HEEDC++ 2.10 (28.02.2010)

Programs and class library for modeling ionization produced by fast charged particles in gases

User's Guide

I. B. Smirnov

The generation date of this version of this document: April 12, 2010.

Contents

1 Introduction

When a charged particle passes through gas (or, with some reservations, through liquid or solid substance), it leaves a part of its energy in a distinct collisions with atoms. The atoms emit photoelectrons and are left ionized with vacancies at atomic shells. These shells are filled by various relaxation processes such as fluorescence and Auger electron emission. The emitted electrons and photons are absorbed and can initiate another relaxation processes. An accurate modeling of these processes is interesting problem, which is important for the simulation and research of gaseous detectors of fast charged particles. The corresponding model, the photoabsorption and relaxation model, or, shortly, PAIR, is described in ref. [1].

The cited paper describes only physics. This model however undermines quite a large computer program, which includes handling large amount of various atomic data, describing materials, performing geometrical calculations, modeling volumes and trajectory propagation through them, deflections of particles by fields, random interactions of particles with matter, emission and absorption of particles, histogram filling. Obviously, all of this can constitute a full-scale Monte Carlo system. A first distributed computer program for this porposes, HEED-F77, "version 1", was developed in 1994-97 in Fortran-77. Fortran is Fortran. Although many processes were encoded, the program was limited in many respects. In particular, it had simplified geometry (a parallel layers model), and did not take into acount fields (in the author's original version). Speaking scientifically, it was not generic, scalable and reusable enough. Speaking ordinarily, it was also tedious. It was not a model, but rather an algorithm written in an obsolete programming language for restricted purposes. However, the old program has reportedly found some interesting applications in the detector research for high energy physics (see references in [1]). This has motivated the author to remake this program in C++ as a more coherent and conceptual computer model, to prepare this manual, and to offer the new program, or, more precisely, a class library and a set of example or test programs, as possible replacement of the old one.

It is assumed that the program can be used for free for academic non-commercial research and for educational purposes.

The program is provided "as is" without express or implied warranty.

The term "fast" means that the particle is moving faster then the atomic electrons do. Otherwise the theoretical expressions loss their precision. Also, the energy loss is assumed to be much less than the initial kinetic energy of the incident particle, and the latter is not changed. The both circumstances mean that the incident particle is not assumed to be stopped in the media (this is needed for many researches and frequently asked). But the δ-electrons, which are traced less precisely, can stop in it. In practice, the program was tested and compared with experimental data for velocities of the incident particle around the ionization minimum and higher, up to Fermi plateau and including it, see ref. [1].

The program currently does not change the energy of the incident particle along its trajectory. This means that the particle initial energy has to be much higher than the possible energy loss in the layer.

The program may also trace photons of moderate energies and absorb them by photoeffect with further relaxation processes. (The energy is restricted due to neglecting the Compton effect and other types of interactions which are significant at higher energies.)

Obviously, this program offers also a convenient platform for other simulations of interactions of particles with matter and for detector studies, e.g. for wider range of processes. This platform can also be used (and actually was used) for a treatment of experimental data.

Most of the gas-filled detectors used in high energy physics and in other fields are proportional chambers of various kinds. They amplify the initial ionization deposited in them, but their output signals are usually proportional to the charge deposited. Therefore the simulation of initial ionization can be applied for estimate of the final chamber characteristisc. If the user wants to simulate ionization drift and amplification, the coordinates of conduction electrons can be output from this program and input in another detector simulation code.

To solve a particular problem with the help of HEED the user is expected to write the main C++ program, which initializes objects of certain classes such as atoms, moleculas, gases, cross sections, geometrical volumes, possibly with other embeded volumes, and compound objects involving gases, cross sections and volumes. The main program can also initialize histograms. Then it should run the loop over necessary number of events, fill histograms (if any) and possibly save them to disk for further study.

Also provided is the possibility to call the new program from a Fortran-77 program like the old Fortran-77 version of HEED was called.

-

2 Dependencies on other software

2.1 Histogram filling and visualization

Obviously the most significant issue related to dependencies is histogramming issue, which is mentioned above. Of course, the user can edit the source text as well as build scripts and connect the core physics program to any other histogramming system or disconnect the program from any histograms at all. This would not be extremely complicated since most of histograms are located in "main" programs of examples, thus the physics parts are almost independent on them. But in the supplied version of package histograms are included and we will describe them as they are.

The HEED++ program was developed and thoroughly tested with an old HBOOK and PAW programs for histogram filling and visualisation [2] (HEED-F77 was developed on the base of HBOOK and PAW too). This HBOOK-PAW system is sufficient for the purpose of HEED and perhaps still enough for many other purposes. But being based on the obsolete FORTRAN-77/ZEBRA technology this system is not supported anymore in CERN, in which it was created. One can expect that it might be more and more heavier to obtain and to install this system on new computers, as well as to find people who possess knowledge about it, who understand how to run it or who wish to study it. ("Dependencies on other software" frequently implies also dependencies on other knowledge.) But fortunately HEED++ was previously connected to this system through a C++ interface package, which could easily be readjusted to another newer libraries. We may call this package "histdef" by the name of its `histdef.h` and `histdef.c` files This readjustment is now done as by-product of other works. By means of macro-definitions "histdef" can be switched to either HBOOK or one of alternative "drivers", whereas the user interface of "histdef" remains (nearly) constant. Be warned that these alternative possibilities are not carefully tested and oversights related to them are not excluded. So use these possibilities with care on your own responsibility (in fact, as well as the whole program). These possibilities are AIDA-interface of OpenScientist [3] and a stand-alone GHist (General Histogram) histogramming package written by the author and interfaced with ROOT [4] for visualisation. The current recommendation is to invoke these alternative possibilities only if you are expert in AIDA/OpenScientist or ROOT programs and have them in your workstation and if you are not aware of HBOOK-PAW, do not have it, and to not wish to change these circumstances.

The AIDA-interface of OpenScientist fills and writes histograms by the means of OpenScientist. Therefore it requires the OpenScientist's include files and libraries to be present at the compile time of HEED++, and, since OpenScientist supplies dynamic libraries, its libraries are necessary also at execution time (provided that the connection to OpenScientist is requested; obviously, otherwise any OpenScientist's components are not necessary). To see histograms you will need OpenScientist's binaries installed as well. (Currently the author has some troubles in running these binaries; probably the troubles are temporary and not related to supplied package. But in general the author is not convinced that either AIDA or OpenScientist is the best choice. AIDA is based on dangerous raw pointers and awkward "factory" technique. Visualization through OpenScientist's "opaw" is not free from deficienies.)

The GHist histogramming package fills and writes histograms itself (in its own format), and therefore it does not require anything else be present. This very new and absolutely experimental package currently supports visuallisation only through ROOT (and only a small subset of GHist is currently visualized, but this is enough for HEED++). To see these histograms you will need ROOT system installed and a dynamic library invoking GHist and ROOT components compiled. Of course, the source code and build script for this library are supplied. In theory, you may have ROOT system installed at another computer and GHist package installed on both. The author assumes (but this is yet to prove in practice) that the combination of histogram filling through a system like GHist (not histdef or GHist though histdef, but HEED++ can call GHist only though histdef) and plotting probably through ROOT may be the optimal choice for many (or most) physics research in the current cituation. Note that ROOT with GHist is used only as a kind of graphics driver. Author does not belong to school of thought or group of exclusive supporters of ROOT or AIDA or OpenScientist. All have deficiencies. (But we should anyway be gratefull to all authors of these products for their exploration of these different concepts or methodologies and for making the results available for public use, which allows us today to choose between them.) Some funny reading about "HEP soft-wars" presenting one point of view is currently presented at [3]. Some other opinions are also gathered at [5].

2.2 Random numbers

HEED++ in its original configuration uses also an old random-number package RANLUX from CERNLIB. The use of this set of subroutines is motivated by its verification in other applications and good reputation. The HEED++ program is connected to this set of Fortran subroutines through an interface package. This make possible to replace Fortran random number generators by something else.

Alternatively the random numbers can be generated either by CLHEP random-number subsystem, or by a local subsystem which is copy of CLHEP random-number classes with corrected errors (see `https://savannah.cern.ch/bugs/?58514`).

Thus the program needs either "packlib" component of CERNLIB, or CLHEP to be installed, or can use locally kept corrected copy of CLHEP.

2.3 Summary for include files and libraries

Thus, the external dependencies are packlib from CERNLIB (where HBOOK, RANLUX and PAW resides), `std::string` from STL (presented at any contemporary system), `SystemOfUnits.h` and `PhysicalConstants.h` from CLHEP (the local copies are avaliable, so the installation of CLHEP is not necessary), and possibly OpenScientist or ROOT as alternatives to PAW.

If you do not use HBOOK and RANLUX from CERNLIB, the CERNLIB is not necessary, and any Fortran ingredients are not involved. If in addition you don't have CLHEP installed, you can use locally kept corrected copy of random number package. In this case there are no external dependencies at all!

If you use RANLUX from CERNLIB, you will need also Fortran-77 compiler, in order to compile some interface file.

