)$$

is used, where $\beta$ is the ratio of the electron speed and the speed of light, and $\mu$ is the cosine of the angular deflection of the photon relative to velocity vector of the incident electron or positron. The default values of $E_{min}$ and $E_{max}$ are 1 keV and 1 GeV, respectively. Due to simplicity of the function $p(\mu)$ defined by Eq. (5.4), the values of $\mu$ corresponding to that distribution are sampled analytically, by solving the following equation:

$$\int_{-1}^{\mu} p(\mu')\mathrm{d}\mu' = r\,, \qquad (5.5)$$

where $r$ is a random number that is uniformly distributed between 0 and 1. By substituting the expression (5.4) into the integral (5.5) and solving for $\mu$, the following expression of the sampled value of $\mu$ is obtained:

$$\mu = \frac{2r + \beta - 1}{\beta(2r-1)+1}. \tag{5.6}$$

The mentioned default value of $E_{\min}$ (i.e., 1 keV) is mainly for comparison with MCNP6: as stated in [1], the analytical distribution defined by by Eq. (5.4) "is not really appropriate for low energies, and its presence in the current code is a temporary expedient". The user may set other values of $E_{\min}$ and $E_{\max}$. $E_{\min}$ must be specified in the input file after the keyword "BREMS_ANGULAR_DISTR_LOW_EN", and $E_{\max}$ must be specified after the keyword "BREMS_ANGULAR_DISTR_HIGH_EN" (the unit of energy is the same as for all other energies specified in the input file, i.e., it is defined by the keyword "E_UNIT" described n Section 4.1). In order to use a more realistic angular distribution of bremsstrahlung photons at low energies, $E_{\min}$ should be set to zero. If necessary, the analytical distribution defined by Eq. (5.4) can be applied for all values of electron energy. This is achieved by setting $E_{\min} >= E_{\max}$ (for example, by setting $E_{\max} = 0$). Then the values of $E_{\min}$ and $E_{\max}$ are not used during the simulation, and the data files with bremsstrahlung angular probability density data are not loaded (in such a case, the text entered after the keywords "BREMS_BINARY_DATA" and "BREMS_ANGULAR_DISTR_DIR" is ignored).

(c) Partial suppression of positron bremsstrahlung in comparison with electron bremsstrahlung

As reported in [9], the ratio of the radiative stopping powers for positrons and electrons can be approximated as a function only of the variable $E / Z^2$, where $E$ is the energy of the incident electron or positron and $Z$ is the atomic number. The values of that ratio, which are tabulated in [9], can be approximated by the following function of the variable $x = \ln(E / Z^2) - 5.87522$, where $E$ is expressed in units of eV:

$$\frac{\Phi^{+}_{\mathrm{rad}}}{\Phi^{-}_{\mathrm{rad}}} = \begin{cases} 0.020 & (E/Z^2 \le 0.1), \\ 0.606133 + 0.140534x + 0.0102012x^2 + 0.000217602x^3 & (\ln(0.1) - 5.87522 < x \le 0), \\ 0.606133 + 0.140534x - 0.0163087x^2 + 0.000609304x^3 & (0 < x < \ln(5\cdot 10^5) - 5.87522), \\ 1 & (E/Z^2 \ge 5\cdot 10^5). \end{cases} \tag{5.7}$$

The relative deviation of the values calculated according to (5.7) from the values tabulated in [9] is less than 1.4 % for all values of $E$.

If the current particle is a positron, then the mentioned partial suppression of positron bremsstrahlung is taken into account by multiplying the electron bremsstrahlung cross section by the factor calculated according to Eq. (5.7). This is the default behavior. The user may "turn off" simulation of this effect using the keyword BREMS_POSITRON_CORRECTION in the MCNelectron input file. If the number that follows that keyword is zero, then this effect will not be simulated, i.e., the positron bremsstrahlung cross section will be equal to the electron bremsstrahlung cross section.

(d) Non-isotropic angular distribution of photoelectrons

If the integer number that follows the keyword "PE_ANGULAR_DISTR" is non-zero, then photoelectrons will be emitted non-isotropically (this is the default behavior). Otherwise, the photoelectrons will be emitted isotropically. In the case of non-isotropic emission of photoelectrons, two forms of their angular distribution are used, depending on the energy of the emitted electron. For electron energies below 50 keV, the angular probability density function $p(\mu)$ is of the form

$$p(\mu)\mathrm{d}\mu \sim (1-\mu^2)[\mathrm{Im}((\alpha + \mathrm{i}\cdot s)^{-n'-1})/s]^2 \mathrm{d}\mu, \tag{5.8}$$

where $\mu$ is the cosine of the angle between incident photon and photoelectron wave vectors, $s$ is the absolute value of the vector that is equal to the difference of those two wave vectors, "i" is the imaginary unit, and notation "Im(…)" means the imaginary part of a complex number. If the electron is ejected from one of the first 4 shells (K, L, M or N), then $n'$ coincides with the principal quantum number ($n$) of that shell. For higher shells, $n'$ is fixed at 4. $\alpha$ is the factor in the exponent

of the approximate (nodeless) wave function of the electrons in that shell. The approximate wave function is of the form

$$\psi(r) \sim r^{n'-1} \mathrm{e}^{-\alpha r}, \tag{5.9}$$

$$\alpha = \frac{Z - \sigma}{n^* a_\mathrm{B}}, \tag{5.10}$$

where $Z$ is the atomic number, $\sigma$ is the screening constant, $n^*$ is the effective quantum number and $a_\mathrm{B}$ is the Bohr radius. The values of $n^*$ and $\sigma$ are calculated according to the Slater rules [10]. Eq. (5.8) has been obtained by calculating the matrix element of the perturbation operator describing the incident electromagnetic plane wave, when the initial state of the electron is described by the wave function (5.9) and the final state of the electron is described by another plane wave. In [11], this procedure was applied to the case when $n^* = n' = n = 1$ (K electrons). The angular distribution corresponding to the latter case is called "the Fischer distribution".

For photoelectron energies above 50 keV, the angular probability density function $p(\mu)$ is

$$p(\mu)\mathrm{d}\mu \sim \frac{1-\mu^2}{(1-\beta\mu)^4}\left[1 + \frac{1}{2}\gamma(\gamma^2-1)(1-\beta\mu)\right]\mathrm{d}\mu, \tag{5.11}$$

where $\gamma$ is the ratio of the photoelectron kinetic energy and the electron rest energy, $\beta$ is the ratio of the electron speed and the speed of light, and $\mu$ is the cosine of the angle between directions of the incident photon and the photoelectron. The distribution defined by Eq. (5.11) is called "the Sauter distribution" [12].

The mentioned value of 50 keV, which corresponds to transition from the low-energy angular probability density (5.8) to the high-energy angular probability density (5.11), is the same as in MCNP, where a similar (although probably not exactly the same) approach has been applied [1, 13]. This "transition energy" may be modified by the user. Its value must be specified in the input file after the keyword "PE_ANGULAR_DISTR_THR" (the unit of energy is the same as for all other energies specified in the input file, i.e., it is defined by the keyword "E_UNIT" described n Section 4.1).

In addition, it is possible to modify the number of values of $\mu$ in the tables of photoelectron angular probability densities, which are calculated by MCNelectron before the simulation. The default value of that number is 21 (i.e., $\mu$ is varied in increments of 0.1 from −1 to 1). In order to change that number, it must be specified in the input file after the keyword "PE_ANGULAR_DISTR_N". In any case, the values of $\mu$ will be equidistant.

(e) Non-uniform distribution of positron and electron energies during pair production

If the integer number that follows the keyword "PP_ENERGY_DISTR" is non-zero, then the energy distribution of positrons emitted when a photon with energy $E$ is absorbed in a pair-production event is non-uniform (this is the default behavior). Otherwise, the positron energy will be sampled from a uniform distribution. Once the positron energy is sampled, the electron energy is uniquely determined by conservation of energy.

The non-uniform energy distribution function is calculated using the following formula, which is an approximation of the Bether-Heitler singly differential cross section for pair production:

$$\frac{\mathrm{d}\sigma}{\mathrm{d}x} \sim [x^2 + (1-x)^2]\varphi_1 + \frac{2}{3}x(1-x)\varphi_2, \tag{5.12}$$

where $x = (E_+ + m_\mathrm{e}c^2) / E$ is the reduced energy, i.e., the fraction of the photon energy carried off by the positron ($E_+$ is the positron kinetic energy, $m_\mathrm{e}$ is the electron rest mass, $c$ is the speed of light), and factors $\varphi_1$ and $\varphi_2$ are defined as follows:

$$\varphi_1 = 2[1 + \ln(a^2)] - 2\ln(1+b^2) - 4b\arctan(b^{-1}), \tag{5.13a}$$

$$\varphi_2 = 2[(2/3) + \ln(a^2)] - 2\ln(1+b^2) + 8b^2[1 - b\arctan(b^{-1}) - 0.75\ln(1+b^{-2})], \tag{5.13b}$$

where $a$ depends only on the atomic number $Z$:

$a = 122.8$ for $Z = 1$,
$a = 90.8$ for $Z = 2$,
$a = 100$ for $Z = 3$,
$a = 106$ for $Z = 4$,
$a = 111.7$ for $Z > 4$,

and $b$ is defined as follows:

$$b = \frac{a \cdot Z^{-1/3} m_e c^2 E}{2(E_+ + m_e c^2)(E - E_+ - m_e c^2)}. \qquad (5.13c)$$

The expression (5.12) has been obtained from the formula provided in [12] by eliminating the terms proportional to the high-energy Coulomb correction (as stated in [12], "because of the approximate nature of this correction, it should not be used for photon energies of less than about 100 MeV"). The two terms of Eq. (5.12), which are proportional to $\varphi_1$ and $\varphi_2$, are the same as in a more complex formula derived in [14]. However, the expressions of the factors $\varphi_1$ and $\varphi_2$ that are given in [12] are slightly different from their definitions provided in [14]. The expressions (5.13a) and (5.13b) have been taken from [14].

The argument of the positron energy probability density that is used during simulation is defined slightly differently than the reduced energy $x$ in Eq. (5.12): it is the fraction $y$ of the total *kinetic* energy $T_{max} = E - 2m_e c^2$ that is carried away by the positron. Unlike $x$, the variable $y$ can be equal to any number from 0 to 1. Before starting the simulation, MCNelectron calculates tables of positron energy probability densities $p(y)$ corresponding to 12 values of the incident photon energy from 1022 keV to 100 GeV. During simulation, the probability densities corresponding to the current photon energy are calculated by interpolation.

(f) Angular distribution of electrons and positrons created in pair production events

If the integer number that follows the keyword "PP_ANGULAR_DISTR" is non-zero, then the directions of positrons and electrons emitted in pair production events will be sampled using the probability density function $p(\mu)$ defined by Eq. (5.4), where $\beta$ is the ratio of the particle speed (i.e., the speed of the positron or the electron) and the speed of light [12]. The corresponding formula for calculating the sampled values of $\mu$ is (5.6). This is the default behavior. Otherwise (i.e., in the case "PP_ANGULAR_DISTR 0"), both the positron and the electron will continue in the direction of the incident photon.

(g) Doppler broadening of energy distribution of incoherently scattered photons

If the integer number that follows the keyword "INCOH_DOPPLER" is non-zero, then MCNelectron simulates the so-called Doppler broadening of the energy spectrum of incoherently scattered photons (caused by the distribution of the electron momentum inside the atom). This is the default behavior. Otherwise, the energies of the scattered photon and the recoil electron will be calculated as though the electron is initially free and at rest. Doppler energy broadening is simulated exactly as described in [15, 16]. This simulation requires the so-called Compton profiles for each subshell of each chemical element that is present in the target material. Those Compton profiles were published in [17]. They are stored in the ASCII file "Data\ComptonProfiles.dat", which is included in the MCNelectron distribution package. That file was downloaded from http://ftp.esrf.eu/pub/scisoft/xop2.3/DabaxFiles/ComptonProfiles.dat .

Another aspect of incoherent scattering of photons is controlled by the keyword "INCOH_SUBTRACT_BINDING_E". If the integer number following that keyword is non-zero, then the recoil electron energy is additionally reduced by subtracting the binding energy of the subshell from which the electron was ejected (if the resulting energy value is negative, then the recoil electron is not emitted). This is the default behavior. Otherwise, the recoil electron energy is simply the difference of the incident and scattered photon energies. Thus, in the case with "INCOH_SUBTRACT_BINDING_E 0" and with simulation of Doppler energy broadening "turned

on", conservation of energy and momentum is not simulated accurately (except in the high-energy limit, when atomic binding effects become negligible): the scattered photon energy is calculated under the assumption that the electron initially has some momentum, whereas the recoil electron energy is calculated under the assumption that its initial energy (and momentum) is zero. Another non-physical result of such simple calculation of the recoil electron energy is related to possible subsequent emission of X-ray photons or Auger electrons from the ionized atom: those secondary photons and electrons also carry off some energy and interact with the target material. Thus, the total energy carried off by the secondary particles (the recoil electron and the secondary photons and electrons from atomic relaxation) is slightly greater than the energy loss of the photon. This result violates conservation of energy. If the integer number following the keyword "INCOH_SUBTRACT_BINDING_E" is non-zero, then the violation of conservation of energy is eliminated: the mentioned binding energy is, in effect, the excitation energy of the ionized atom, and a part of that excitation energy may be carried off by Compton fluorescence photons and Auger electrons.