2.4 Operating systems

The program was developed and tested on 32-bit Red Hat Linux systems of various versions with GNU compilers and, correspondingly, it is expected to function correctly on them. Some components of `wcpplib` (see below, namely `stream`, `util`, `safetl`, `math`, `matrix`) were also compiled and tested at Windows under Microsoft Visual Studio. The corresponding description will also be included for the sake of completeness and having in mind that the main problem with porting of the rest of the program under Windows is perhaps absence of free time which should be devoted for it, while some of users could try to do this themselves. Nevertheless, no other comments can currenly be given for the other packages and systems.

All the UNIX commands and scripts are written for (t)csh command interpreter. The Windows scripts for the ported parts of `wcpplib` are written for command prompt run from the Visual Studio desk and correspondingly having necessary compiler's environment variables initialized.

3 Structure and directories

The program consists of a general-purpose library called "wcpplib" (which was deemed to mean "the working C++ library") residing in directory with name `wcpplib`, and a special directory structure `heed++` (from "high energy electrodynamics") devoted to ionization and residing in directory `heed++`. The fragments of `wcpplib` library distributed with HEED++ contain about 30 thousand lines of C++ code. The source code in `heed++` directory (without database and auxiliary programs used in its preparation) totals about 14 thousand lines.

3.1 WCPPLIB

The `wcpplib` package prepared for `heed++` has the following components residing in own sub-directories of wcpplib:

- `stream` – handling streams and indentation for output listings.
- `util` – miscellaneous utility classes and macro-definitions, in particular, debug facilities and facilities for emulation of some operators of the new C++ standard for old compilers.
- `safetl` – SafeTL or safe template library: special smart pointers (see appendix in [6], [7], and section 9.3 farther in this guide), dynamic arrays and lists;
- `math` – various mathematics classes, in particular, `linet.h` template library for integration and interpolation, and many many other classes and functions;
- `matrix` – matrix operations with arrays determined in SafeTL.
- `random` – interface for random number generators, currently connected to `RANLUX` from CERNLIB.
- `graphics` – histograms, namely a generic interface, which is currently adapted for HBOOK Fortran subroutines from kernlib, which is from CERNLIB.
- `geometry` – geometrical calculations, very large package which starts from definition of vector and finishes by tracing generic particles (which here means just a point in 3-D space with a position and speed) in a tree-like structure of arbitrary volumes.
- `matter` – definitions of atoms, moleculas, gases.
- `particle` – definitions of particles (real physical particles with mass, charge etc.)
- `ioniz` – definitions of various Bete-Bloch formulas (for debug and comparisons).
- `clhep_units` – copy of system of units from CLHEP.
- `cfortran` – auxiliary files for connection to Fortran.

- `install` – installation utilities, currently with a single program "`concom`" which simplifies interface with make-related staff at UNUX. All "`build.b`" scripts use this program. Therefore it should be compiled first prior to wcpplib by a regular script (which is trivial and it is supplied) and put into directory registered in the `PATH` environment variable. Of course, `concom` does not depend on the rest of wcpplib, but wcpplib at UNUX is compiled with the help of the executable `concom`.

- `doc` – GNU LGPL license (covering a part of this software), small installation summary, file `version.txt` determining the version number, and file `date_of_wclhd_arch_creation.txt` determining the date of creation of this package, that is the date when the source files were put into archive by running corresponding script (they, of course, are created earlier themselves). These version number and date should (reasonably) correspond to what you see at the title page of this manual.

The following important files reside in `wcpplib` directory:

- `buildhd.b` – building (compiling and making the libraries) for the total `wcpplib` for UNIX.

- `clearbin.b` – removing all binary libraries, compiled files ("object files") and template database (for those systems where it exists) for UNIX

- `build_vs.bat` – building in Windows (compiling and making the libraries for the part of `wcpplib`, which was ported to Windows.

- `clearbin_vs.bat` – removing all binary libraries, compiled files ("object files") and template database (for those systems where it exists) in Windows.

3.2 HEED++

The `heed++` directory has three main subdirectories:

- `code` – the heed library and a range of test programs. This directory should have also `fig` subdirectory, where some test programs output data files (if it is not exist just after unpacking of tar-archive, please create it).

- `database` – database of atomic data. This is not classic "relational" database, but just a collection of variuos files, which are read at run-time (normally at the initialization stage).

- `forinterf` – interface package for calling `heed++` from Fortran programs.

4 File types and extensions

The following rules are used for naming files.

- `*.c` – C++ source code (this extension is usually reserved for C-language in other programs; since the author does not use the plain C language in this program, mistakes are impossible; shorter names are more convenient for handling disk files).

- `*.ic` – inlined C++ code.

- `*.h` – header files for C++ code.

- `*.f` – Fortran-77 source code.

- `*.o` – compiled source files ("object files") for UNIX.

- `*.obj` – compiled source files ("object files") for Windows. Windows also generates additional service files `*.ilk` and `*.pdb`.

- `build.b` – scripts for compilation of source files in a given directory or its subdirectories and building the libraries.
- `buildhd.b` – the same but adapted for `heed++`. Only those files and subdirectories are included in the archive distributed with `heed++`, which are used in it. If in the source author's directory there is something irrelevant compiled by the original `build.b`, the absence of such a file in the user's directory would naturally lead to failure of compilation. Therefore in such cases instead of `build.b` the archive contains modified public versions `buildhd.b` in which the references to extra files are deleted. Not very convenient, but working.
- `build_vs.bat` – scripts fot compilation of source files in a given directory or its subdirectories and building the libraries for Windows Microsoft Visual Studio C++.
- `lib*.a` – static library of "object files" for UNIX.
- `lib*.lib` – static library of "object files" for Windows.
- `lib*.so` – dynamic libraries for UNIX. Today only one such library is presented (because it is necessary). This is `libRootMGHist.so` loaded in ROOT in order to make it able to read GHist-created histograms.
- `AnyName_ts.c` – test program for testing the class or the package indicated by `AnyName`.
- `AnyName_ts.b` – script for compilation and linking the test program for UNIX.
- `AnyName_vs_ts.bat` – script for compilation and linking the test program for Windows.
- `AnyName_ts.e` – executable file of the test program in UNIX.
- `AnyName_vs_ts.exe` – executable file of the test program in Windows.
- `AnyName_ts.r` – script for running the test program that produces very long output listing. Only for UNIX. The sole mission of these scripts is to delete old file with listing, to run the executable file and to redirect new output to this file.
- `AnyName_ts.out` – output listings of test programs, which are generated by *.r scripts.
- `AnyName_ts.eout` – output listings of test programs supplied by the author in order to allow the user to compare output at his computer with it. Use command
  `diff AnyName_ts.eout AnyName_ts.out`
  in UNIX or
  `FC AnyName_ts.eout AnyName_ts.out`
  in Windows.
- `*.dat` – files with computer readable (`heed++/database`) or writable (`heed++/code/fig`) data.
- `*.hbook` – histogram files generated by HBOOK and intended for reading and visualization by PAW.
- `*.hbook.l` – files with macros, which determine the symbolic names for the histograms. The symbolic names are used in `*.kumac`-files, which is incredibly more convenient than the use of numbers.
- `*.kumac` – script files for PAW.
- `*.aida` – histogram XML files generated by AIDA/OpenScientist and intended for reading and visualization by opaw or onX.

- `*.ghist` – histogram files generated by GHist and intended for reading and visualization by ROOT with GHist-supporting dynamic library.
- `*.ghist.l` – auxiliary files with names of histograms and some additional information, which are simply intended to be opened in a separate window and used to remember and to type quickly their notations in ROOT session.
- `*.txt` – ASCII files with documentation.
- `*.ps` – Post-Script files with documentation.
- `*.eps` – encapsulated Post-Script files with plots, which might be created by some `*.kumac` scripts.
- `*.mktar` – scripts to make tar-archives. The archives supplied were made with these scripts. As the reader could see from the text of these files, before archiving these scripts are copied into the header archived directory and then they are archived as ordinary files for the purpose of investigation of various archiving missings and bugs.

5 Installation

We will consider in this section installation at UNIX-like systems.
It is meant installation of source code and its compilation, link and running.
The installations sequense consists approximately of the following:

1. Copy from my internet page `http://cern.ch/ismirnov/heed/` and install in your working directory two files `wclhd.tar.gz` and `heed_ex.tar.gz` (unless you have done that already).
2. Unpack them with `gunzip filename` and `tar -xvf filename` commands. You will see two directory trees rooted from `wcpplib` and `heed++`, respectively.
3. Enter to subdirectory `wcpplib/install` and run stript `concom.b` . This will compile and link a little program `concom.c`, which is used for simple generation of makefiles. The executable version should now appear in the same directory with name `wcpplib/install/concom.e` . But it is not mandatory that it resides there. It can be placed in any directory included in your `PATH` environment variable and have name `concom` without extension. So, after you find an appropriate place for it, you please rename it accordingly. Also make sure that it is seen by your shell. You can check this by command "`which concom` " . It sometimes happens that shell does not see new commands which appear after its launching. If this is the case, you need either to re-run the shell or to use shell built-in command `rehash`.
4. Look in chapter 6, and take a short look in chapter 7 of this manual.
5. Take file `wcpplib/install/wclhd_setup.b`, put it or its content in convenient place and modify it accordingly to your preferences.
6. Run it by “source”.
7. Change to some small and “trivial” enough subdirectory of wcpplib, for example to `wcpplib/stream` and run build script (`buildhd.b` or `build.b`). It should compile the source files and put them into library file. In some rare cases instead of “archiving” a library such scripts simply copy the “object” files to upper directory, from which they are took later on at linking other programs.

Look what build*.b says and depending on that you might need to edit the content of wcpplib/install/wclhd_setup.b additionally, then you are advised to clean binaries by rm *.o (because the "mechanics" of build*.b does not usually detect the changes of environment variables) and to run build.b again.

Also be aware that I do not normally check the functioning of all these compilation and linking scripts in all mossible modes and as if they are installed on another's computer, clean of anything pre-installed. Because of huge number of various possible combinations of environment variables, scripts, files to compile, test programs to link, directories, computer environments, operating systems, compilers, librariries it is above my possibilities to test every possible way of work. I mainly supply the packages as they are currently (or may be with some delay) work at the computers at which I am running them for my purposes. Therefore you might find in scripts and these instructions any my problems as well, starting from missing some files in supplied archives (about which you are kindly invited to inform me) and to any problems in scripts resulted to compilation or linking errors (about which you are also invited to inform me, if you have constructive information, but not just that "it does not work").

8. When "a sample" subdirectory is compiled successfully, change to root directory of wcpplib, that is to .../wcpplib, and run buildhd.b. Again you might need to adjust your environment and run wcpplib/clearbin.b script in order to clean before the next attempt of full rebuild.

9. When you finish this, you may test if this stuff links and works at your computer. For this change to wcpplib/stream,
run prstream_ts.b,
run prstream_ts.e,
run prstream_ts.e > ttt.txt,
run diff prstream_ts.eout ttt.txt.
If you see difference, something could be wrong. For this trivial test it is unlikely that any difference could be detected. For some much more complicated tests diff sometimes detects differences, for example when I change the code and forget to rewrite *.eout files, or when a raw machine addresses are output — they are naturally to be different in different computer. Therefore study and interpret the output of diff according to common sence.

   Much more complicated tests could be done by changing to wcpplib/safetl, and running run_all_tests_ex.b. This will try many tests, that is link many tests programs, run each and and for each run diff to compare results. In the contrast to the procedure described above this script does all the procedure automatically. One of the programs is run by it many times in different modes. During these trials its error-detection system triggers various errors, which should coincide with what is expected. You have to analyse the results which are output to your screen. It is usually clear whether everything is OK or not OK. If diff detects difference, its output is quite obvious. The tests themselves takes a few seconds of computer time.

   Next test which you could execute before trying to compile HEED is very short and trivial program wcpplib/sranlux_ts.c. However, it might be far less than trivial to compile and link it, as well as to execute it! It tests random generators which are either RANLUX from CERNLIB, or some "engine" from CLHEP. Correspondingly, you will need to adjust your environment, and when this program is executed (I do not provide a file for comparison, just look at output) you can congratulate youself that you have adjusted this non-easy issue.

   Then, especially if you have choosen HBOOK as your graphic driver, you can pass to subdirectory wcpplib/geometry and execute test vec_random_ts.b and vec_random_ts.e.

The latter will produce 3 files `vec_random_ts*.out`, which can be then read and "shown" by executing

```
shell> paw
paw> exe vec_random_ts
```

... and push "enter" a few times to change to the next figure. If you see strongly non-uniformly filled areas (filled by points or lines), the random number generator you choosen works not well, and otherwise well. There should be some figures filled.

The next possible step is to move to `wcpplib/graphics` and to execute
`simple_graph_prog.b` and `simple_graph_prog.e`
and then try to show histograms graphically according to receipts from chapter 12.

10. Change to directory `.../heed++/`.
11. Take file `heed++/install/heed_setup.b`, put it or its content in convenient place and modify it accordingly to your preferences.
12. Run it by "source".
13. Change to subdirectory `.../heed++/code`.
14. Run `build.b`.
15. Run `Spectr_ts.b`.
16. Run `Spectr_ts.e`.
17. Study its output histograms according to your environment and receipts from chapter 12.

6 Environment variables

Many scripts used for compilation and linking require some environment variables to be set. In this chapter we will describe them.

6.1 For wcpplib

For compilation and link of `wcpplib` you will need the following UNIX environment variables installed:

- `CPPMAKEOPT` – is read in program concom (using a C++ function `getenv(char* name_of_var) )` and is supplied to cpp (C preprocessor) in order to instruct it to output a rule suitable for make. Recommended values:

  ```
  setenv CPPMAKEOPT "-MM"     # linux
  setenv CPPMAKEOPT "-xM"     # Solaris
  ```

  If this variable is not defined, `concom` will use `-M`.

- `CPPCOMPILE` – compilation command
  `CPPCOMPILED` – command for compilation with debug information
  `CPPLINK` – compilation and link command
  `CPPLINKD` – command for compilation and link with debug information
  Normally used in `build.b` scripts as the first argument of call of `concom`.

  Recommended values (for a few typical systems):

– Scientific Linux, version 4.1:

```
setenv CPPCOMPILE "g++ -c  "
setenv CPPLINK    "g++     "
setenv CPPCOMPILED "g++ -g -c  "
setenv CPPLINKD    "g++ -g  "
```

– Linux, RedHat version 8.0:

```
setenv CPPCOMPILE "g++ -DCOPY_RETURNS_COMMON_BASE -c  "
setenv CPPLINK    "g++ -DCOPY_RETURNS_COMMON_BASE   "
setenv CPPCOMPILED "g++ -DCOPY_RETURNS_COMMON_BASE -g -c  "
setenv CPPLINKD    "g++ -DCOPY_RETURNS_COMMON_BASE -g  "
```

(Remark: this makes all classes handled by active pointers derived from a special common base class. This is needed in order to bypass a missing feature in older GNU compilers. The test program `wcpplib/safetl/ActivePtrT_ts.b` has to be compiled without `-DCOPY_RETURNS_COMMON_BASE`. This program is concieved as trivial example, which does not need derivation from common base and for clarity it does not contain macros that normally allow to switch between this derivation and non of it.)

– Linux, RedHat older versions:

```
setenv CPPCOMPILE "g++  -fhuge-objects -c  "
setenv CPPLINK    "g++ -fhuge-objects  "
setenv CPPCOMPILED "g++ -fhuge-objects -g -c  "
setenv CPPLINKD    "g++ -fhuge-objects -g  "
```

– Solaris and most of the others:

```
setenv CPPCOMPILE "CC -c"
setenv CPPCOMPILED "CC -g -c"
setenv CPPLINK  "CC "
setenv CPPLINKD  "CC -g"
```

- FORTRAN77 – name of fortran compiler. Recommended values:

```
setenv FORTRAN77  g77   # linux
setenv FORTRAN77  f77   # most of other platforms
```

- `ARCOMMAND` – command for making libraries of wcpplib and heed (archives of object files). Recommended value:

```
setenv ARCOMMAND "ar -rs"
```

- `WCPPLIB` – location of the root directory of wcpplib for searching include files. The include files in wcpplib are included with prefix `wcpplib/subdirectory/...` like the following: "`#include "wcpplib/safetl/AbsArr.h"`". This allows to avoid mess between the local files and possible system files with the same names (although no such coincidences are ever noticed). This variable is used in `build.b` scripts as argument of `concom` command, which passes it to preprocessor, `make`, and compiler. The example of initializstion:

```
setenv WCPPLIB $HOME
```

(if subdirectory wcpplib is located right in `HOME`)

- `WCPPLIBOBJ` - location of libraries and some object files of wcpplib. Used at link of executable programs. For example:

```
setenv WCPPLIBOBJ $WCPPLIB/wcpplib
```

- `CERNLIB` - location of CERNLIB, for example:

  ```
  setenv CERNLIB /cern/pro/lib
  ```

- `SPACKLIB` - the "special" packlib library. Normally only a little portion of packlib is necessary. The other huge amount of code contains calls of some system-related routines stored in other libraries. To guarantee portability of the user code the user can prepare a restricted variant of library and point to it by this variable. If the user can link his program with the original total library, he can point by `SPACKLIB` to the regular `packlib`. For example, under linux it is made like this:

  ```
  setenv SPACKLIB "-L/cern/2000/lib -lpacklib  -lnsl "
  ```

  In CERN at lxslc3 cluster one will need to add `-L/usr/local/lib  -shift`. The signature of missing this is

  ```
  /cern/pro/lib/libpacklib.a(cfclos.o)(.text+0xa): In function ‘cfclos_’:
  : undefined reference to ‘rfio_close’
  ...
  ```

  CERN itself seems to distribute a binary version of `libpacklib.a` without `shift`, which is called `libpacklib_noshift.a`. If you are outside CERN, you can either set

  ```
  setenv SPACKLIB "-L/cern/2000/lib -lpacklib_noshift  -lnsl "
  ```

  or to replace (copy) `libpacklib.a` by `libpacklib_noshift.a` and set

  ```
  setenv SPACKLIB "-L/cern/2000/lib -lpacklib  -lnsl "
  ```

- `CFLIBSERV` – contains notation of any auxiliary "serving" library(ies) in the form readable by linker. The library, one or many, is meant to contain auxiliary routines, which are necessary for linking Fortran subroutines to a C/C++ program. Recommended values:

  ```
  setenv CFLIBSERV "-lg2c"    # linux
  setenv CFLIBSERV " -lF77 "  # most of other platforms
  ```

- `USE_SRANLUX` – use `SRANLUX()` inline function to supply random number. This variable should always be switched on.