In any case, the direction of motion of the recoil electron is the same as the direction of the photon momentum transfer vector (i.e., the difference of the incident and scattered photon momentum vectors). This also is only accurate in the high-energy limit: since the mass of the target atom is much larger than the electron mass, the atom can absorb any amount of recoil momentum and, consequently, the directions of the scattered photon and the electron are not constrained by kinematics [12].

## 6. MCNelectron output file format

An MCNelectron or MCNelectron_CUDA output file consists of the following 10 sections:

1) the command line that was used to generate the current file,
2) contents of the input file and of the alternative cross sections info file (if it was used),
3) a table with concentrations and densities of materials, as well as with cell volumes and masses,
4) two tables with some geometry information,
5) the elapsed time and the maximum number of banked particles,
6) the minimum and maximum coordinates of randomly generated source points (only when the directive "SOURCE_POSITION 3" is present in the input file; see Section 4.2),
7) CUDA device statistics (optional; see Section 4.11),
8) two tables with information about photon and electron activity in each cell,
9) tables with summary statistics characterizing various types of interactions,
10) average energy loss per one secondary electron.

Tables No. 3, 4, 6 and 8 are present only in complex geometry mode. The atomic concentrations are given in $cm^{-3}$, and densities are in $g/cm^3$. The values of cell volumes ($cm^3$) and masses (g) are provided only for the cells with a known volume. I.e., the volume must have been either specified by the user in the cell definition using the keyword "VOL", or estimated by MCNelectron. The program can calculate the volumes and masses only for the finite cells that are "bracketed" by coordinate planes along all three Cartesian axes. Those estimates of the volume and mass are approximate, except for rectangular cells whose faces coincide with the mentioned coordinate planes. The absolute standard deviations of the volume and mass are given in columns "Volume_err" and "Mass_err", whereas the relative standard deviations are given in columns "Vol_err%" and "Mass_err%", respectively. For the cells with user-specified volume, the standard deviations are zero.

The first of the mentioned two tables with geometry information contains the positions of coordinate planes bracketing each cell. The second table with geometry information contains the minimum and maximum coordinates of the points that were randomly generated inside finite cells for estimation of cell volumes (this is done only for the cells that are bracketed by coordinate planes along all three Cartesian axes).

Each of the two tables with information about particle activity in each cell contains numbers of entry and escape events, the corresponding average energies of entering or escaping photons and electrons, numbers of photon and electron collisions in each cell, and the average energy loss per collision. In addition, the electron activity table has two columns with the values of the total absorbed energy and the average absorbed dose in each cell (the unit of dose is Gy). The absorbed doses are given only for the cells with a known finite mass.

Information about various types of interactions is presented in six tables: three tables with information pertaining to creation and interaction events of photons and three tables with information pertaining to creation and interaction events of electrons and positrons. Each of those six tables consists of three columns: number of interaction events, their weight per source particle and the total energy of secondary particles created in those events or the total energy lost in those events (the energy is also given per source particle). All statistics under "electrons" include the contribution from positrons, too. In simple geometry mode (except when the option "TRACKPOS 0" is used), there are two additional lines before those tables, with the values of the number of source particles that have been scattered or absorbed in the layer and the value of source energy that has been absorbed.

Among the six tables with interaction data, there are two tables pertaining to photon creation and two tables pertaining to electron creation. The first of those two tables contains only the statistics for the events when the energy of the created photon or electron exceeded the low-energy cutoff value specified in the input file (see Section 4.1, keywords CUT_P and CUT_E, respectively). The second of those two tables contains statistics for all particle-creation events, including those when the energy of the created particle was less than the cutoff value. The first of those two tables is mainly included for comparison with MCNP6 output, because, as mentioned in Section 1, MCNP6 outputs only the statistics pertaining to particles with energy greater than the low-energy cutoff.

In order to facilitate comparison with MCNP6 output, the names of the rows in the mentioned tables and their order are similar to those in the MCNP6 output file. There is only one item in MCNelectron output that is absent in the MCNP6 output: the tables with particle interaction events have a row corresponding to the events that are not accompanied by energy loss, i.e., coherent scattering of photons and elastic scattering of electrons. Another difference between the tables with energy loss information in MCNelectron and MCNP6 output files is in the meaning of the number "N" (column "tracks" in MCNP6 output). In MCNP6 output, that number is the actual number of lost particles. Electrons and positrons can be lost only by three mechanisms:

- escape from the system,
- decrease of electron energy below the low-energy cutoff value,
- positron annihilation (it occurs after the positron energy drops below the low-energy cutoff).

Photons can be lost only by four mechanisms:

- escape from the system,
- decrease of photon energy below the low-energy cutoff value,
- photoelectric absorption ("capture"),
- pair production.

Consequently, the number of events given for all other energy loss mechanisms in the MCNP6 output file is zero. Conversely, MCNelectron outputs the number of *all* interaction events. Consequently, the number in the column "N" of the fifth table is non-zero whenever the number of interaction events of the corresponding type is non-zero, even when that type of events did not cause loss of the incident particle (in such a case, only the energy loss values can be compared between MCNelectron and MCNP6 output). However, for comparison with MCNP6 output, the number in the row "Total loss" of MCNelectron output files includes only the events when particles were actually lost.

The last line contains the average energy loss per one secondary electron (the so-called "*W*-value"). Two values of that energy loss are given: 1) taking into account only the electrons with

energy greater than the low-energy cutoff value; 2) taking into account all secondary electrons, including those that were not tracked due to low-energy cutoff. Obviously, the latter value is closer to the true *W*-value of the target material.

If the simulation is performed in complex geometry mode and output of the overall cell-by-cell interaction statistics is required (see the description of the keyword OUTPUT_CELL_INTERACTION_STATS in Section 4.2), then the same sets of statistics are output for each cell of the simulated system, too. The cell-by-cell interaction statistics are written to a separate file, which is placed into the same subfolder as the files with tally data (the subfolder name format is "Files_<output_file_name>"). The name of the file with cell-by-cell interaction statistics is obtained by appending "_cellInteractionStats" to the name of the main output file. The tables with interaction statistics for individual cells have one additional item in comparison with the corresponding tables in the main output file: the particle creation tables for individual cells have the row "entry", and the particle loss tables have the row "source escape". The difference between the rows "entry" and "source entry" is that the former provides information about all events of a particle's entry into a given cell, whereas the latter provides information only about the particles that entered a cell without any intervening interactions since their emission from the source (i.e., if they had passed through any other cells before the entry, they had not collided with any atom). The source particles that did not enter any cell are not included in the statistics (the number of such "lost" source particles is given in the same line where the total simulation time is shown). The difference between the rows "escape" and "source escape" in the particle loss tables for individual cells is that the former provides information about all events of a particle's escape from a given cell, whereas the latter provides information only about the source particles that escaped the given cell without any interactions since their entry into that cell. The numbers in the row "escape" have a slightly different meaning in the sets of the overall interaction statistics for the entire system and for individual cells: in the case of the entire system, only the particles that escaped the system are counted, whereas in the case of individual cells, any escape event is counted, including the events followed by entry into another cell (or another union of cells), or into the same cell (or the same union of cells).

## 7. Information about MCNelectron test files

Folder "Test" of the MCNelectron distribution package contains two subfolders: "Samples" and "Verification_for_K_and_L_X-rays". Subfolder "Samples" contains several MCNelectron and MCNP6 input files corresponding to identical simulation conditions, demonstrating the agreement between the results produced by the two codes. Subfolder "Verification_for_K_and_L_X-rays" contains the data used for a rigorous statistical verification of MCNelectron for simulations of characteristic K and L X-ray emission. This verification is based on a comparison between the X-ray photon counts computed by MCNelectron and MCNP6.1, using the method of the $\chi^2$ test ("chi-squared test"). An Excel file with the values of the photon count and the calculated summary statistics is also in subfolder "Verification_for_K_and_L_X-rays".

Each of the mentioned subfolders contains a pdf file with detailed information about the files contained in the same subfolder. The same information is presented below.

### *7.1. Information about files in subfolder "Test\Samples" of the MCNelectron distribution package*

Subfolder "Test\Samples" contains eleven MCNelectron input files and eleven MCNP6 input files corresponding to identical simulation conditions, together with respective output files. In addition, there is an MCNelectron input file where the option "ION_DWBA 1" is specified, and the corresponding output file (since this option is not available in MCNP, there are no corresponding MCNP6 input and output files). Input file names start with the word "input", and output file names start with the word "output". Six of the mentioned 11 simulations use a source of monoenergetic electrons, three simulations use a source of monoenergetic photons (the names of the corresponding

MCNelectron input files end with "P"), one simulation uses a source of X-ray photons with continuous spectrum (the name of the corresponding MCNelectron input file ends with "X-rays"), and one simulation uses a source of monoenergetic positrons (the name of the corresponding MCNelectron input file ends with "E+"). The names of MCNP6 input and output files are obtained by appending "_MCNP6" to the names of the corresponding MCNelectron input and output files.

Each of the 12 simulations using MCNelectron can be run either manually (by entering a command in the console window), or by executing one of the batch files, which are in the "Test\Samples" subfolder. The file "Batch_E.bat" runs the simulations where the source particles are electrons or positrons, and the file "Batch_P.bat" runs the simulations where the source particles are photons. The other two batch files ("Batch_E_MCNP6.bat" and "Batch_P_MCNP6.bat") attempt to run the corresponding simulations using MCNP6 in single-event mode.

A more detailed information about the test simulations is presented below.

In two of the test simulations, the main aim is estimation of the average energy per one secondary electron (the so called "*W*-value") when the target material completely absorbs energy of incident monoenergetic electrons. In both those simulations, the target material is pure xenon. Both those simulations make use of the ability of MCNelectron to "turn off" elastic scattering of electrons and tracking of particle coordinates (this makes the simulation much shorter). The corresponding MCNelectron input files are:

"input_Xe_1MeV_10.96_W-value_E.txt": energy of incident electrons is 1 MeV,
"input_Xe_100MeV_10.96_W-value_E.txt": energy of incident electrons is 100 MeV.

The number after the energy value in those two file names is the cutoff energy for electrons (eV). It is equal to ionization energy of xenon (as stored in the ENDF/B library). In order to ensure that all energy of source electrons is absorbed by the target material when MCNP6 is used, the geometry of the simulated system is defined in the MCNP6 input files as an isotropic point source at the center of a sphere with radius $10^{12}$ cm, which is filled with xenon at a density of 1 g/cm$^3$.

In six test simulations, interaction of source particles with a layer composed of Xe, Ar and He with atomic fractions 0.1, 0.2, 0.7 is simulated (since the aim of those simulations is comparison of MCNelectron output with MCNP6 output, the composition of the target material has been chosen arbitrarily, without any regard to its practical or scientific value). Density of the target material is 1 g/cm$^3$ and thickness is 1 cm, excluding the simulation with the continuous-spectrum X-ray source, where the thickness is $10^{-5}$ cm. It should be noted that results of the simulation only depend on the value of mass thickness $\rho d$, where $\rho$ is density and $d$ is thickness of the layer. Three of those six simulations use a monoenergetic source emitting particles with energy 1 MeV, and only differ by the particle type (electrons, photons or positrons). The fourth and fifth simulations use a monoenergetic source that emits photons or electrons with energy 100 MeV. Those two simulations are the most complex ones, because all physical processes and effects that MCNelectron can simulate are well represented in those two simulations. The sixth simulation uses a realistic spectrum of X-rays emitted by a Mo-tube with anode voltage 35 kV and filtered by a layer of Pyrex glass with thickness 1.2 mm.

The 9$^{th}$ simulation setup is a 25 keV electron beam that is incident normally on a 2 μm layer of $U_3O_8$ with density 8.3 g/cm$^3$. This is one of the two simulations using a plane-crossing tally (a table of energy distribution of photons escaping the layer through the surface exposed to the incident beam). There are two versions of that simulation: one of them uses the option "ION_DWBA 0", and another one uses "ION_DWBA 1". The MCNP6 counterpart exists only for the version with "ION_DWBA 0".