  Some files from wcpplib which needed random numbers can work by direct call of alternative CLHEP random engine, but this feature is not implemented in the whole library and in the whole program (because of uncertainty whether this package is for long. Previously some 5-7 years ago it did not work properly, and somewhen during this perious it seemed to be replaced by similar but different parkage. So what will be later nobody knows.) So the normal way of operation in calling random numbers indirectly through `SRANLUX()`, which can be switched (in file `wcpplib/random/ranluxint.h`) either to Fortran RANLUX or to CLHEP "engine". You may find the calls of `SRANLUX()` in namy places of this program.

  Therefore variable `USE_SRANLUX` should always be switched on.

  `USE_CPP_SRANLUX` – instructs the system to switch SRANLUX to CLHEP "engine" instead of Fortran runlux, which is by default.

Please be aware that in “Fortran mode” `SRANLUX()` generates floating-point numbers (that is 32-bit at 32-bit systems, no comments can currently be done regarding 64-bit ones), whereas in CLHEP mode `SRANLUX()` generates and returns “double”, that is 64-bit numbers.

Also note that HEED++ was not tested explicitly in CLHEP mode of `SRANLUX()`.

- If two the above variables are initialized, HEED will need CLHEP.

  `CLHEP_INCL` – path to directory tree with include files of CLHEP. In the include directives of the source C++ code the first ”next” directory have to be called `CLHEP`, next — subject-related one. For example,
  `#include "CLHEP/random/concrete_file_names.h"` .

  If the user preferes to use the local corrected copy of random number package from CLHEP instead of CLHEP, he should not define this environment variable. In this case compilation scripts will link to the local copy.

  In older versions this variable had name `OBJECTLIB`. So in the case you used the older versions, you should correct your setup scripts. (This name migh still be appear also in some my scripts which I might forget to correct.)

  `CLHEPROOT` – directory in which the binary library of CLHEP resides.

  `CLHEPLIBNAME` – name of binary library containing CLHEP. (So, if the total file name is `libCLHEP.a` or `libCLHEP.so`, this variable has to keep the string `"CLHEP"`).

  If the variable `USE_CPP_SRANLUX` is not initialized, only system-of-units header files from CLHEP are necessary for work of HEED, but there are the local copies of these files installed together with wcpplib. In order the local copies be involbed, the “total CLHEP” entry in files
  `wcpplib/clhep_units/WSystemOfUnits.h` and
  `wcpplib/clhep_units/WPhysicalConstants.h`
  (which are included in HEED instead of CLHEP-ones) should be commented off like this:

  ```
  #include "wcpplib/clhep_units/SystemOfUnits.h"   // local copy
  //#include "CLHEP/Units/PhysicalConstants.h"   // total CLHEP
  ```

  This is currently so by default.

- `LD_LIBRARY_PATH` – path to libraries loaded dinamically. CLHEP is usually installed with such kind of libraries, so user should include their location in this variable. Syntax is like ordinary `PATH`.

- `USE_HBOOK`, `USE_OPEN_SCIENTIST`, `USE_GHIST`, `USE_HISTOG` – 4 environment variables controlling building in `*.b`-files.

  ONE AND ONLY ONE OF THEM should be defined! If you initialize two, the building scripts would not detect this, but would do something wild. If you initialize none, they will hopefully detect this as error.

  Currently packages which implements `USE_HISTOG` are not supplied. This variable is still kept in some scripts to make possible future changes. So now this option should not be used.

  Previously the same macro-definition should also be switched in file `wcpplib/graphics/control.h`. This macro controls preprocessor behaviour. Now the mentioned file does not initialize one of these macros. Instead it only checks that at least one of them is on. (But it will be satisfied also if two or more is on, which would be an error.) One of these macros is assumed to be passed to compiler through `-D` compiler option. This is done automatically in `build.b-` and `*_ts.b` scripts, which set this option accordingly to environment variables. Simultaneously they set the proper include and library options.

Previously either one of these `USE_*` options could be swithed on or none might be switched. In the latter case the program was intended to be linked with `Histog` package. But this package was and is quite old and it is not supplied with "export" version of HEED. Today the absence of one of these options is considered as error. `Histog` package is not distributed, as previously.

The choosing of one these envorinment variables leads to choosing not only way of compilation but also to including or not header files from external libraries and linking or not with external binary libraries.

As was partially explained earlier, `USE_HBOOK` requires packlib from CERNLIB to be available for link, and PAW from CERNLIB to be available for histogram browsing. `USE_OPEN_SCIENTIST` is naturally requires OpenScientist to be installed for compilation, link, and running, and possibly for histogram browsing. `USE_GHIST` requires ROOT to be available for histogram browsing, but does not require it for compilation, link, and running of HEED.

The recommended option is `USE_HBOOK`. `USE_GHIST` is also OK.

- `TEMPLATESLIB` – It might be obsolete variable for the Solaris operating system. The author did not work at Solaris during a long time. Below it explained how it was arranged there 7 years ago:
  "The compiler of Solaris stores templates in a special library. To include it at linking you are to give here its location and name."

6.1.1 For Windows and Visual Studio

The bat-files for Windows were not updated from previous version, so they might need some manual intervention of the user. Below I repeat the old text, which was also not updated.

As Windows users know, the Windows environment variables can be initialized through screen buttons (in backward translation from Russian)
`Start->System->Additional->Enviroment variables->Enviroment variables for user`
or in interpreter by command
`set var_name=var_value`.
In the following examples the second form are printed. As already was mentioned, it is assumed that the Visual Studio C++ environments are also set in the working command interpreter. This can be done by running the command shell from the Tools menu of the Visual Studio window. The examples are tested under Visual Studio C++ version 8 in Windows XP.

- `WCPPLIB` – location of the root directory of wcpplib for searching include files. The include files in wcpplib are included with prefix `wcpplib/subdirectory/...` like the following: "`#include "wcpplib/safetl/AbsArr.h"`". This allows to avoid mess between the local files and possible system files with the same names (although no such coincidences are ever noticed).

  The example of initialization:

  ```
  set WCPPLIB=''C:\Documents and Settings\user name\working directory name''
  ```

- `WCPPLIBOBJ` - location of libraries and some object files of wcpplib. Used at link of executable programs. For example:

```
set WCPPLIB=''C:\Documents and Settings\user name\working directory name\wcpplib''
```

- VSCPPCOMPOPT – additional compiler options. This variable was initially included in order to switch between the linked libraries. If your main program is being debugged, you might want to include debug information in your libraries as well. If you compile "release" version of main program, you might want to exclude the debug info from the libraries either. Anyway, my experience (incompleted so far) shows that if one use different types of debug information, the program will not be linked. Moreover, the difference in speed of calculations in debug and release configurations is huge (unlike Linux). Therefore I have included this variable in every compilation and linking script. This allows to change the debug information simultaneously everywhere (except the main Visual Studio Environment, in which the user should edit `Tools->Progect pages` system of menu manually). Example of this variable:

  ```
  set VSCPPCOMPOPT=/MTd
  ```

  for debug and

  ```
  set VSCPPCOMPOPT=/MT
  ```

  for release configuration.

6.2 For HEED++

For compilation, link and run of `heed++` programs you will need the following environment variables installed:

- HEED_PREF – location of the root directory of heed (`heed++`) for searching include files (analog of WCPPLIB). The include files in `heed++/code` are included with prefix `heed++/code/...` like the following: "`#include "heed++/code/HeedMatterDef.h"`". This variable is used as argument of `concom` command in `heed++/code/build.b`, which passes it to preprocessor, `make`, and compiler, amd also in building scripts for compilation and linking executable probrams.

  Example of initialization:

  ```
  setenv HEED_PREF $HOME
  ```

  (if subdirectory `heed++` is located just in `$HOME`)

- HEEDDIR – is used to locate the library `libheed.a` which is normally resides in ".../heed++/libheed.a" (analog of WCPPLIBOBJ). Example:

  ```
  setenv  HEEDDIR $HEED_PREF/heed++
  ```

- HEED_DATABASE – should point to location of database ".../heed++/database". Used at run-time.

7 Macro variables

There are some marco variables (macros) which may be changed in order to adapt wcpplib for particular system, compiler and envorinment. At contemporary systems there may be no necessity to change them. Some of them were not used during long time. But we will describe all of them here for completness.

7.1 Emulation of new C++ standard at old systems

The file `wcpplib/util/emul_new_stand.h`:

- `S_ENWSD` – if defined, the programs will be switched to emulation of cast operators: instead of `static_cast` as well as `dynamic_cast` it will be direct type transformations: `((type)(name))`.
- `S_NO_TYPEID` – if defined, there will be no dynamic type identification and therefore no `dynamic_cast`. Instead of `dynamic_cast static_cast` will be inserted. It works well sometimes, but with HEED it will not work. HEED needs dynamic type identification categorically. It is used in `heed++/code/HeedParticle.c` for identification of volume for which the cross section of ionization loss is defined (the total volume object should be derivative from `class EnTranfCSType`). Similarly it is used in `heed++/code/HeedDeltaElectron.c` to identify whether the δ-electron cross section is defined and whether it is "sensitive" volume, that is whether the conduction electrons are assumed to be deposited here. It is also used for similar purposes in `heed++/code/HeedPhoton.c` and in many "header" programs.
- `S_EEXCEPT` – Not used in HEED. Sign to Emulate Exceptions. If defined, they are emulated, but this happens in a package, which is not supplied.
- `S_EMUTABLE` – sign to emulate directive "mutable". If defined, they are emilated (in theory it is not good style, but some old compilers were unable to understand them, so I needed to apply this somewhere).

This file also issues directive `using namespace std`, which appears to necessary at newest systems.

7.2 Strings

The development of wcpplib was started before the standard library was firmly established, and when there were many other options and it was not clear which of them will predominate. Wcpplib does not depend on sophisticated operations with strings. The strings are used mainly for notation of various objects and files. The typical operations are initialization from characters array, convertion to characters array, initialization from integer number, and addition of strings. Such simple operations were expected to be supported by any sensible library. Therefore wcpplib was based on references to a symbolic name `String`, which was expected to be redefined somehow to a specific package in file `wcpplib/util/String.h`. Unfortunalety, STL, which dominates now, differs from sensible libraries by syntax of convertion from string to character array (notorious `.c_str()`). Therefore when strings are disappeared from CLHEP, I needed to edit many files and I included switches like this:

```
#ifdef USE_STLSTRING
  ofstream file(file_name.c_str());
#else
  ofstream file(file_name);
#endif
```

Such things still remain in many files. `USE_STLSTRING` is a macro defined in file `wcpplib/util/String.h` and meaning that std::string is invoked at the place of `String`:

```
typedef std::string String;
```

In this file also useful conversion fron `long` to `string` is defined:

```
inline String long_to_String(long n)
{ std::ostrstream s; s << n <<'\0'; return String(s.str()); }
```

7.3 Smart pointers

All the programs are based an a system of two ortogonal (in some sense) smart pointers, which are quite complicated technically (ideas are described in [6, 7] and in section 9.3). They are defined in file `wcpplib/safetl/AbsPtr.h`. There are many macro variables and their descriptions. One variable which might need adjustment is `COPY_RETURNS_COMMON_BASE`. If it is defined, it permits to bypass a problem in old GNU compilers. When it is defined, it leads to defition of some other macro variables and `class CommonBase`, from which all classes pointed by `ActivePtr` have to be derived. This all is made automatically. So at GNU gcc compilers with versions 3.2 and higher this variable can be undefined (that is "define" is commented). At older compilers it perhaps have to be defined. If it has to be defined, but it is not, the compiler should say:

```
sorry, not implemented:
adjusting pointers for covariant returns
```

The file `wcpplib/safetl/AbsPtr.h` contains much more macro-variables, but all they are assumed to be adjusted to the optimal values in the supplied version. In case of doubts see comments around them in this file.

7.4 Templated arrays

The linear array (which is frequntly called "vector" by C++-programmers nowadays) is called here `DynLinArr` (dynamic linear array) and is defined in `wcpplib/safetl/AbsArr.h` (and, of course, in ....`c`, the name `AbsArr` is translated as an abstract array). The violation of boundaries by the array index is checked at each access, if the macro `ALR_CHECK_BOUND` is defined. Of course, the check takes some time at execution, but it appears that in real physical Monte Carlo simulations the slowing down of calculations is barely noticed. Also note that at Linux it is not possible to switch this macro on in one files and off in other (at least the author was not able to do this): it will appear either switched on or off in the whole program. Therefore this variable is just permanently defined just in `wcpplib/safetl/AbsArr.h` and hence the boundaries are always checked.

The program uses also many-dimensional array `DynArr`, defined in the same files. It can have as many dimensions as necessary. Index by each dimension can run from 0 to some fixed for this dimension boundary. There is another macro name assotiated with it: `ALR_CHECK_EACH_BOUND`. Like `ALR_CHECK_BOUND` it is set permanently in `wcpplib/safetl/AbsArr.h`. But user can change the both. Actually `DynArr` is inplemented through `DynLinArr`. If `ALR_CHECK_EACH_BOUND` is not set, only the total boundaries are checked. This guarantees only the the element will be inside `DynArr`, but not necessary at the correct place: the boundary of particular index can be violated. If `ALR_CHECK_EACH_BOUND` is set, each index is checked separately, and the total boundaries are not checked.

Both `DynLinArr` and `DynArr` have functions (`acu()`) for access without check at all (independently on setting of described macros). Use them with caution.

The program uses another one-dimensional array that rises without copying the content (it can allocate elements in separated blocks, many elements per block, new elements can be put in another new block), see files `wcpplib/safetl/BlkArr.h(.c)` (translated as block array).

The program uses doubly linked lists, see files `wcpplib/safetl/AbsList.h(.c)` (abstract list). All arrays use `DynLinArr` internally as a driver and are checked, if the linear array is checked. The lists are checked always.

7.5 Streams

Two default stream notations are used thorought the program: `mcout` and `mcerr`. (These notations actually mean "my cout" and "my cerr".) The both are macro-variables defined in file `wcpplib/stream/prstream.h`. Currently the both are substituted by `cout`.

7.6 Run-time error detection

This package is designed for checking assertions and printing clear diagnostic in the case of error. It is defined in file `wcpplib/util/FunNameStack.h`. It can be partially switched on or off (the latter is provided in order to speed up the calculations, but no noticable effect can usually be detected in real programs, in which various kinds of calculations are mixed).

- `FUNNAMESTACK` – writting the function name in internal stack, when the macro call `mfunname("function name in any readable form")` is invoked. Does not make effect for permanent variant of this call `mfunnamep("function name in any readable form")`. (the permanent variant is usually used in the source text if there are assertions right in this function; the assertion requires either of them for successful compilation, when `DO_CHECKS` is defined, see below). Such calls are usually placed right after the title of the function. The phycical copying of string does not occur. Only the address of a static string is memorized in the stack. Thus there is not noticable effect for calculation speed. The stack is usually printed after triggering error by an assertion statement. Thus the user can instantly see the name of the function in which error is detected and all the functions through which this function was called, which is a precious feature. If the function was compiled with `-g` option, the program will usually print also the file name and the line number. The program prints also the condition that triggers error, the variable names involved and their values. So in most of the cases there is no need to run the program second time under debugger, to reproduce the situation, and to recover the information from the debugger. After these printing the program normally flushes the streams and stops (terminates). The program can also throw an exception, if the value of a global variable `int s_throw_exception_in_spexit` is equal to one. The assertions can also be switched on or off for speeding up with the help of the next macro.

  At multithreading environment the keeping of stack with function names is greatly complicated.

  The HEED program does not need multithreading, therefore you can use one-threading mode and ignore the following text.

  In the case of multithreading the package needs to keep separated function name stack for each thread. For this purpose the package is equipped with the methods to identify the current thread at the each call of any of its functions. At the output it prints the name stack of that thread, in which the error is detected. These possibilities are switched on by macro `USE_BOOST_MULTITHREADING`.

  Another complicated issue is the time when all these stacks must be initialized. The general rule is that stack is initialized at the moment, when any of functions detects the new thread. This is done through a singleton model known in literature as "GoF" approach. Correspondingly the header list of stacks is never deleted, because it is impossible to determine the moment, when all the registered functions are passed and no one will ever be called at the program termination. Therefore memory checkers could find some non-freed memory after the program termination, but this is inavoidable.

  The general advice: never use multithreading unless you have a lot of free time, which you urgently need to make busy.

  Also note that in the source of this package there are come controls for an original memory checker `clean_new`, but it is not supplied, and these switches should be ignored.

- `DO_CHECKS` – if defined, check the assertions of the type `check_econd(...)` (condition for triggering error with further termination), or `check_wcond(...)` (condition for issuing warning without program termination), or exclude them from the source text otherwise (this however does not make noticeable effect on speed).

7.7 Inline or not inline?

In some sub-directories the functions which can possibly be inline (if they are very small and simple and without loops), were declared by such a way, that they can be switched and appear as either inline or non-inline. First of all, the word `inline` was replaced in the source text by the word "`wl_inline`". This latter is a macro defined in the file `wcpplib/util/inlinec.h` as either empty macro or "`#define wl_inline inline`". The total file now loook by the following way:

```
//#define WCPPLIB_NO_INLINE

#ifdef WCPPLIB_NO_INLINE
#undef WCPPLIB_INLINE
#define wl_inline
#else
#define WCPPLIB_INLINE
#define wl_inline inline
#endif
```

So with the commented first line (as it is now) the empty macro `WCPPLIB_INLINE` is defined and the macro `wl_inline` means `inline`. The inlined functions are placed in files `*.ic`, which are not directly compiled. In the ends of the header files there are such sentences:

```
#ifdef WCPPLIB_INLINE
#include "wcpplib/subdirectory/filename.ic"
#endif
```

So in the case of inline, the files are added at the end of the header files. In source files there are opposite sentences:

```
#ifndef WCPPLIB_INLINE
#include "util/FunNameStack.ic"
#endif
```

and these "inlined" functions get non-inlined. In theory, the inlining leads to faster execution and more memory expense, where as non-inlining causes the opposite effect. But no considerable differenses were ever registered. So I am always working with inlining.