The last two simulations are the only ones using the "complex geometry" option in MCNelectron. The simulation setup corresponding to the MCNelectron input file with the name "input_Geometry_test.txt" consists of three cells and an isotropic point source of electrons with energy 100 keV (the same system is illustrated in Fig. 4.1 in Section 4.2). Four tallies are calculated in the MCNelectron version of this simulation: a pulse-height tally, a plane-crossing tally and two cell-entry tallies. The MCNP6 counterpart of this simulation contains only the pulse-height and

plane-crossing tallies. The simulation setup corresponding to the MCNelectron input file with the name "input_Macrobodies_1MeV_P.txt" consists of eight cells defined as different types of standard macrobodies, one cell defined as the intersection of a general macrobody and a facet of a standard macrobody, one cell defined as the intersection of two standard macrobodies, and an isotropic point source of photons with energy 1 MeV. Since two of the macrobodies (ARB4 and MB) are not available in MCNP, they have been replaced in the MCNP6 counterpart of this simulation by the surfaces that are recognized by MCNP.

All test simulations were done using MCNelectron v1.2.0 or MCNelectron_CUDA v1.2.0, except for the simulation with macrobodies, which was done using MCNelectron v1.2.5 or MCNelectron_CUDA v1.2.5. The simulations with $U_3O_8$ and with macrobodies used 10 concurrent threads on the CPU, and all other mentioned test simulations used 8 CPU threads. In addition, MCNelectron_CUDA used two Nvidia GeForce GTX 780 Ti cards for the two simulations with $U_3O_8$. All simulations were done on a computer with an Intel Core i7-4930K processor running 64-bit Windows 8.1 or Windows 10. In order to avoid system slowdown, it is recommended to set the number of CPU threads to a value that does not exceed the number of logical processors (the latter number can be seen in Windows Tasks Manager, tab "Performance"). In the case of simulations that use CUDA devices, the number of CPU threads specified after the keyword "TASKS" must be reduced by 1 additionally, because, as mentioned in Section 4.1, that number does not take into account one CPU thread that is reserved for exchanging data with CUDA devices (that thread is not used for the actual simulation of particle interactions). ***Note***: If the number of threads does not exceed the number of logical processors on the computer, then the computer time specified in MCNP6 output files is larger than the true simulation time by a factor equal to the square of the number of threads. In the case of 8 threads, the true time of an MCNP6 run is obtained by dividing the specified time by 64. In the case of 10 threads, the true time of an MCNP6 run is obtained by dividing the specified time by 100.

Comparison of MCNelectron and MCNP6 output data shows a very good agreement between the simulation results computed by those two programs. Comparison of the times of CPU-only simulations shows that MCNelectron significantly outperforms MCNP6 in all cases (the MCNelectron execution time is usually less than half of the MCNP6 execution time). In hybrid CPU/GPU computations using two GTX 780 Ti cards, the computation time is additionally reduced approximately by 40 %.

### *7.2. Information about files in subfolder "Test\Verification_for_K_and_L_X-rays" of the MCNelectron distribution package*

Folder "Test\Verification_for_K_and_L_X-rays" of the MCNelectron distribution package contains two subfolders – "MCNelectron" and "MCNP6.1". They contain the input and output files corresponding to the simulations done using MCNelectron v1.2.0 and MCNP6.1, respectively. The aim of those simulations is estimation of characteristic K ($=K_\alpha + K_\beta$), $K_\alpha$ or $L_\alpha$ yield (i.e., the number of characteristic photons per one incident electron and per one steradian) when a thick target consisting of a single chemical element is bombarded by monoenergetic electrons. The target is composed of one of those 15 elements: C, Al, Si, Ti, Fe, Cu, Zn, Ge, Zr, Sn, Sm, Ta, W, Au and Pb. Incident electron energy is from 5 keV to 30 keV.

MCNP6.1 was used in single-event mode. MCNelectron was used without any of the four replacements of ENDF/B-VII.1 data described in the last paragraph of Section 1.1, i.e., using essentially the same data that is used by MCNP6.1 in single-event mode. Consequently, the results obtained with MCNelectron should be statistically identical to the results obtained with MCNP6.1, if both codes apply the same "post-processing" procedures to the original tables of cross sections (i.e., the same interpolation rules, no change of the energy grid, etc.), and use the same cutoff energies. In practice, however, some systematic differences are inevitable because of the "additional" effects, which are not included in specification of the ENDF/B library (some of those effects are described in Section 5), and because of possible differences between the "post-

processing" procedures applied by the two codes (the only type of "post-processing" applied by MCNelectron is interpolation using the rules mandated by the ENDF/B library). The $\chi^2$ test ("chi-squared test") has been performed to determine the magnitude of those differences and their statistical significance. Folder "Test\Verification_for_K_and_L_X-rays" of the MCNelectron distribution package contains Excel file "Comparison_Xyields.xls" with the computed photon counts and the summary statistics of the $\chi^2$ test.

The description of the data contained in subfolder "Test\Verification_for_K_and_L_X-rays" will be divided into five parts:

1) Simulation setup,
2) Directory structure and tally data format,
3) Methodology of the $\chi^2$ test,
4) Results of the $\chi^2$ test,
5) Description of the Excel file with the photon counts and the summary statistics of the $\chi^2$ test.

*7.2.1. Simulation setup*

Two angles of incidence were used: 0° (normal incidence) and 45°. The tallied photons are the ones whose energy corresponds to the $K_\alpha$, K (=$K_\alpha + K_\beta$) or $L_\alpha$ characteristic X-ray line. In the case of normal incidence, the tallied photons are either $K_\alpha$ (for the elements Ti, Cu and Ge) or K (for the elements C, Al and Si). In the case of non-normal incidence, the tallied photons are either $K_\alpha$ (for the elements Fe, Zn and Zr) or $L_\alpha$ (for the elements Sn, Sm, Ta, W, Au and Pb). ***Note***: Subfolder "MCNelectron" also contains the input and output files corresponding to non-normal incidence on targets composed of Ti, Cu and Pt, which were not used for the $\chi^2$ test.

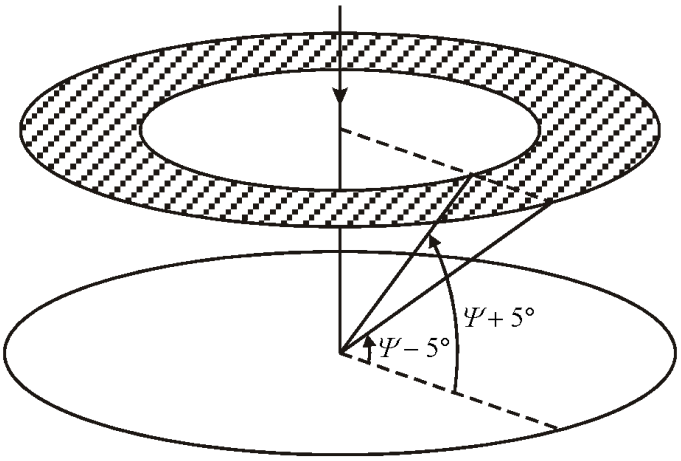

Fig. 7.1. Simulation geometry used to compute the X-ray yields

In all cases, an annular detector was assumed, i.e., only the photons hitting a ring-shaped area centered above the point of incidence were counted (see Fig. 7.1). The distance between the plane of incidence and the detector was 10000 cm, i.e., much larger than the part of the target where the photons originate. This, together with the fact that radiation emitted during atomic relaxation is isotropic, justifies the use of the annular detector even when the angle of incidence is not equal to 0°. The outer and inner radii of the ring correspond to takeoff angles $\Psi - 5°$ and $\Psi + 5°$, respectively, where $\Psi = 40°$ in the case of normal incidence, and $\Psi = 45°$ in the case of non-normal incidence. The simulation was terminated when the relative standard deviation of the photon count corresponding to the mentioned characteristic X-ray line dropped below 1 %, or when the total number of source electrons reached $10^9$, whichever happened first. In the case of the simulations done with MCNelectron, interaction forcing was used (except for carbon target). This caused a significant decrease of the number of incident electrons needed to achieve the required statistical uncertainty (the number of incident electrons in the simulations using MCNelectron was in most cases less than the number of incident electrons in the simulations using MCNP6.1 by an order of magnitude, especially when simulating $L_\alpha$ yields). It should be noted that an even more significant decrease of the simulation time would be achieved using MCNelectron v1.2.2 or later.

Some MCNP6.1 tallies were formed by combining several tallies from independent runs (on different computers) simulating the same system (those simulations differed only by the starting

seed of the stream of random numbers). This approach was applied only to some of the target materials. Thus, all simulations using MCNP6.1 were of two types:

- a "long run" defined in the same way as for MCNelectron, i.e., terminated after $10^9$ incident electrons, or after a decrease of the relative standard deviation of the photon count in the "main" bin below 1 %, whichever happens first,
- a set of five "short runs", terminated after $2 \cdot 10^8$ incident electrons, or after a decrease of the relative standard deviation of the photon count in the "main" bin below $\sqrt{5} \cdot 1\,\% = 2.236\,\%$, whichever happens first.

For some combinations of the target material and incident electron energy, both those approaches were applied. Since the "long run" used the same starting seed as the first of the "short runs", those two runs were not independent. Consequently, in those cases the data of the first "short run" were replaced by the data of the "long run". As a result, up to $1.8 \cdot 10^9$ simulated histories, or combined relative uncertainties of the photon count less than 0.75 % were achieved in some cases.

### *7.2.2. Directory structure and tally data format*

For every simulation, there is the main output file with some overall statistics, and the input file (format of MCNelectron input and output files is described in Sections 4 and 6). Although the formats of the file names of MCNelectron and MCNP6.1 are different, they all have some common components identifying the simulated system: the element symbol, the atomic number, the energy of incident electrons (in MeV), and the character string "45deg" for the simulations using the non-normal incidence (in addition, the MCNelectron file names corresponding to the non-normal incidence contain the letter "K" or "L" indicating the type of the characteristic X-ray emission that was simulated). The names of MCNelectron input and main output files contain the string "ENDF" indicating the source of cross section data used (i.e., the ENDF/B library). The names of MCNP6.1 input and main output files contain the string "MCNP6_SE" (the letters "SE" indicate that the simulation was run in single-event mode). The name of each main output file is obtained by appending the characters "_out" to the name of the corresponding input file.

The most important result of each simulation run is a tally of photons emitted into the above-mentioned range of takeoff angles. For every simulation, there is a separate file with the data of that tally. The files with the tally data for each simulation are in a separate subfolder of the same folder where the corresponding input and main output file are (the name of that subfolder includes the name of the output file). The names of the tally data files are the following:

1) "PTALLY1_E1.txt" for MCNelectron tallies,
2) "Surface3 F1 photons (cos -0.707107 to -0.573576).txt" for MCNP6.1 tallies corresponding to normal incidence,
3) "Surface3 F1 photons (cos -0.766044 to -0.642788).txt" for MCNP6.1 tallies corresponding to non-normal incidence

(the numbers in the latter two file names define the range of the sine of the takeoff angle). Since each of those tallies has three bins, the tally data consist of three lines in the case of MCNelectron, and of four lines in the case of MCNP6.1 (not counting the line with column headers). The energy of characteristic X-ray radiation belongs to the bin corresponding to the next-to-last line of this file (this bin will be called the "main" bin). The previous line and the next line (the last line of the file) correspond to the two bins that are adjacent to the main bin and contain only bremsstrahlung photon counts (they will be called "bremsstrahlung bins" No. 1 and No. 2, respectively). The latter two bins may be used to eliminate the contribution of bremsstrahlung to the photon count in the main bin. This is achieved by subtracting the average photon count in the two bremsstrahlung bins from the photon count in the main bin. Each line of the tally data contains three values:

1) the upper bound of the energy bin (in MeV),
2) the number of photons per one source electron,
3) the relative standard deviation (statistical uncertainty) of the photon count.

All numbers and column headers in the tally data files are tab-delimited.

The input and main output files of all simulations done with MCNelectron are in one folder ("Test\Verification_for_K_and_L_X-rays\MCNelectron"). The input and main output files of all simulations done with MCNP6.1 are placed into different subfolders of the folder Test\Verification_for_K_and_L_X-rays\MCNP6.1", corresponding to different target materials (the names of those subfolders contain the atomic number, the element symbol, and the characters "45deg" for the case of non-normal incidence). In the cases when the MCNP6.1 simulation results were split over five short runs (or one long run and four short runs), those results were placed into subfolders with names "1", "2", "3", "4" and "5", indicating the sequence number of the run.

The MCNP6.1 simulations were automatically interrupted once every minute, the output file was examined and the run was resumed if the required precision was not reached yet (MCNP6 generates a so-called "dump file" with all information needed for resuming the simulation). The approximate duration of each MCNP6.1 run (in minutes) can be determined from the number of the last dump given after the words "starting from dump no." at the beginning of each MCNP6.1 main output file (those times should not be used for any comparisons of computing performance, because the simulations using MCNP6.1 were done on 8 computers with different processors).