7.8 Link to Fortran

Although the directory `cfortran` is included in the package, the known package named usually "`cfortran.h`" (and seem to be dirtributed now in CERN) is not used. At the time when I adjusted this interface, there were some problems with "`cfortran.h`", such as a lot of warnings and something else, which I don't remember. I was tired from them and made own interface for just a few Fortran subroutines that the C++ program needs to call. This is just several HBOOK subroutines and also several random number generators. The file `wcpplib/cfortran/ftypes.h` defines the data types for C, equivalent to standard types in Fortran. The C-types can differ at 64-bit computers. Therefore this file needs attention, if such computer is involved. This file is very simple, and it can be totally reproduced here:

```
#ifndef FTYPES_H
#define FTYPES_H

typedef int fint;       // C++ int equivalent to fortran integer
typedef float ffloat;   // C++ float equivalent to fortran real*4
typedef double fdouble;   // C++ float equivalent to fortran real*8

#endif
```

The HBOOK subroutine headers necessary for C++ are defined in file `wcpplib/cfortran/hbookm.h`. The general scheme accepted for interface is simple. First the Fortran function header is declared in terms of C++ with adjusted number of underscores, like this:

```
#ifdef FORT_UNDERSCORES_02
extern "C" void hfill__(fint*, ffloat*, ffloat*, ffloat*);
#elif FORT_UNDERSCORES_01
extern "C" void hfill_(fint*, ffloat*, ffloat*, ffloat*);
#elif FORT_UNDERSCORES_11
extern "C" void _hfill_(fint*, ffloat*, ffloat*, ffloat*);
#else
extern "C" void hfill(fint*, ffloat*, ffloat*, ffloat*);
#endif
```

The Fortran compilers are known to add underscores to function names. The adjustment used in this fragment is apparently redundant: all computers I am working at recently add one underscore after the name. This means that the macro `FORT_UNDERSCORES_01` should be defined at compilation of the C++ functions which calls `hfill`. The call `hfill` should also appear with one underscore or with switch. The latter is in fact inconvenient. Therefore one else step is made: definition of an interface C++ function:

```
inline void HFILL( fint num, ffloat p1, ffloat p2, ffloat koef)
{
#ifdef FORT_UNDERSCORES_02
  hfill__(
#elif  FORT_UNDERSCORES_01
  hfill_(
#elif  FORT_UNDERSCORES_11
  _hfill_(
#else
  hfill(
#endif
        &num, &p1, &p2, &koef);
}
```

This interface function also replaces addresses by ordinary variables. When the program needs to call `hfill`, really it has to call `HFILL`. The same rules are followied for any other Fortran functions: adjustment of underscores and interface function by upper case name. The macro `FORT_UNDERSCORES_01` is passed to compiler in those build.b scripts, when the Fortran is involved, e.g. in `wcpplib/graphics/build.b` and in `wcpplib/random/build.b`.

7.9 Graphics

As already was described in "Environment variables" section, the file `wcpplib/graphics/control.h` contains three commented off lines, one of which may be uncommented, like this:

```
#define USE_HBOOK
//#define USE_OPEN_SCIENTIST
//#define USE_GHIST
```

This should be synchronized with environment variabls.

7.10 Size of cross section object

Several important macros controlling the size of cros section object are described in section 17.4:

```
#define EXCLUDE_A_VALUES  // exclude absorption values
#define EXCLUDE_VAL_FADDA  // exclude unecessary for MC values
#define EXCLUDE_MEAN  // exclude calculations of means, which are
// unnecessary for MC

#define CLEAR_ARRAYS
```

8 Compilation scripts

The program is too large to be re-compiled completely each time a single change is made. This would not be convenient if some program development is going on. Therefore the selective compilation of new, changed or affected files is actual.

In UNIX the selective compilation is usually made by the "`make`" programm, which interprete make-files. The latter describe file dependencies and commands to execute. However, the program HEED seems also to be too large for writting its make-files manually.

More or less automatic generation of make files is possible through the Autoconf-automake system. But using this system assumes getting skilled in about 360 pages of their non-trivial manuals, which is too large for author's busy schedule. In addition, its rules and scripts do not look transparent enough, and in general it seems to be designed rather for system-related works, than for pure calculations. For latter it seems to be somewhat redundant.

The acceptable alternative solution, which permits us to write very small and clear compilation scripts, is provided by the possibility to generate the dependence instructions automatically with `g++` compilers.

Unfortunately no such possibility is found for Windows Visual Studio compiler. There are some " `nmake`" utility. The author did not study it in detail so far, but it seems to have the same problems as UNIX' `make`. Therefore for Windows compiler only the scripts which perform the full compilation of each sub-package of `wcpplib`, as well as the script, which perform the full compilation of the total `wcpplib` (selected subpackages; this script simply calls each sub-package script one by one) are provided.

But in Linux (in which we can use of mentioned automatic generation of dependence instructions) it is possible to create a very simple program (currently 265 lines in C++ only, including reach comments!) working as a frontend of both the compiler and the "make' program and allowing one to write the compilation command for selective compilation for all necessary files (new, changed, or affected by new or changed ones) in a single line! Calling such a program one has just to provide as its arguments the compilation command with all its arguments and supply a list of source files (`*.c` files, `*.h` are not mentioned at all, which greatly simplifies the work, and each `*.c` is mentioned only once, which also simplifies everything; however, if one wants to compile a library, one need to mention equivalent object file two times more, see below, but this is tolerable complication). The program is called "`concom`", which is translated as "connect commands" (may be not very felicitous name, but it's lazy to change it in very many places, where it is already imprinted).

Each compilation script typically has call of this program. Another couple of lines of typical compilation script is the instruction for the ordinary make program to create library from the object files, of course, in the cases, when the library is necessary. And one else attribute of the compilation scripts is running the "make" itself. Let us consider a simple real example, but first we describe the syntax of `concom`.

The `concom` program takes the following arguments:

1 The name of compilation command (e.g. `$CPPCOMPILED`). Note that this name can also contain some compiler options, normally general for the full compiled system.

2–n where $n+1$ is the argument number (index) with one of the following extensions: `.c`, `.cpp`, `.CC`, `.f`, `.f77`, `.for`, `.F`. All the arguments from 2-nd to n-th are interpreted as an additional compilation options and are passed to preprocessor and compiler. Obviously the program simply scans arguments one by one and stops at the first argument with one of the listed extensions.

n+1... The source files to compile. All they should have one of the mentioned extensions. At least one source file should be given. But normally all source files to be compiled with identical options are given in one line.

The program `concom` processes the files one by one, for each it generates with the help of preprocessor the dependence instruction in `make` format, which is output to default stream. Its rediretion to makefile is done manually in `build.b`, see the example below.

So, the typical building script, creating library libmatter.a and also writing in a certain directory a separated file `GasLib.o` (the file names are related to HEED) is given in the following real example:

```
#! /usr/bin/env tcsh
rm makefile
cat > makefile <<EOF1
#This makefile is created by build.b
EOF1
#wc particle_def.h eparticle.h lparticle.h particle_def.c eparticle.c lparticle.
c
cat >> makefile <<EOF2
$WCPPLIBOBJ/libmatter.a: AtomDef.o MoleculeDef.o MatterDef.o GasDef.o
        $ARCOMMAND $WCPPLIBOBJ/libmatter.a AtomDef.o MoleculeDef.o MatterDef.o G
asDef.o
EOF2
cat >> makefile <<EOF3
$WCPPLIBOBJ/GasLib.o: GasLib.o
        cp  GasLib.o $WCPPLIBOBJ/GasLib.o
EOF3

concom $CPPCOMPILED  -I$WCPPLIB  -DFUNNAMESTACK -DDO_CHECKS \
AtomDef.c MoleculeDef.c MatterDef.c GasDef.c GasLib.c >> makefile
if ($status != 0) then
    echo "Processing of source files by concom failed"
    exit
endif


make -r $WCPPLIBOBJ/libmatter.a
make -r $WCPPLIBOBJ/GasLib.o
```

These 27 lines are converted by `concom` into a 150-line makefile, starting as follows (we will not reproduce it completely):

```
#This makefile is created by build.b
/home/ismirnov/s06/wcpplib/libmatter.a: AtomDef.o MoleculeDef.o MatterDef.o GasDef.o
ar -rs /home/ismirnov/s06/wcpplib/libmatter.a AtomDef.o MoleculeDef.o MatterDef.o GasDef.o
/home/ismirnov/s06/wcpplib/GasLib.o: GasLib.o
cp  GasLib.o /home/ismirnov/s06/wcpplib/GasLib.o
AtomDef.o: AtomDef.c /home/ismirnov/s06/wcpplib/matter/AtomDef.h \
  /home/ismirnov/s06/wcpplib/util/String.h \
  /home/ismirnov/s06/wcpplib/safetl/AbsArr.h \
  /home/ismirnov/s06/wcpplib/stream/prstream.h \
  /home/ismirnov/s06/wcpplib/util/FunNameStack.h \
  /home/ismirnov/s06/wcpplib/stream/prstream.h \
  /home/ismirnov/s06/wcpplib/util/inlinec.h \
  /home/ismirnov/s06/wcpplib/util/FunNameStack.ic \
  /home/ismirnov/s06/wcpplib/util/emul_new_stand.h \
  /home/ismirnov/s06/wcpplib/util/String.h \
  /home/ismirnov/s06/wcpplib/safetl/AbsPtr.h \
  /home/ismirnov/s06/wcpplib/math/minmax.h \
  /home/ismirnov/s06/wcpplib/stream/prstream.h \
  /home/ismirnov/s06/wcpplib/util/FunNameStack.h \
  /home/ismirnov/s06/wcpplib/stream/definp.h \
  /home/ismirnov/s06/wcpplib/stream/prstream.h \
  /home/ismirnov/s06/wcpplib/util/String.h \
```

```
  /home/ismirnov/s06/wcpplib/util/FunNameStack.h \
  /home/ismirnov/s06/wcpplib/safetl/AbsArr.h \
  /home/ismirnov/s06/wcpplib/safetl/AbsList.h \
  /home/ismirnov/s06/wcpplib/clhep_units/WPhysicalConstants.h \
  /home/ismirnov/s06/wcpplib/clhep_units/PhysicalConstants.h \
  /home/ismirnov/s06/wcpplib/clhep_units/WSystemOfUnits.h \
  /home/ismirnov/s06/wcpplib/clhep_units/SystemOfUnits.h \
  /home/ismirnov/s06/wcpplib/stream/prstream.h \
  /home/ismirnov/s06/wcpplib/util/FunNameStack.h
g++ -g -c -I/home/ismirnov/s06 -DFUNNAMESTACK -DDO_CHECKS AtomDef.c

MoleculeDef.o: MoleculeDef.c \
...
```