### *7.2.3. Methodology of the chi-squared test*

The null hypothesis is that the set of relative differences of the yields predicted by MCNelectron and MCNP6.1 in single-event mode (a total of 82 yields for each method) is equal to the set of the maximum likelihood estimates of the mentioned relative difference for each of the target materials. The maximum likelihood estimate of the relative difference is based on the assumption that the probability distributions of the relative differences corresponding to different electron energies and to the same target material are:

1) independent,
2) normally distributed,
3) with the same mean.

Of these three assumptions, the first two are certainly true if the random number generators implemented in the two compared Monte Carlo codes (MCNP6.1 and MCNelectron) are of sufficiently high quality, because in such a case the photon counts are distributed according to the Poisson distribution, which can be accurately approximated by a normal distribution when the average count is of the order of 10 or greater (this is certainly true in the case discussed). The maximum likelihood estimate of the relative difference, which follows from the mentioned three assumptions, is equal to the weighted average with the weight factors proportional to the inverse variance of each relative difference [18]:

$$d = \sum_{i=1}^{n} \frac{x_i}{s_i^2} \Bigg/ \sum_{i=1}^{n} \frac{1}{s_i^2}, \tag{7.1}$$

where $x_i \equiv (y_{i,2} / y_{i,1}) - 1$ is the value of the relative difference corresponding to the energy value No. $i$ for a given target material ($y_{i,1}$ and $y_{i,2}$ are the characteristic X-ray yields obtained using MCNP6.1 and MCNelectron, respectively), $s_i$ is the standard deviation of the relative difference, and $n$ is the number of electron energy values for a given target material. Thus, the $\chi^2$ test statistic for a given target material is defined as [18]

$$\chi^2 = \sum_{i=1}^{n} \left( \frac{x_i - d}{s_i} \right)^2. \tag{7.2}$$

The value of $s_i$ is calculated as follows:

$$s_i = \frac{y_{i,2}}{y_{i,1}} \sqrt{s_{i,1}^2 + s_{i,2}^2}, \tag{7.3}$$

where $s_{i,1}$ and $s_{i,2}$ are the relative standard deviations of the two characteristic X-ray yields obtained by MCNP6.1 in single-event mode and MCNelectron, respectively (the "relative" standard deviation of the X-ray yield is equal to the absolute standard deviation of the X-ray yield divided by the yield itself). The latter standard deviations can be reliably estimated from Poisson statistics. In

the case of MCNP6.1, the standard deviations of the photon counts in each of the three bins used to calculate the yield (i.e., the bin containing the sum of characteristic X-ray photon count and the bremsstrahlung term, and two adjacent bins containing only bremsstrahlung photons) are equal to the square root of the corresponding photon count. In the case of MCNelectron (with interaction forcing), those standard deviations are equal to the square root of the sum of squared weights of the photons counted in the corresponding bin. The standard deviations of the characteristic X-ray yields are estimated using the standard rules of propagation of independent random errors in arithmetic expressions (this propagation involves adding errors in quadrature [18]).

Let $\chi_0^2$ denote the value of $\chi^2$ that was actually obtained (i.e., the "empirical" value of $\chi^2$). The probability to obtain the value of $\chi^2$ greater than $\chi_0^2$, with the condition that the null hypothesis is true, is equal to [18]

$$P(\chi^2 > \chi_0^2) = \frac{\Gamma\left(m/2, \chi_0^2/2\right)}{\Gamma(m/2)} \equiv \frac{1}{\Gamma(m/2)} \int\limits_{\chi_0^2/2}^{\infty} t^{(m/2)-1} \mathrm{e}^{-t} \mathrm{d}t \,, \tag{7.4}$$

where $m$ is the number of degrees of freedom, $\Gamma\left(m/2, \chi_0^2/2\right)$ is the upper incomplete gamma function, and $\Gamma(m/2)$ is the ordinary gamma function. The outcome of the $\chi^2$ test is based on a comparison of the probability $P(\chi^2 > \chi_0^2)$ defined by (7.4) with a predefined "critical" value (the "significance level"). The typical values of the critical probability are 5 % and 1 %. If $P(\chi^2 > \chi_0^2)$ is less than the critical probability, the difference between the compared datasets is said to be significant at the given level. A decrease of the level at which the difference is significant makes the null hypothesis less likely to be true. Consequently, if $P(\chi^2 > \chi_0^2)$ is less than 5 %, the null hypothesis is said to be "rejected at the 5 % level".

The expression of the half-width of the 95 % confidence interval of the weighted average $d$ follows from its definition (7.1) and from the assumption that $x_i$ ($i$ = 1, 2, …, $n$) are independent and normally distributed variables:

$$\Delta = 1.96 \left( \sum_{i=1}^{n} \frac{1}{s_i^2} \right)^{-1/2} . \tag{7.5}$$

*7.2.4. Results of the chi-squared test*

The $\chi^2$ test described above has been performed with 16 samples: the overall sample of 82 values and 15 smaller samples corresponding to each of the 15 chemical elements separately. The results of this test are presented in Table 2. Column 3 of Table 2 contains the maximum likelihood estimate of the average relative difference for each sample ($d$), column 4 contains the half-width of the 95 % confidence interval ($\Delta$) of $d$, column 5 contains the observed value of the reduced chi-squared (i.e., $\chi_0^2/m$, where $m$ is the number of degrees of freedom), and column 6 contains the probability that $\chi^2 > \chi_0^2$ under the condition that the null hypothesis is true, i.e., that the mentioned maximum likelihood estimate is the true value of the relative difference for each electron energy. Since estimation of the weighted average consumes one degree of freedom, the number of degrees of freedom for each target material is one less than the sample size $n$. Since this estimation has been done for each of 15 target materials separately, the number of degrees of freedom for the overall sample of 82 values is 82 – 15 = 67.

As evident from Table 2, the results of the $\chi^2$ test indicate that the observed and expected statistical distributions of the relative difference between the two sets of characteristic X-ray yields are consistent with each other at the 5 % level for all samples. If the same $\chi^2$ test is performed using the null hypothesis that there is *no* difference between the two sets of X-ray yields, then the null hypothesis is rejected at the 1 % level for 5 out of 15 target materials, as well as for the overall sample of 82 observations. It follows that there is probably some systematic (i.e., non-random)

difference between the X-ray yields calculated by MCNP6.1 and MCNelectron. This is also in accord with the fact that the 95 % confidence interval of $d$ does not include the value 0 for 7 out of 15 target materials (i.e., $|d| > \Delta$ in 7 out of 15 cases). However, the values of $d$ and $\Delta$ presented in columns 3 and 4 of Table 2 indicate that this difference is relatively small (of the order of a few percent or less).

**Table 2**. Results of the $\chi^2$ test on the set of relative differences between characteristic X-ray yields predicted by MCNelectron and by MCNP6.1 in single-event mode

| Element | Sample size ($n$) | Average relative difference ($d$) | Half-width of 95 % conf. interval ($\Delta$) | Reduced chi-squared | $P$ |
|---|---|---|---|---|---|
| C | 6 | 2.40% | 1.15% | 1.934 | 8.5% |
| Al | 7 | 2.05% | 1.07% | 0.916 | 48.2% |
| Si | 7 | 0.93% | 1.07% | 1.958 | 6.8% |
| Ti | 5 | -1.17% | 1.23% | 1.521 | 19.3% |
| Fe | 6 | -2.56% | 1.29% | 2.142 | 5.7% |
| Cu | 4 | -2.45% | 1.38% | 0.348 | 79.1% |
| Zn | 6 | -2.47% | 1.29% | 1.020 | 40.4% |
| Ge | 4 | -1.46% | 1.36% | 1.663 | 17.3% |
| Zr | 6 | -1.30% | 2.19% | 0.326 | 89.7% |
| Sn | 5 | -0.90% | 1.33% | 1.954 | 9.9% |
| Sm | 4 | -0.53% | 1.70% | 2.296 | 7.6% |
| Ta | 5 | -1.81% | 1.52% | 0.823 | 51.0% |
| W | 6 | -0.54% | 1.57% | 0.747 | 58.8% |
| Au | 5 | -0.52% | 1.91% | 0.645 | 63.0% |
| Pb | 6 | -1.48% | 1.91% | 0.934 | 45.7% |
| Total | 82 | -0.50% | 0.35% | 1.275 | 6.4% |

*7.2.5. Description of the Excel file with the photon counts and the chi-squared test statistics*

Excel file "Comparison_Xyields.xls" in folder "Test\Verification_for_K_and_L_X-rays" contains the following types of data:

1) The tally data (single-run or combined from five independent runs). Those data are in columns with headers "*N*_source_electrons" (meaning the number of incident electrons) "Brems1", "Main", "Brems2" (meaning the photon counts per one incident electron in bremsstrahlung bin No. 1, main bin and bremsstrahlung bin No. 2, respectively), and "Brems1_relErr1", "Main_relErr", Brems2_relErr" (meaning the corresponding relative standard deviations).
2) Ratio of bremsstrahlung and total photon counts in the main bin (column "Brems/Tot").
3) Characteristic X-ray count after eliminating the bremsstrahlung term, and its relative standard deviation (columns "Char. X-ray count" and "Char_X_relErr", respectively).

The above statistics are given both for the simulations performed using MCNP6.1 in single-event mode, and for the simulations performed using MCNelectron. The statistics described below pertain to differences between the characteristic X-ray counts obtained by those two methods (the run-by-run statistic are shown in green, and the summary statistics are shown in red):

4) Relative difference of characteristic X-ray counts, and its standard deviation (columns "RelDiff" and "RelDiff_SDev", respectively),
5) Auxiliary run-by-run statistics for calculating the summary statistics: ratio of the deviation of the mentioned relative difference from the weighted average (7.1) to the corresponding standard deviation (column "RelDiff_SD_ratio"), and inverse squared standard deviation (column "1/SDev^2").

The summary statistics (shown in red):

6) Average relative difference for each target material and for the overall sample of 82 observations (column "Average_RelDiff"),
7) Half-width of the 95 % confidence interval (column "95 % conf. interval"),
8) Reduced chi-squared (column "Reduced Chi-sq"),
9) The probability to obtain the value of $\chi^2$ greater than the observed value $\chi_0^2$ (column "P").

For convenience, the summary statistics are shown in two identical tables: one on the left-hand side of the worksheet, and another on the right-hand side. Those are the same statistics as in columns 3–6 of Table 2, but in a slightly different order (Table 2 lists all target materials in the order of their atomic numbers, whereas the Excel file lists all statistics corresponding to normal incidence before the statistics corresponding to non-normal incidence).

By default, the null hypothesis corresponding to the values of $\chi^2$ shown in Excel file "Comparison_Xyields.xls" is that the set of relative differences of the yields predicted by MCNelectron and MCNP6.1 in single-event mode is equal to the set of the maximum likelihood estimates of the mentioned relative difference for each of the target materials. It is also possible to calculate the reduced $\chi^2$ corresponding to the null hypothesis that the mentioned relative differences are equal to zero. This is achieved by entering "0" into the cell "T2". This causes a change of the numbers in columns "RelDiff_SD_ratio", "Reduced Chi-sq" and "P". ***Note***: Only 0 and 1 are allowed in cell T2.

## 8. Information about files in subfolder "Simulations\X-rays" of the distribution package

The subfolder "Simulations\X-rays" of the MCNelectron distribution package contains 99 MCNelectron_CUDA input files and the corresponding output files (the output file name is obtained by appending "_out" to the input file name). Each of them is used to simulate characteristic X-ray emission from a thick target composed of one of those elements: C, Al, Si, Ti, Fe, Cu, Zn, Ge, Zr, Sn, Sm, Ta, W, Pt, Au and Pb, bombarded by monoenergetic electrons with energy from 5 keV to 30 keV (the element symbol and electron energy are included in the names of input and output files). Those simulations were performed using ENDF/B-VII.1 data, with the exception of inner-shell electron impact ionization cross sections, which were calculated using the distorted-wave Born approximation (this is achieved by specifying the option "ION_DWBA 1" in the MCNelectron input file). Interaction forcing was used (see Section 4.8). Those simulations were performed using MCNelectron v1.2.0 in "hybrid" CPU/GPU mode on a computer with an Intel Core i7-4930K processor and two Nvidia GeForce GTX 780 Ti video cards. The operating system was 64-bit Windows 8.1.

Two angles of incidence were used: 0° (normal incidence) and 45°. In the latter case, the file names contain the character string "45deg". The tallied photons are the ones whose energy corresponds to the $K_\alpha$, K ($=K_\alpha + K_\beta$) or $L_\alpha$ characteristic X-ray line (the energy range containing that energy value is specified using the keyword "E1" in the input file). In the case of normal incidence, the tallied photons are either $K_\alpha$ (for the elements Ti, Cu and Ge) or K (for the elements C, Al and Si). In the case of non-normal incidence, the tallied photons are either $K_\alpha$ (for the elements Ti, Fe, Cu, Zn and Zr) or $L_\alpha$ (for the elements Sn, Sm, Ta, W, Pt, Au and Pb). In the latter case, the characteristic X-ray line is specified by the letter "K" or "L" inserted after "45deg_".