Note that directly in the file `AtomDef.c` only the following include statements are given:

```
#include <iomanip.h>
#include "wcpplib/matter/AtomDef.h"
#include "wcpplib/clhep_units/WPhysicalConstants.h"
#include "wcpplib/stream/prstream.h"
#include "wcpplib/util/FunNameStack.h"
```

Comparing this list with the dependence statement from the makefile one has to conclude that not only the explicitly included files are checked, but also those "included from included". Thus, all the hierarcy is checked, except the system files (like `<iomanip.h>`). The latter is perhaps unnecessary to check. If system is changed, it is wise to recompile everything.

Surely it is much easily to write 27 lines than to write 150 lines! Such a system is also much more reliably with respect to manual typing dependencies. If you miss some dependency, the system does not detect this error and you will eventually obtain wrong outdated program. If the makefile is gererated automatically, such an error is not possible. You can also change the program, adding new dependencies at any moments, and they will be immediately be taken into account.

Thus, the combination of the `concom` program and the `build.b` script allows user to change freely any of the source or included files or to add new included files into the source files. This system detects it since it remakes "makefile" each time the `build.b` command is run. But the system does not check the modification of `build.b` itself. It cannot also detect in general case the change of environment variables on which it could depend. If it or environments are modified, I usually remove all "object files" from relevant directory by "`rm *.o`", or run `wcpplib/clearbin.b` to clean everything and then run `build.b` again.

All the compilation of files in particular directories is done by scripts `build.b` or `buildhd.b`. The script `wcpplib/buildhd.b` compiles all wcpp library (in the described sense, that is it creates makefiles and runs "`make`"). It simply jumps to each necessary sub-directory and run its local `build.b` or `buildhd.b`. The script `heed++/code/build.b` compiles all heed class library. It however does not jump to `wcpplib` and does not check it. So the sequense for linking any test (or user) program is running `wcpplib/buildhd.b`, running `heed++/code/build.b`, and running `heed++/code/NameOfTestProgram_ts.b`.

We repeat that the program `concom` is placed in `wcpplib/install` directory. It should be compiled prior to `wcpplib` and `heed++` by a regular script `concom.b` and put manyally into a directory registered in `PATH` environment variable. Of course, `concom` does not depend on the rest of wcpplib. For Windows all this is not valid. Just run `wcpplib/buildst_vs.bat`, and that's all.

9 Miscellaneous

9.1 The quantities, indexes, and arrays

For the sake of clarity and consistency the author follows a rule that the numbers of something are started from `q` or `q_`. Most frequantly such variables are sizes of arrays or the number of valid elements in arrays (which are not always the same: the second can be smaller than the first). The character "q" is very convenient to use in order to distinguish the *quantities* from the *indexes*, the number of something from the something's number (at least for a non-native English speaker). The indexes and the something's numbers are usually started from characters `n,i,k,l,n,m`.

Then, if `q_something`, i.e. `q_point` or `q_particle`, is the quantity (the number) of points or particles, the user could expect to find near to its declaration the declaration of arrays of these items, named `point` or `particle` respectively. Note that both the name of array and the meaningfull part of the name of its size are in *singular* mode: `point` and `q_point`, not `points` and `q_points`. This is justified by the following considerations. When you address an element of such an array, you write, for example, `point[n]` or `particle[i]`. Having these inscriptions looking like `points[n]` or `particles[i]` would definitely lead to confusion, whether they address and mean a single point and a single particle or many of them allocated in these particular elements. So it is adopted that if the array name in plural, this means that its elements mean plural things themselves, that is they are usually arrays themselves. And much more frequent case of array with single elements is denoted by a single noun. Finally, in order to provide the notation of quantity to be as similar as possible to the notation of array and to be easily identified in the program as it, it is reasonable to leave it in the same singular mode, i.e. `q_point`.

All templated arrays, defined in wcpplib (in the directory `wcpplib/safetl`), use the standard C++ way of indexing, that is the first index is 0. For example, the size of the array `DynLinArr< double > ar` is obtained by `long q = ar.get_qel()` (get quantity of elements), or changed by `ar.put_qel(q)`. Then the allowed indexes are from `0` to `q-1`:
`double first_el = ar[0]`.

The array of this type can be an element of another array, which allows to build many-dimensional arrays with variable sizes (size by the next dimension can be initialized separately for each index by previous dimension and therefore it can be different).

If one wants to define an array with the constant in this sense sizes, that is a rectangular array, one should use another template class called `DynArr`. The number of dimensions of `DynArr` can be varied, but all dimensions are "fixed" along other indexes. The indexes and the sizes of `DynArr` are in general expressed by a variable of type `DynLinArr< long >`, rather than by a single `long`, as for `DynLinArr`. It is therefore not surprising that both in the interface and in the implementation of `DynArr` one can find many appearances of `DynLinArr< long >`. In particular, the normal fast access to elements of `DynArr` is provided by functions `.ac(const DynLinArr<long>& ind)`, rather than by `[][]....` The latter are also provided in the last version, but this is advised to be used only when sintax (of already existed or maximally flexible programs) assumes these brackets `[][]....` Note that in the case of `DynArr` this type of de-indexing works somewhat slowly due to some inavoidable problems in its implementation. For two and three-dimension cases special, more simple and very fast interface is also provided, e.g. `.ac(long i1, long i2, long i3)`. The first indexes are always 0.

In contrast with STL vectors, the both arrays, `DynLinArr` and `DynArr`, do not have "hidden" elements or unused memory reserved after the last elements. If the arrays have to be increased, a new memory block is allocated and the existing elements are copyed by assignment of each element. This is sometimes inconvenient by efficiency and due to loosing pointers addressed the array elements.

The array that can rise without copying content is `BlkArr`, the blocked array. The elements are stored in a sequence of blocks. It represents a trade off between the efficiency of rise, the efficiency of access by indexes, and the system memory expense. It is assumed that at the rise by one element each next block will have larger size than the previous one (twice as large). The user can also initialize the first block with reasonably large for his problem size. This allows to keep the total number of blocks reasonably small, while not occuping too much system memory. But the physical and logical sizes of this array are different values. The logical size is obtained or is put by similar `get_qel()` and `put_qel()` functions. The physical size is determined automatically except the initialization of the first block, which can be initialized by `void BlkArr::allocate_block(long fqel)`. The indexes are counted from zero as well.

At copying all arrays copy all elements by assignment. At deletion all elements are deleted as well.

There is also a list class `AbsList` which works by similar way: copying by assignment and deletion. We don't use interators, but iterate through the lists by a classic manner, like this:

```
class Element
{
  ...
};
ostream& operator<<(ostream& file, const Element& f) {...}
Element el1, el2;
AbsList< Element > a_list;

// filling the list

a_list.append(el1);  // copies el1 to list
a_list.prepend(el2); // copies el2 to list and puts it before el1

AbsListNode< Element >* pointer_to_node = NULL;
while( (pointer_to_node = a_list.get_next_node(pointer_to_node)) != NULL)
{
  // do something with element, for example flush it to stream
  mcout << pointer_to_node->el;
}
```

All arrays and the list can be elements of each other.

9.2 Private and public data

The utility classes of this program like containers, smart pointers etc. encapsulate (hide), as usually, by `private` statement the data which are not supposed to be read or changed. In the contrary, many "application" classes in this program, espesially in `heed++/code` directory have many public data members or all public data members. But this should not be interpreted as a permision to change them after the object initialization. Many objects would be broken after such at action. The most of the application classes acquire all their values right at the initialization and are not expected to be changed manually after it. The user should read comments, study the source code, or act by analogy.

9.3 The pointers

The plain pointers are not used in the classes as class members. Otherwise they could create complete mess. When an object needs to refere at another object, I first decide whether that other object belongs to the first.

The notion "to belong", that is the "ownership", is usually coherent with the concept of object integrity, with an exclusive reference (one and only one referee per each refered object), and with the lifetime of refered object not exceeding the lifetime of the referee. The opposite notion, that is the "external" or "independent" object, is opposite in all these senses; it is assotiated with possibly many objects refering to that one and with the lifetime possibly exceeding the lifetime of referees and independent on them.