In all cases, an annular detector was assumed, i.e., only the photons hitting a ring-shaped area centered above the point of incidence were counted (see Fig. 7.1 in Section 7.2). The distance between the plane of incidence and the detector was 10000 cm, i.e., much larger than the part of the target where the photons originate. This, together with the fact that radiation emitted during atomic relaxation is isotropic, justifies the use of the annular detector even when the angle of incidence is not equal to 0°. The outer and inner radii of the ring correspond to takeoff angles $\Psi - 5°$ and $\Psi + 5°$, respectively, where $\Psi = 40°$ in the case of normal incidence, and $\Psi = 45°$ in the case of non-normal incidence. Each simulation was terminated when the relative standard deviation of the

photon count corresponding to the mentioned characteristic X-ray line dropped below 1 %, or when the total number of source electrons reached $10^9$, whichever happened first.

The computed tally data are in subfolders of the folder “X-rays”. The names of those subfolders include the names of the corresponding output files. In order to obtain the characteristic X-ray yield (in units of photons / (sr · electron)), the number of tallied photons must be divided by the solid angle subtended by the annular detector. The solid angle corresponding to the range of takeoff angles from 35° to 45° is 0.839 sr, and the solid angle corresponding to the range of takeoff angles from 40° to 50° is 0.774 sr. ***Notes***: **1)** The input files contain the option “tallies_per_source 1”, which ensures that the photon numbers in the computed tally data are already given per source electron. **2)** In the case of normal incidence, two photon tallies were calculated: the one mentioned above and another one corresponding to smaller takeoff angles. The name of the photon tally data file corresponding to takeoff angles from 35° to 45° is “PTALLY1_E1.txt”.

Each of the mentioned tallies contains three bins. The energy of characteristic X-ray radiation belongs to bin No. 2. Bins No. 1 and No. 3 contain only bremsstrahlung photon counts. Consequently, they may be used to eliminate the contribution of bremsstrahlung to the photon count in bin No. 2. This is achieved by subtracting the average photon count in bins No. 1 and No. 3 from the photon count in bin No. 2.

## Appendix A. The MCNEcode programming language

The user's code must be in "code blocks", which are embedded inside the MCNelectron input file. There are no restrictions on the number of the code blocks and their placement in the file. Each code block starts with the header, which must have one of the following four formats:

PROGRAM <ProgramName> – for defining a program,
SUBROUTINE <ProgramName> – for defining a subroutine,
PARAMETERS – for defining global parameters (must be the first executable program in the file),
ARRAYS – for defining and initializing the arrays (must precede all executable programs).

The end of a code block is indicated by the corresponding "END" statement: "END PROGRAM", "END SUBROUTINE", "END PARAMETERS", or "END ARRAYS". The "END" statement must be on a separate line. The first executable statement of a code block may be on the same line as the block header. The names of programs and subroutines cannot contain spaces and must conform to all other requirements for identifiers (i.e., no special symbols, uniqueness, etc).

There are only two differences between a "program" and a "subroutine":

1) The code defined using the "PROGRAM" header is executed immediately when encountered in the MCNelectron input file, whereas the code defined using the "SUBROUTINE" header can be executed only when called (invoked) programmatically from inside another program or subroutine.
2) Subroutines can have up to 20 formal parameters, which are initialized at runtime as arguments listed in the call to the subroutine. Programs have no formal parameters.

However, each code block defined using the header "PROGRAM" can also be called from another program or subroutine. In this respect, "programs" may be considered subroutines with no formal parameters.

The main difference between the code block "PARAMETERS" and a "regular" program is that all variables appearing on the left of the assignment operator in the "PARAMETERS" block are treated as "global" parameters, i.e., they are visible in all other programs defined in the same file. If a variable was not defined as a global parameter, then it will be visible only in the program where it was assigned a value. In addition, unlike "regular" programs, the "PARAMETERS" code has no name and cannot therefore be called from other programs or subroutines. The "PARAMETERS" block must precede all other code blocks of the same file, excluding the "ARRAYS" block.

Each program or subroutine can call only the subroutines and programs whose definitions precede the definition of the current program or subroutine. Further on, all executable code blocks (i.e., the code blocks with the headers "PROGRAM", "SUBROUTINE" and "PARAMETERS") will be collectively called "programs", except when discussing the formal parameters of subroutines in Section A.3.

The built-in compiler of MCNElectron automatically compiles each user program to a so-called "bytecode". In this process, each binary operation (i.e., addition, multiplication, raising to a power, comparison, logical operations, etc.) is replaced by the pointer to the corresponding static function (written in C++), which performs the corresponding operation. Each call to a built-in function (such as square root, logarithm, trigonometric functions, address of a variable, generation of a random number, sorting of an array, displaying a line of text, etc.) is similarly replaced by the pointer to the corresponding static function (written in C++ and in most cases involving a call to a function from the C runtime library or an open-source function whose code was downloaded from a repository of software, such as www.netlib.org). After that, the instructions of the bytecode are ready for execution. The built-in bytecode interpreter of MCNelectron executes the bytecode directly (one instruction at a time).

The descriptions of the built-in functions used for data output and input, as well as a short overview of the similarities and differences between MCNEcode and C were presented in Section 4.12. The detailed description of all other features of MCNEcode is presented below.

### *A.1. MCNEcode syntax and built-in functions*

#### *A.1.1. The binary operators and other basic elements of the syntax*

Each non-empty line of a program must contain an expression (with the assignment operator or without it). The exceptions to that rule are the lines containing only the curly brace (it may be the only character in a line), or only the keyword "else". Each expression may contain any number of arithmetic, logical or comparison operations and calls to various functions. The value of the last calculated expression is the final value of the computed program (in other words, it is the value "returned" by the program). Below is the list of the binary operators and special symbols that may be used in a program:

Arithmetic operators:

+ (addition),
- (subtraction),
* (multiplication),
/ (division),
^ (raising to a power),
= (assignment of a value).

For example, the statement "a = a + 1" increases the value of the variable "a" by 1.

Comparison operators:

< ("less than"),
<= ("less than or equal to"),
> ("greater than"),
>= ("greater than or equal to"),
== ("equal to"),
!= ("not equal to"),
<> ("not equal to").

If a given inequality or equality is true, then the result of the comparison statement is 1, and if it is false, then the result is 0. Although the comparison operators are most frequently used in the conditional construct "if…else" and in the loop operators "for" and "while", they may also be used in arithmetic expressions. For example, if a = 3 and b = 2, then the expression "(a > b) + 1" is equivalent to the expression "1 + 1", because the inequality "a > b" is correct. The expression "a > b + 1" is equivalent to the expression "3 > 3", because, when parentheses are not used, the arithmetic operations are computed before comparison operations (see below about operator priority). Since the inequality "3 > 3" is incorrect, the result of the latter statement is 0.

Logical operators:

& (logical operation "AND", also called "conjunction"),
| (logical operation "OR", also called "disjunction").

If both operands are non-zero (e.g., "(a > b) & (b > 1)" or "a/2 & –b", when a = 3 and b = 2), then the result of the logical operation "AND" is 1. If at least one of the operands is zero, then the result of the logical operation "AND" is 0. The result of the logical operation "OR" is 1 when at least one operand is non-zero. For example, if a = 3 and b = 2, the value of the expression "(b/2 > a) | (2 < a)" is 1, because the second inequality is correct. If both operands are zero, then the result of the logical operation "OR" is 0.

***Note***: There is no negation operator. Instead, there is the built-in function "not(x)", which returns 1 when $x = 0$, and 0 when $x \neq 0$.

Separators:

Parentheses “(” and “)”, which are used for grouping operands of binary operations, as well as for indicating the start and the end of a function argument list or of the header of the “if…else” construct or the loop operator “for” or “while” (see Section A.1.2 for the description of the “if…else”, “for” and “while” operators);

curly braces “{” and “}”, which are used to denote the start and the end of a “branch” of the “if…else” construct or of the “body” of the “for” or “while” loop operator, i.e., the sequence of statements that should be executed when a specified condition is true (the curly braces are needed only when that sequence consists of two or more statements);

brackets “[” and “]”, which are used for grouping operands of binary operations, as well as for indicating the start and the end of a list of array indices;

the comma “,”, which is used to separate function arguments in function calls, or array indices in references to arrays;

the semicolon “;”, which is required to separate statements that are on the same line. The semicolon is not required at the end of a line with code (except when the line ends with a “for” loop without a “body”). However, if the semicolon is written at the end of a line where it is not required, the compiler will not treat this as an error.

Identifiers of variables, arrays and functions are case-insensitive. Identifiers cannot be longer than 50 characters (otherwise the trailing part of the identifier, starting with character No. 51, will be ignored).

The operator priority, when it is not indicated by parentheses, is determined according to the usual rules. The assignment operator has the lowest priority, i.e., the assignment operation is done last. The priority of logical “AND” is higher than the priority of logical “OR” (for example, the expression “a & b | c & d” is equivalent to “(a & b) | (c & d)”). The comparison operation priority is higher than the logical operation priority, but lower than the arithmetic operation priority (excluding the assignment operation). Multiplication and division priority is higher than addition and subtraction priority, and the priority of raising to a power is higher than multiplication and division priority. Addition has the same priority as subtraction. This means that a sequence of addition and subtraction operations will be evaluated left to right. Similarly, multiplication has the same priority as division. The priority of function calls is higher than the arithmetic operation priority. For example, the expression “2 * a + 3 * b*sin(x)^2” would be evaluated in this order: first, the built-in function sin($x$) is calculated, then this result is raised to the power 2 and multiplied by 3$b$, then the value of 2$a$ is added. Thus, using parentheses, the same expression could be written as follows: “(2 * a) + (3 * b *((sin(x))^2))”.

Comment lines may be inserted into programs. Each comment line begins with “//”. If this symbol is encountered in a line of the user’s code, the compiler will ignore the remainder of the line. In order to insert the comment symbol into an output line of text (using the built-in functions “write”, “print”, “dir” and the related functions), it cannot be simply specified between the double quotes as one of the arguments of the function, because then it would be interpreted as a comment of the user’s code rather than as the text that should be included in the output line. In order to insert the comment symbol into the output line, the two slashes should be assigned to different strings like this: Write("/","/comment"), because the mentioned functions concatenate consecutive strings.

If a syntax error is found during compilation of a program, then the corresponding error message will be inserted in the MCNelectron output file after the line of code where the error was found. Only the first detected error is pointed out. If there is no error message, this means that the syntax of the program is correct.

*A.1.2. The control flow statements*

According to Wikipedia, "a control flow statement is a statement whose execution results in a choice being made as to which of two or more paths should be followed". There are three types of control flow statements in MCNEcode: the conditional construct "if…else" and the loop operators "while" and "for". They are described below.

The conditional construct "if…else"

In the simplest case, the format of the "if…else" statement is the following:

```
if (expression_1) assignment_statement_1
else if (expression_2) assignment_statement_2
...
else assignment_statement_n
```

For example:

```
if (t < 3) y = 0.5
else if (t < 6) y = 2
else y = 3
```

The program evaluates each of the expressions in parentheses (those expressions are also called "conditions") until an expression with a non-zero value is encountered. Then the statement (or several statements) corresponding to that condition are processed and control returns to the point after the if…else statement (i.e., the subsequent conditions are not evaluated). The "else if" and "else" branches are optional; only the initial "if" branch is required. If neither condition evaluates to a non-zero value, then the "else" branch is executed (if it is present). The keywords "if" and "else" and assignment statements may be written on different lines, e.g.

```
if (t <= 5)
    y = 0.5
else
    y = 3
```

"if", "else if" or "else" branch may consist of several statements. Then, the start and end of the statements belonging to that branch must be indicated by curly braces, e.g.,

```
a = 5
if (t <= a) {
    b = exp(0.5*(t - a)) - 1
    y = 2*b + 2
}
else  { b = t - a + 1
    y = 2+ln(b) }
```

The loop operator "while"

In the simplest case, the format of the "while" loop operator is the following:

```
while (expression) assignment_statement
```

For example:

```
while (t < 3) t = t ^ 1.1
```

The program evaluates the assignment statement repeatedly while the expression in parentheses is non-zero (that expression is the so-called "condition" of the loop operator). If the assignment statement must be evaluated a predetermined number of times, then a temporary variable must be used, which is incremented by one after each evaluation. In such a case, the "body" of the "while" operator consists of more than one statement and curly braces must be used (they are used exactly as in the "if…else" construct). The following code calculates the factorial of the number 10:

```
fact = 1; i = 2
while (i <= 10) {
    fact = fact * i
    i = i + 1 }
```

***Note***: In the above example, the last calculated expression is the comparison operation "i <= 10". Its last value is zero. Therefore, if those lines of code were the last lines of the program, it would return zero. In order to ensure that the program returns the value of the factorial, an additional line is needed (for example, it could be "a = fact", or just "fact").

The loop operator "for"

The "for" operator differs from the "while" operator by inclusion of two additional statements in the header of the loop: the initialization statement and the update statement. The initialization statement is inserted before the condition of the loop, and the update statement is inserted after the condition. All three statements must be separated by semicolons. The initialization statement is executed only once, just before testing the condition for the first time. The update statement is executed after each iteration. If the body of the "for" loop is never executed, then the update statement will never be executed, too. Below is the code that is equivalent to the previous example (the factorial of number 10), but it uses the "for" loop instead of the "while" loop:

```
fact = 1
for (i = 2; i <= 10; i = i + 1) fact = fact * i
```

Every "for" loop can be replaced by an equivalent "while" loop. However, as illustrated by the last example, the code that uses "for" loops is usually more compact (and hence more readable) than the code that uses only "while" loops.

### *A.1.3. The system variables*

MCNEcode has 13 "system variables", whose identifiers are given below:

@0, @1, …, @9 – ten variables whose values can be modified only on the command line used to start MCNelectron. If any of those variables is not specified on the command line, then it will be assigned the value 0.

IER, AbsErr and nEval are modified in each call to any built-in integration function. Their meanings are: the error code, the estimate of the absolute error and the number of integrand evaluations, respectively (for more information about the built-in integration functions, see Section A.4). Those three variables are set to zero before starting execution of each user program.

The identifiers of the system variables may not be used on the left-hand side of the assignment operator or as the argument of the built-in function "loc".

### *A.1.4. The built-in functions*

There are 70 built-in functions in MCNEcode. This number excludes the variants of the same function, differing from the original function only by the added ability to modify some parameters of the calculation process (the identifier of such a variant is obtained by appending "2" to the identifier of the original function). The descriptions of the built-in functions are given below.

Basic built-in functions:

| | |
|---|---|
| not(x) | (returns 1 when $x = 0$, and returns 0 when $x \neq 0$), |
| exp(x) | (the exponential function), |
| ln(x) | (the natural logarithm), |
| lg(x) | (the decimal logarithm), |
| sqrt(x) | (the square root), |
| sin(x), cos(x), tg(x) | (the trigonometric functions), |
| arcsin(x), arccos(x), arctg(x), arctg2(y,x) | (the inverse trigonometric functions), |
| sinh(x), cosh(x), tanh(x) | (the hyperbolic functions), |
| j0(x), j1(x), jn(n,x) | (Bessel functions of the first kind: orders 0, 1, *n*, respectively), |
| y0(x), y1(x), yn(n,x) | (Bessel functions of the second kind: orders 0, 1, *n*, respectively), |
| erf(x) | (the error function), |

| | |
|---|---|
| erfc(x) | (the complementary error function), |
| gamma(x) | (the gamma function), |
| lngamma(x) | (the natural logarithm of the absolute value of the gamma function), |
| gammp(a,x) | (the incomplete gamma function), |
| abs(x) | (the absolute value of a number), |
| max(x,y), min(x,y) | (the larger or smaller of two values), |
| select(x,y,z) | (if $x \neq 0$, then returns $y$, otherwise returns $z$), |
| ldexp(x,y) | ($x \cdot 2^y$), |
| fmod(x,y) | (the floating-point remainder of $x / y$), |
| ceil(x) | (the smallest integer that is greater than or equal to $x$), |
| floor(x) | (the largest integer that is less than or equal to $x$), |
| near(x) | (the integer that is closest to $x$), |
| time() | (the number of seconds elapsed since midnight, January 1, 1970), |
| clock() | (the number of milliseconds elapsed since the start of execution of MCNelectron), |
| rand() | (a pseudorandom integer in the range 0 to 32767), |
| srand(x) | (sets the starting point for generating a series of pseudorandom integers: the next call to rand() will start a new sequence of pseudorandom numbers, which depends only on $x$; the function srand(x) always returns zero), |
| rand2(type) | (if "type" is 0, then a uniform random number in the range [0, 1]; if "type" is 1, then a standard normal deviate), |
| srand2(x) | (sets the starting point for rand2), |
| Return(x) | (immediate termination of the current program or subroutine, returning the value of "x"). |

***Note***: Although the function "Return(x)" does return the value of "x", this value cannot be used in the program that called the function "Return(x)", because any statement containing a call to "Return" will be unfinished: it will be interrupted by that call. For example, the statement "a = exp(t) + Return(10) + sin(x)" would be executed by first calculating the value of exp(t) and then terminating the current program with the return value of 10. Thus, the term "sin(x)" would not be calculated, and the value of the variable "a" would not be modified.

Advanced built-in functions:

loc(a) – memory address or sequence number of the programming object "a", where "a" may be the identifier of a variable (parameter), array, subroutine or another built-in function. If the argument is the identifier of a variable or an array, then the returned value is its memory address, and if it is a function name, then the returned value is the sequence number of that function in the set of all defined functions.

Size(loc(a)) – number of elements of the one-dimensional array "a";
Size2(loc(a), i) – length of dimension No. i of the array "a".

memcpy(destAddr, srcAddr, nBytes) – copies nBytes bytes from the address specified by srcAddr to the address specified by destAddr.

indirect(addr) – value of an 8-byte floating-point number in the memory location specified by the address "addr".

dir("string1",a,"string2",b,…) – creates an input directive, which may be optionally written to the MCNelectron log file (see also Section 4.12.1),

dir2("string1",a,"string2",b,…) – appends formatted text to the current input directive and does not append the newline character, so that additional text can be appended to the current input directive by subsequent calls to "dir" or "dir2" (see also Section 4.12.1),

write("string1",a,"string2",b,…) – writes formatted text to the file opened with "fopenw" (if there is such a file) and appends the newline character. If there is no file opened with "fopenw", then "write" is equivalent to "dir" (see also Section 4.12.2),

write2("string1",a,"string2",b,…) – same as "write", but does not append the newline character. If there is no file opened with "fopenw", then "write2" is equivalent to "dir2" (see also Section 4.12.2),

print("string1",a,"string2",b,…) – displays formatted text on the screen and appends the newline character (see also Section 4.12.1),

print2("string1",a,"string2",b,…) – same as "print", but does not append the newline character,

system("string1",a,"string2",b,…) – passes the formatted line of text for execution to the operating system (see also Section 4.12.2),

fopenw("string1",a,"string2",b,…) – formats the file name and opens the specified file for writing (see also Section 4.12.2),

fopenr("string1",a,"string2",b,…) – formats the file name and opens the specified file for reading (see also Section 4.12.2),

SetNumberFormat("string") – sets the default number format for output (see also Section 4.12.1).

ReadArray(loc(a)) – reads the data of the one-dimensional array "a" from the file opened with "fopenr" (see also Section 4.12.2),

ReadArray2(loc(a), n) – reads up to n initial elements of the one-dimensional array "a" from the file opened with "fopenr" (see also Section 4.12.2),

SkipLines(n) – skips n lines in the file opened with "fopenr" (see also Section 4.12.2).

fclosew() – closes the file opened with "fopenw",

fcloser() – closes the file opened with "fopenr".

Invert(loc(a)) – inversion of the square two-dimensional array (matrix) "a". This function replaces the original matrix with the inverse matrix and returns the determinant of the original matrix. The array data type must be 8-byte floating point ("double").

Find(x,loc(a),i0) – the sequence number of the largest element of a one-dimensional array "a", whose value is less than or equal to the value of "x". It is assumed that elements of the array form a non-decreasing sequence. "i0" is the sequence number of the starting element, i.e., the location in the array where the search starts (if i0 < 1 or if i0 exceeds the number of elements in the array, then the search starts from the middle element of the array). The search algorithm depends on the size of array "a": if the number of array elements is less than 40, then array elements are checked sequentially starting from the element No. i0, otherwise the method of repeated bisection of the index range is applied. If *x* is less than the first element of the array, then the function "Find" returns zero. The array data type must be 8-byte or 4-byte floating-point ("double" or "float").

Find2(x,loc(a),i0,n,m) – the extended version of the function "Find". The meaning of the first three arguments was explained in the previous paragraph. "n" is the number of the initial elements of the array "a" that are to be included in the search (i.e., the effect is the same as though the array "a" was replaced by a smaller array, which consists of the initial *n* elements of the array "a"). If $n < 1$ or if *n* exceeds the number of elements of the array "a", then all elements of that array are included in the search. The argument "m" determines the search algorithm: if $m = 1$, then a simple sequential search is performed, and if $m = 2$, then the method of repeated bisection of the index range is applied.

Hist(loc(a),loc(b),loc(c)) calculates frequencies of values of the elements of the array "a" in intervals ("bins") defined by the array "b" (the "frequency" in this context means the number of times a value belonged to a given bin). The calculated frequencies are assigned to elements of the array "c". The values of the array "b" must be sorted in ascending order (equal values are not allowed). A value is placed into a bin if it is less than or equal to the upper edge of that bin and greater than its lower edge. All values that are less than or equal to the first element of the array "b" are placed into the first bin, and the values that are greater than the last element of the array "b" are not counted. The function "Hist" returns the number of elements of the array "a" whose values do not exceed the value of the last element of the array "b". The data type of the arrays "a" and "b" must be "double" or "float" (i.e., 8-byte or 4-byte floating-point). The array "c" must be of the data type "double" or "int" (8-byte floating-point or signed 4-byte integer).

Hist2(loc(a),loc(b),loc(c),n,i) – the extended version of the function "Hist". The meaning of the first three arguments was explained in the previous paragraph. The argument "n" is the maximum number of initial elements of the array "a" that must be processed, and the argument "i" indicates if elements of the array "c" should be set to zero prior to processing. If "i" is equal to zero, then elements of the array "c" are set to zero before calculating the frequencies, otherwise they are not modified initially, and they are incremented during processing. If "i" is zero and "n" is less than 1 or greater than or equal to the size of the array "a", then behavior of "Hist2" is identical to "Hist".

Sort(loc(a)) – sorting of the one-dimensional array "a". The array may be of any data type;

Sort2(loc(a), loc(p)) – sorting of the one-dimensional array "a" with permutations returned in the array "p". The array "a" may be of any data type. The array "p" must of the data type "int" (signed 4-byte integer). The array "p" contains the original position (i.e., index value) of each element of the sorted array.

Int(f,x,a,b) – integral of expression "f" with respect to *x* from *a* to *b* (e.g., "Int(1/sqrt(exp(t*x)+x),x,0,5)").

Inti(f,x,a,1) – integral of expression "f" with respect to *x* from *a* to +∞;
Inti(f,x,a,-1) – integral of expression "f" with respect to *x* from –∞ to *a*;
Inti(f,x,a,2) – integral of expression "f" with respect to *x* from –∞ to +∞ (the argument "a" is not used).

Intw(f,x,a,b,w,1) – integral of expression $f(x)\cos(wx)$ with respect to *x* from *a* to *b*;
Intw(f,x,a,b,w,2) – integral of expression $f(x)\sin(wx)$ with respect to *x* from *a* to *b*.

Sum(f,i,i1,i2) – sum of terms "f" when the summation index *i* varies from i1 to i2 (e.g., "Sum(ln(i),i,2,10)");
Sum2(f,t,a,b,dt) – sum of terms "f" when the summation variable *t* varies from *a* to *b* in increments of dt.

Iter(f,i,i1,i2) – repetition ("iteration") of expression "f" when the iteration index *i* varies from i1 to i2.

Root(f,x,a,b) – root of the nonlinear equation $f(x) = 0$. *a* and *b* are the limits of the interval that should be searched for the root. For example, the expression "Root(sqrt(x)-1+x^3,x,0,1)" is equal to the root of the equation $\sqrt{x} - 1 + x^3 = 0$, i.e., 0.60542342357183.

For more information about the built-in functions "Int", "Inti", "Intw", "Sum", "Iter" and "Root", see Section A.4.

***Notes***: 1. If the value of the array element whose number is returned by the function "Find(x,loc(a),i0)" or "Find2(x,loc(a),i0,n,m)" occurs more than once in the array "a", then the returned result may correspond to any of the occurrences, depending on the values of the arguments "i0", "n" and "m".

2. If elements of the array "a" do not form a non-decreasing sequence, then the functions "Find(x,loc(a),i0)" and "Find2(x,loc(a),i0,n,m)" usually return an incorrect result.

Some of the mentioned mathematical functions are special functions, which are defined as integrals. Below are the definitions of those functions:

Bessel function of the first kind, order *n*:

$$J_n(x) = \frac{1}{\pi}\int_0^{\pi} \cos(n\theta - x\sin\theta)\mathrm{d}\theta \qquad (n = 0, 1, 2, ...) \tag{A.1}$$

Bessel function of the second kind, order *n*:

$$Y_n(x) = \frac{1}{\pi}\int_0^{\pi} \sin(x\sin\theta - n\theta)\mathrm{d}\theta - \frac{1}{\pi}\int_0^{\infty} [\mathrm{e}^{nt} + (-1)^n \mathrm{e}^{-nt}]\mathrm{e}^{-x\sinh t}\mathrm{d}t \qquad (n = 0, 1, 2, ...) \tag{A.2}$$

Error function:

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}}\int_0^{x} \mathrm{e}^{-t^2}\mathrm{d}t \tag{A.3}$$

Complementary error function:

$$\mathrm{erfc}(x) = 1 - \mathrm{erf}(x) = 1 - \frac{2}{\sqrt{\pi}}\int_0^{x} \mathrm{e}^{-t^2}\mathrm{d}t \tag{A.4}$$

Gamma function:

$$\Gamma(x) = \int_0^{\infty} t^{x-1}\mathrm{e}^{-t}\mathrm{d}t \tag{A.5}$$

Incomplete gamma function:

$$\gamma(a,x) = \frac{1}{\Gamma(a)}\int_0^{x} t^{a-1}\mathrm{e}^{-t}\mathrm{d}t \tag{A.6}$$

### *A.2. Using data arrays*

A data array is a collection of numbers ("elements"), each selected by one or more indices that can be computed at runtime by the user program. Arrays in MCNEcode can have up to 20 dimensions (indices). A two-dimensional array, i.e., an array that has two indices, can be visualized as a table of numbers, where the row number is equal to the value of the first index, and the column

number is equal to the value of the second index (a two-dimensional array is often called "a matrix"). Array element values can be accessed and modified in user-defined programs. Array elements are referenced using the format "a[i1, i2, i3, …]", where "a" is the array name, and "i1, i2, i3" are the array indices (positive integer numbers). For example, if "A" is a two-dimensional array, then the following statement sets the value of the element at row 2 and column 3 equal to the sum of two elements: element at row 5 and column 10, and element at row 20 and column 30:

```
A[2,3] = A[5,10] + A[20,30]
```

Any expression can be used instead of each array index. The value of each such expression is rounded to the nearest integer number. For example, if "N" is a variable that was previously assigned the value 10.6, then the following example is equivalent to the previous one:

```
A[2,N-8] = A[N/2,10] + A[9+N,3*N-2]
```

8 array data types are supported (8- or 4-byte floating-point, and 4-, 2- or 1-byte signed or unsigned integer). However, in all arithmetic operations, the array element values are converted "on the fly" to 8-byte floating-point format.

In the case of a multi-dimensional array, the array layout in computer memory is such that the adjacent elements of the array differ by the value of the last index (as in C).

Arrays in MCNEcode are global objects, which are visible in all programs defined in the same file. A new array can be created only in the code block "ARRAYS", which must precede all other code blocks defined in the same file. Arrays are defined by specifying their data type, name and lengths of the dimensions. The names and meanings of the array data types are listed below:

double – 8-byte floating point,
float – 4-byte floating point,
int – 4-byte signed integer,
uint – 4-byte unsigned integer,
int2 – 2-byte signed integer,
uint2 – 2-byte unsigned integer,
int1 – 1-byte signed integer,
uint1 – 1-byte unsigned integer.

The lengths of the array dimensions must be listed between brackets after the array name; they are comma-delimited. If no initial values are given, all array elements will be initially set to zero. The initial values of array elements may be specified between braces after the equality character following the array definition (as in C). The number of array initializers may be less than the total number of array elements (then only the first elements of the array will be initialized). The type name may be followed by definitions of several arrays, for example:

```
ARRAYS
        int a[10] = {10, 8,5, 4,
        3, 7, 6, 1, 2, 9}, b[20, 10]
        double c[30]
END ARRAYS
```

The commas between the initial values are optional (the values may be separated by spaces), and consecutive commas are ignored. As illustrated by the last example, the list of array initializers may be spread over multiple lines. However, breaking the line before the array name is not allowed (in the last example, the name of the array "b" cannot be placed on the next line), because each non-empty line in the code block "ARRAYS" must start either with the type name or with the value of an array element, or with the keyword "INDEX_BASE" (see the next paragraph).

By default, the array indices in MCNelectron are one-based, which means that the smallest allowed value of an array index is 1, and the largest allowed value is equal to the length of the array dimension (specified in the definition of the array). The array indices can be made zero-based (as in C) by inserting the directive "INDEX_BASE 0" at the beginning of the code block "ARRAYS". Then the smallest allowed value of an array index would be 0, and the largest allowed value would be one less than the length of the array dimension. The directive "INDEX_BASE 1" may be used to

indicate that the array indices are one-based (this would be redundant however, because the array indices are one-based by default).

If there is insufficient memory for an array, it will not be allocated. In such a case, MCNelectron would quit with an error message.

Some built-in functions that operate on arrays require a specific array data type (for example, the function "Invert" requires the array to have the data type "double"). In such cases, checking for the array data type is done at runtime, and if an unsupported data type is used, then the program will quit with the corresponding runtime error message (for more information about runtime error handling, see Section A.5).

### *A.3. Using subroutines*

According to Wikipedia, "a subroutine is a sequence of program instructions that perform a specific task, packaged as a unit. This unit can then be used in programs wherever that particular task should be performed." When discussing subroutines, it is important to distinguish between formal and actual parameters of a subroutine. Another definition from Wikipedia: a formal parameter is "the variable as found in the function definition, while *argument* (sometimes called *actual parameter*) refers to the actual value passed". For example, let us suppose that subroutine FUNC defines the following function of one variable: $f(x) = \ln(1 + x)$. Then the variable $x$, which is used in that definition, is the formal parameter of the subroutine. Now, if there is a variable "a" defined in the calling program, then in order to calculate the value of $\ln(1 + a)$, the subroutine FUNC must be called as follows: "FUNC(a)". In this call, the value of "a" is the actual parameter (or argument) of the subroutine FUNC.

In MCNEcode, each subroutine must return a value, and calls to subroutines may be used in arithmetic expressions. In this respect, subroutines are similar to built-in functions (see Section A.1.4). The value returned by a subroutine is the value of the last computed expression. However, execution of a subroutine can be terminated at any time by calling the built-in function "Return(x)", which was described in Section A.1.4.

A new subroutine is defined in the code block with the header "SUBROUTINE <SubroutineName>". The name of the subroutine can be optionally followed by the list of formal parameters. The list of formal parameters must be between parentheses. The left parenthesis must be on the same line as the subroutine name, immediately after it (if there is no left parenthesis after the name of the subroutine, or if the left parenthesis is on the next line, then the compiler will assume that the subroutine has no formal parameters). The list of formal parameters may be spread over several lines, and it may include comments and empty lines. The formal parameters must be separated by commas, spaces, tabs or line breaks. An example of a definition of a subroutine with four formal parameters is given below:

```
SUBROUTINE FUNC1(a, loc(b), loc(c[ , ]), loc(f(x, loc([ ]), loc([ , ])) )
    b = 2 * a
    c[1, 2] = sqrt(b)
    f(a, loc(c[5, 3]), loc(c))
END SUBROUTINE
```

If a given argument (actual parameter) of the subroutine is the address of a variable, array or function, then the built-in function "loc(…)" must be used when calling that subroutine. This is reflected in the list of formal parameters (see the last example). The rules of using the construct "loc(…)" in the list of formal parameters are most easily explained by explaining the meaning of each of the four formal parameters in the above example:

Formal parameter No. 1 ("a") is a variable that cannot be modified in the subroutine (the corresponding actual parameter is the value of a variable or an expression).

Formal parameter No. 2 ("b", declared as "loc(b)") is a variable that can be modified in the subroutine (the corresponding actual parameter is the address of a variable).

Formal parameter No. 3 ("c", declared as "loc(c[ , ])") is a two-dimensional array (the corresponding actual parameter is the address of a two-dimensional array).

Formal parameter No. 4 (“f”, declared as “loc(f(x, loc([ ]), loc([ , ]))”) is a user-defined subroutine whose first argument is the value of a variable or an expression (declared as “x”), the second argument is the address of a one-dimensional array (declared as “loc([ ])”), and the third argument is the address of a two-dimensional array (declared as “loc([ , ])”). “f” could also be a built-in function (if there was a built-in function that accepts such arguments).

If the formal parameter is a function, then the corresponding actual parameter must be “loc(<function name>)”. For example, if “x” and “y” are variables (parameters), “z” is a two-dimensional array, and “g” is a three-argument subroutine, whose formal parameters conform to the requirements stated in the description of parameter No. 4 in the above example, then the following call to the subroutine “FUNC1” corresponding to the mentioned example is allowed:

```
FUNC1(x, loc(y), loc(z), loc(g))
```

If the formal parameter is an array, then the corresponding argument (actual parameter) may be a part of an existing array. This is achieved by specifying one or more array indices in the “loc(…)” construct when calling the subroutine. The following convention is used: if the array used as an actual parameter has $N$ dimensions and if the number of indices of the actual parameter is $m$, then the subroutine treats that parameter as an array which has $N - m + 1$ dimensions and which starts from the element specified by the indices of the actual parameter. This convention is employed in the last statement of the subroutine “FUNC1” defined above: although “c” is a two-dimensional array, the construct “loc(c[5, 3])” is interpreted as the address of a one-dimensional array which contains elements of row No. 5 of the original array, starting with the element No. 3 of that row.

Recursion is allowed in subroutines (“recursion” means a call of a subroutine from within the same subroutine). However, unlike in some other programming languages (e.g., C), all instances of one subroutine share one memory space. This means that any temporary variable that has been assigned a value in one instance of a subroutine will have the same value in all other instances of that subroutine. The same is true of the formal parameters, too (except for the formal parameters that are pointers to arrays or functions): if a subroutine is called from within the same subroutine, then the values of the formal parameters of the calling instance of that subroutine (and of all higher-level instances of that subroutine) automatically become equal to the argument values used in that call. For example, let us assume that the global parameter “d” indicates the call depth of the subroutine “FUNC2(a)” and that initially d = 0. In addition, let us assume that the first four lines of the code of that subroutine are

```
d = d + 1
b = d
if (d < 2) FUNC2(a + 1)
d = d – 1
```

Then, if the formal parameter “a” is initially equal to 2 (e.g., if the code of the calling program is “FUNC2(2)”), its value immediately after executing the third line will become 3 (because such is the value passed to subroutine FUNC2 in the call to it from within FUNC2). The value of the temporary variable “b”, which was initially assigned the value 1, will become equal to 2.

***Notes***: **1.** Unlike formal parameters that are variables or pointers to variables (e.g., formal parameters No. 1 and 2 in the definition of the subroutine “FUNC1” above), formal parameters that are pointers to arrays or functions (e.g., formal parameters No. 3 and 4 in definition of the subroutine “FUNC1”) are not shared among different instances of one subroutine.

**2.** The last example shows how to track the call depth (also called “nesting level”). However, such a method of tracking the call depth would only work if there are no runtime errors when executing the subroutine FUNC2. After a runtime error, execution of the current instance of FUNC2 would be terminated immediately, and the global parameter “d” would not be decremented by 1 (for more information about runtime error handling, see Section A.5).

### *A.4. Built-in integration, summation, iteration and root finding functions*

MCNEcode has eleven built-in integration, summation, iteration and root finding functions. Seven of them were listed in Section A.1.4. The remaining four functions ("Int2", "Inti2", "Intw2" and "Root2") are "extended" versions of the previously-mentioned functions ("Int", "Inti", "Intw" and "Root") with additional arguments, allowing more control over the computation process (the original versions of those functions use the default values of the additional arguments). The integration functions were taken from the QUADPACK library for numerical integration of one-dimensional functions (the QUADPACK library is in public domain, web page: http://www.netlib.org/quadpack/), and the nonlinear equation solver is based on one of subroutines of the HOMPACK suite for solving nonlinear systems of equations, which is also available from the Netlib repository (web page http://www.netlib.org/hompack/). Those subroutines were translated from Fortran into C using the Fortran-to-C converter "f2c.exe" (also in public domain, web page: http://www.netlib.org/f2c/mswin/). Below are short descriptions of those functions:

> The function "Int" is a translated double-precision version of the Fortran subroutine QAGS (the double-precision version is named DQAGS). According to the QUADPACK readme file, "QAGS is an integrator based on globally adaptive interval subdivision in connection with extrapolation (de Doncker, 1978) by the Epsilon algorithm (Wynn, 1956)."
>
> The function "Inti" is a translated double-precision version of the Fortran subroutine QAGI (the double-precision version is named DQAGI). According to the QUADPACK readme file, "QAGI handles integration over infinite intervals. The infinite range is mapped onto a finite interval and then the same strategy as in QAGS is applied."
>
> The function "Intw" is a translated double-precision version of the Fortran subroutine QAWO (the double-precision version is named DQAWO). According to the QUADPACK readme file, "QAWO is a routine for the integration of COS(OMEGA*X)*F(X) or SIN(OMEGA*X)*F(X) over a finite interval (A,B). OMEGA is specified by the user. The rule evaluation component is based on the modified Clenshaw-Curtis technique. An adaptive subdivision scheme is used connected with an extrapolation procedure, which is a modification of that in QAGS and provides the possibility to deal even with singularities in F."
>
> The function "Root" is a translated Fortran subroutine ROOT. According to the subroutine's description, which is included in the comments section of the original source code, the method used for the solution is a combination of bisection and the secant rule.

The full list of built-in integration, summation, iteration and root finding functions is given below:

Int(f,x,a,b) – integral of expression "f" with respect to $x$ from $a$ to $b$ (e.g., "Int(1/sqrt(exp(t*x)+x),x,0,5)");
Int2(f,x,a,b,epsabs,epsrel,limit) – same as "Int", but with additional arguments:
epsabs – absolute accuracy requested (default value 0),
epsrel – relative accuracy requested (default value $10^{-5}$),
limit – the maximum number of subintervals in the partition of the given integration interval (default value 200).

Inti(f,x,a,1) – integral of expression "f" with respect to $x$ from $a$ to $+\infty$;
Inti(f,x,a,-1) – integral of expression "f" with respect to $x$ from $-\infty$ to $a$;
Inti(f,x,a,2) – integral of expression "f" with respect to $x$ from $-\infty$ to $+\infty$ (the argument "a" is not used);
Inti2(f,x,a,i,epsabs,epsrel,limit), where "i" is ±1 or 2 – same as "Inti", but with additional arguments:
epsabs – absolute accuracy requested (default value 0),
epsrel – relative accuracy requested (default value $10^{-5}$),
limit – the maximum number of subintervals in the partition of the given integration interval (default value 200).

Intw(f,x,a,b,w,1) – integral of expression $f(x)\cos(wx)$ with respect to $x$ from $a$ to $b$;
Intw(f,x,a,b,w,2) – integral of expression $f(x)\sin(wx)$ with respect to $x$ from $a$ to $b$.
Intw2(f,x,a,b,w,i,epsabs,epsrel,leniw,maxp1), where "i" is 1 or 2 – same as "Intw", but with additional arguments:
epsabs – absolute accuracy requested (default value 0),

epsrel – relative accuracy requested (default value $10^{-5}$),
leniw – twice the maximum number of subintervals allowed in the partition of the given integration interval (default value 400, i.e., the default maximum number of subintervals is 200),
maxp1 – an upper bound on the number of Chebyshev moments that can be stored (default value 100).

Sum(f,i,i1,i2) – sum of terms "f" when the summation index *i* varies from i1 to i2 (e.g., "Sum(ln(i),i,2,10)");
Sum2(f,t,a,b,dt) – sum of terms "f" when the summation variable *t* varies from *a* to *b* in increments of dt.
Iter(f,i,i1,i2) – repetition ("iteration") of expression "f" when the iteration index *i* varies from i1 to i2 (the returned value is the last calculated value of expression "f").

Root(f,x,a,b) – root of the nonlinear equation $f(x) = 0$. *a* and *b* are limits of the interval that should be searched for the root. For example, the expression "Root(sqrt(x)-1+x^3,x,0,1)" is equal to the root of the equation $\sqrt{x} - 1 + x^3 = 0$, i.e., 0.60542342357183;
Root2(f,x,a,b,RelErr,AbsErr,loc(flag)) – same as "Root", but with additional arguments:
RelErr – relative accuracy requested (default value $10^{-9}$),
AbsErr – absolute accuracy requested (default value $10^{-100}$),
flag – the "error code", which is modified by the function. If the equation is solved successfully, then "flag" is assigned the value zero, otherwise it is set to an integer number from 1 to 4, depending on the type of the difficulty that was encountered.

Argument No. 1 of all built-in integration, summation, iteration and root finding functions defines the expression to be processed, and argument No. 2 defines the variable of integration (equation) or index of summation, or index of iteration. The expression defined by argument No. 1 stays in scope of the calling program, so that all previously defined variables can be used in it. The variable of integration (equation) or index of summation takes priority over all other variables (including the system variables) and it is "visible" only in the processed expression, so that any identifier can be used in place of that variable. E.g., if there was a variable "x" defined in the calling program before the call to the integration function "Int" in Example No. 1 below, then the value of "x" would not be affected by the integration.

Multiple integrals can be computed by defining the integrand (i.e., the integrated expression) as a call to an integration function (see Example 2 below).

The original QUADPACK integration functions have an integer argument that serves as an "error code". If a difficulty is encountered during integration, then the error code is assigned a non-zero value ranging from 1 to 6 and indicating the type of the error. In such a case, the MCNelectron output file contains an error message inserted after the line of the program code where the error occurred (for more information about runtime error handling, see Section A.5).

Every call to any built-in integration function causes an update of the following three system variables:

IER – the mentioned error code,
AbsErr – estimate of the modulus of the absolute error,
nEval – number of integrand evaluations.

Those three variables are set to zero before starting execution of each user program. If a call to a built-in integration function cannot be initiated (e.g., due to insufficient memory for work arrays of the function), then the system variables "IER", "AbsErr" and "nEval" are not modified.

Examples:

**1.** The integral of the function $\exp(-x^2) \cdot \exp(-(t-x)^2)$ with respect to *x* from 1 to 10:

```
result = Int(exp(-x*x) * exp(-(t-x)^2), x, 1, 10)
```

**2.** The double integral of a user-defined function FUNC1(a,b):

```
result = Int(Int(FUNC1(a,b), a,a1,a2), b,b1,b2)
```

**3.** Example of integration from 5 to $+\infty$:

```
result = Inti(exp(-x*x) * exp(-(t-x)^2), x, 5, 1)
```

(when integrating from $-\infty$ to a finite bound, the last argument should be -1).

**4.** Example of integration from $-\infty$ to $+\infty$:

result = Inti(exp(-x*x) * exp(-(t-x)^2), x, 0, 2)

(in this case, argument No. 4 must be 2 and argument No. 3 can be any value).

**5.** The sum of products of the corresponding elements in row 2 of the matrix A and in column 3 of the matrix B:

result = Sum(A[2,x]*B[x,3], x, 1, Size(loc(A[1,1])))

### *A.5. Runtime error handling*

A runtime error is an error that occurs during execution of a program and that cannot be anticipated at compile time. In order to minimize the risk that a runtime error in a user program will crash the MCNelectron executable, the interpreter of the bytecode generated by the MCNEcode compiler checks for the following types of errors at runtime:

1) out-of-bounds array indices (e.g., negative indices, or indices greater than the number returned by the built-in function "Size2(loc(a), i)"),
2) abnormally long computation (e.g., due to an infinite loop),
3) infinite recursion,
4) runtime errors related to limitations of certain built-in functions (for example, incorrect data type of an array whose address was passed to the function as an argument).

An abnormally long execution of a user program (such as caused by an infinite loop) is handled by displaying the following message if the execution time of the program is longer than 2 seconds:

```
Execution time of the user program exceeds 2 seconds. Press the escape key
and then 'Q' to attempt to interrupt execution of the user program.
```

When this message appears, the execution of the user program is not paused. The user can either continue waiting until the program ends by itself, or stop the computation by pressing 'Esc' and 'Q' (in that order). If only 'Esc' was pressed, then it is possible to "cancel" the escape key by pressing any key except 'Esc' and 'Q'. After that, the 'Esc' key would have to be pressed again in order to interrupt the program.

When a runtime error occurs or when the program is interrupted by the user, execution of the program is terminated immediately. I.e., all operations that should be normally done after the point where the error occurred are skipped. In such a case, the value returned by that program is set equal to the indeterminate number ("-1.#IND"). If the current program is a subroutine, then execution of the calling program continues from the instruction that immediately follows the call to that subroutine. The calling program can check for the occurrence of a runtime error in the subroutine by comparing the value returned by the subroutine to itself (if "value" is equal to the indeterminate number, then the expression "value==value" is equal to 0; otherwise it is equal to 1).
***Note***: Integration errors, which are indicated by a non-zero value of the error code "IER" assigned by built-in integration functions (see Section A.4), do not cause interruption of the program execution. However, runtime errors that may prevent a call to a built-in integration function from being initiated are possible (e.g., insufficient memory for work arrays of the function). After such an error, execution of the program is terminated as described above, and the system variables "IER", "AbsErr" and "nEval" are not modified.

The first runtime error that was detected in the user program is described in an error message inserted after the corresponding line of the program in the MCNelectron output file. If the runtime error occurs in a subroutine, then the line containing the corresponding call to that subroutine is similarly marked in the code of the calling program, too.

**References**


[1] Hughes H. G., *Recent developments in low-energy electron/photon transport for MCNP6* // Progress in Nuclear Science and Technology, Vol. 4 (2014) pp. 454-458

[2] CUDA Toolkit Documentation. [Online]. Available: http://docs.nvidia.com/cuda/index.html

[3] Bote D., Salvat F., Jablonski A. and Powell C. J., *Cross sections for ionization of K, L and M shells of atoms by impact of electrons and positrons with energies up to 1 GeV: Analytical formulas* // At. Data Nucl. Data Tables, vol. 95 (2009) pp. 871-909

[4] Salvat F., Jablonski A. and Powell C. J., *ELSEPA—Dirac partial-wave calculation of elastic scattering of electrons and positrons by atoms, positive ions and molecules* // Comp. Phys. Comm., vol. 165 (2005) pp. 157-190

[5] Perkins S. T., Cullen D. E. and Seltzer S. M., *Tables and Graphs of Electron-Interaction Cross Sections from 10 eV to 100 GeV Derived from the LLNL Evaluated Electron Data Library (EEDL), Z = 1 – 100*, Lawrence Livermore National Laboratory, Livermore, CA, Report UCRL-50400 Series, Vol.31, 1991, 486 p.

[6] X-5 Monte Carlo Team, *MCNP — A General Monte Carlo N-Particle Transport Code, Version 5. Volume II: User's Guide*, Los Alamos National Laboratory, Los Alamos, NM, LA-CP-03-0245, 2008, 508 p.

[7] Trkov A., Herman M. and Brown D. A. (ed.), *ENDF-6 Formats Manual*, National Nuclear Data Center, Brookhaven National Laboratory, Upton, NY 11973-5000, 2012. 378 p.

[8] Köhn C. and Ebert U., *Angular distribution of Bremsstrahlung photons and of positrons for calculations of terrestrial gamma-ray flashes and positron beams* // Atmospheric Research, vol. 135-136 (2014) pp. 432-465

[9] Kim L., Pratt R. H., Seltzer S. M. and Berger M. J., *Ratio of positron to electron bremsstrahlung energy loss: An approximate scaling law* // Phys. Rev. A, vol. 33 (1986) pp. 3002-3009

[10] Slater J. C., *Atomic shielding constants* // Phys. Rev., vol. 36 (1930) pp. 57-64

[11] Frenkel J., *Some remarks on the theory of the photoelectric effect* // Phys. Rev., vol. 38 (1931) pp. 309-320

[12] Salvat F. and Fernandez-Varea J. M., *Overview of physical interaction models for photon and electron transport used in Monte Carlo codes* // Metrologia, vol. 46 (2009) pp. S112-S138

[13] Seltzer S. M., *An Overview of ETRAN Monte Carlo Methods* // Monte Carlo Transport of Electrons and Photons, ed. T. M. Jenkins, W. R. Nelson and A. Rindi, Plenum Press, New York (1988). ISBN 0–306–43099–1

[14] Tsai Y., *Pair production and bremsstrahlung of charged leptons* // Rev. Mod. Phys., vol. 46 (1974) pp. 815-851

[15] Sood A., *Doppler energy broadening for incoherent scattering in MCNP5, Part I*, Los Alamos National Laboratory, LA-UR-04-0487 (2004)

[16] Sood A., *Doppler energy broadening for incoherent scattering in MCNP5, Part II*, Los Alamos National Laboratory, LA-UR-04-0488 (2004)

[17] Biggs F., Mendelsohn L. B. and Mann J. B., *Hartree-Fock Compton profiles for the elements* // Atomic Data and Nuclear Data Tables, Vol. 16, No. 3 (1975), pp. 201-309

[18] Taylor J. R., *An Introduction to Error Analysis: The Study of Uncertainties in Physical Measurements, 2nd Edition*, University Science Books, Sausalito, CA, 1996, 327 p.