Frequently one else category of relations is undermined in the so-called "shared" smart pointers. Sometimes the term "smart pointer" is understood just as a notation of this particular and not very consistent category. For the first glance the "shared" smart pointer expresses a shared-ownership relation. Each object can have many pointers to it, or many "owners". The lifetime of such object could exceed the lifetime of a single pointer to it, but cannot exceed the lifetime of all pointers to it. By the opinion of the author this concept is a mixture of different cases and is rather confusing. For example, copying all the pointers to a certain object together to another place, which might naturally express the creation of another full header object with new instances of the owned part, would not actually trigger cloning the part. So the shared pointer does not indicate any form of "ownership", integrity, or independency (the integrity and independency are basic features by which we distinguish objects in our mind). At least all functionality of HEED can be achieved without "shared" pointers and they are not used here.

Of course, the particular case for expression of the owneship is putting the owned object as a class member of the owner. But this is not always possible since in order to do this we have to know precisely the type of the owned object and its size, which is not always the fact.

The problem of the variable size at fixed type is solved with the help of dynamic arrays.

The problem of a single object of given or derived type can be solved with the help of putting pointer in a class as a class member. But then we will have a terrible mess at initialization, access, copying, and deletion of the owner. The experience suggests that such structures can never be encoded and applied without errors. Therefore we use in these cases well debugged and well defended (fool-proved) active pointer or `ActivePtr`, a templated smart pointer.

The opposite case also need some defence and we use for it a passive pointer, `PassivePtr`, see `wcpplib/safetl/AbsPtr.h`.

The both pointers check the existance of the object at each access by `*`, `->`, and `.get()` and trigger an error if it does not exist. There is also function for dereferring with verification `.getver()`, which just returns `NULL`, if the object does not exist.

The active pointer clones the object at the copying the owner. This is implemented with some tricks, which are decribed in `wcpplib/safetl/AbsPtr.h`.

The passive pointer mainly intended to check that the object exists, and it also has many auxiliary functions, see `wcpplib/safetl/AbsPtr.h` as well.

The active pointer can refere only the objects which are separated entities in dynamic memory. The passive pointer can refere an object allocated by any method (including static and automatic objects or their parts). The object pointed by passive pointer can be removed (independantly on pointer), which normally triggers clearing the pointer and detecting error, if access through it is again requested.

The both pointers are "intrusive", since they assume the referred object is prepared to be pointed, that is it has some predefined components. It is unfortunate, but not too inconvenient.

Inside the function bodies the use of plain pointers is allowed. They are usually obtained by dereferring the smart pointers at the beginning of the functions and aimed for acceleration of access. In such cases it is, of course, assumed that the object does not disappear till the function end.

This concept was first described in the appendix of the preprint [6] (although with different notations of the types of the pointers). One reservation prompted by experience should be made. The implementation of the statement from [6]: "The addressed object should know the addresses of all protected (i.e. passive by the new notations) pointers to it and clear them at its deletion" could lead to too large expenses of computer time in the case of very many pointers referring one object. This can be avoided totally by the introduction of an intermediate reference counter object. Then all functionality (more precisely all significant functionality) is preserved after this re-design. This is how it is done now. More complete (but still not full) description of this methodology with a lot of references to related works appeared in SIGPLAN notices [7].

9.4 New and delete

The operator `new` is used in the application programs only at the first initialization of the object right before it is passed for control of active pointer.

The operator `delete` is practically never used (except for destructors of active pointers, arrays and list). All allocated memory is freed automatically with the help of the active pointers, templeted arrays and lists.

9.5 Output streams

Looking in the program a user can wonder what sentences like this mean:

```
ostream & operator << (ostream & file, const AtomMixDef& f)
{
  mfunname("ostream & operator << (ostream & file, const AtomMixDef& f)");
  Ifile<<"AtomMixDef\n";
  indn.n+=2;
  Ifile<<"Z_mean()="<<setw(3)<<f.Z_mean()
       <<" A_mean()/(gram/mole)="<<f.A_mean()/(gram/mole)<<'\n';
  ...
  Iprintn(file, f.qatom());
  long n;
  indn.n+=2;
  for( n=0; n<f.qatom(); n++)
  {
    Ifile<<"n="<<n<<" atom(n)->notation="<<f.atom(n)->notation()<<'\n';
    indn.n+=2;
    Ifile<<" weight_quan(n)="<<f.weight_quan(n)
         <<" weight_mass(n)="<<f.weight_mass(n)<<'\n';
    indn.n-=2;
  }
  indn.n-=2;
  indn.n-=2;
  return file;
}
```

Here this will be explained briefly. But first let us look what in gives in output stream:

```
AtomMixDef
  Z_mean()=12.2857 A_mean()/(gram/mole)=26.8245
  ...
  f.qatom()=3
    n=0 atom(n)->notation=Ar
       weight_quan(n)=0.642857 weight_mass(n)=0.957366
    n=1 atom(n)->notation=C
       weight_quan(n)=0.0714286 weight_mass(n)=0.031983
    n=2 atom(n)->notation=H
       weight_quan(n)=0.285714 weight_mass(n)=0.0106512
continuation would be here
```

First of all we see that there is a very convenient indentation, which distinguishes the beginning of objects and internal arrays or lists. `AtomMixDef` is notation of a class. This notation is already printed with indentation depending on the number of previous indentations. The indentation is provided because of the first "I" in front of "`file`": `Ifile<<"AtomMixDef\n";` "`Ifile`" is a marco which is defined in the file `wcpplib/stream/prstream.h`:

```
#define Ifile file<<indn
```

Similar macro is defined for mcout:

```
#define Imcout mcout<<indn
```

The notation indn is an object of a special small class, defined in the same file:

```
class indentation
{public:
  int n;     // current number of blanks to print
  int s_not;
  indentation(void){ n=0; }
};

extern indentation indn;

inline ostream& operator<<(ostream& file, indentation& ind)
{
  int n;
  if(ind.s_not==1)
    ind.s_not=0;
  else
    for(n=0; n<ind.n; n++) file<<' ';
  return file;
}
```

Thus, the sencence `file<<indn` means just output of `indn.n` blanks. The sentence `Ifile<<"AtomMixDef\n"` means actually `file<<indn.n<<"AtomMixDef\n"`, and it is not surprising that it provides indentation. The similar effect is provided by `Imcout<<"AtomMixDef\n"`. After printing the class notation one can want to increase the indentation by, for example, 2 blanks in order to express that the following is internals of this class. This is done by command `indn.n+=2` in the listing above.

In the sentence like `Ifile<<"Z_mean()="<<setw(3)<<f.Z_mean()` the notation of the variable `Z_mean()` is repeated. This is rather boring. To make such commands shorter we might define another macro like

```
#define Iprint( file , name ) \
file<<indn<<#name<<"="<<noindent<<name<<yesindent;
```

This macro is actually defined in the same file `wcpplib/stream/prstream.h` The functions

```
ostream& noindent(ostream& f);
ostream& yesindent(ostream& f);
```

are defined in the file `wcpplib/stream/prstream.c`:

```
indentation indn;

ostream& noindent(ostream& f)
{
  indn.s_not=1;
  return f;
}
ostream& yesindent(ostream& f)
{
  indn.s_not=0;
  return f;
}
```

Now we see, that this construction allows us to print the name of the object with indentation, to print the assignment character `=`, to print the value or a dump of the object *without* indentation of its first line (which otherwise would be just unnecessary blanks between `=` and the following content. The macro `Iprint` does not add the next line '`\n`' character at the end of the last line. It is good for printing the complicated objects, which usually do this themselves. For a single numbers it is not good. Therefore there is another macro `Iprintn`, which does this.

There are a number of other similar macros. For example, to print two numbers by one command in one line, we can use

```
#define Iprint2n( file , name1, name2 ) \
file<<indn<<#name1<<"="<<noindent<<name1 \
<<", "<<#name2<<"="<<noindent<<name2<<'\n'<<yesindent;
```

and so on.

9.6 Run-time error detection

In section 7.6 we already described briefly this package. Let us now consider it in more details.

The purpose of this package is detection of run-time logical errors and reporting about them. The term "run-time logical error" means here not things like segmentation fault but typically "a variable is out of bounds". There may be more complicated conditions with two, three variables etc. We have three main problems here. First, it needs to detect the error in the program by a way that does not reduce strongly its speed. Secondly, it needs to report about it by a way that minimizes the further debug effort. Information should be exhausting both about the place of this error and its essense. Thirdly, it needs to take consistent further actions, typically to terminate the program or to issue exception in the case of significant error, or simply to continue in the case of warning.

As example of importance of these tasks, let us consider the diagnistics recently occured in one of the author's programs:

```
 ***** ERROR in HBOOK2 : YMIN.GE.YMAX : ID=       4
 ***** ERROR in HPAK : Unknown histogram : ID=       4
```

Obviously, these messages are sent by the HBOOK package (to which the author's program is linked). We see that the error is initially detected in something called "HBOOK2". It is then not obvious, what is this. In many cases, such as the use of C++ interface to the HBOOK package, when one does not deal directly with HBOOK functions, one may not remember that "HBOOK2" (as well as "HPAK") is the name of a subroutine. This nevertheless can be guessed. Then, this function can be called from thousand places in source code, and one does not see from this message from which this particular erroneous call is made. Next, this user-written function can itself be called from many different places and it is also difficult to determine fron which. The remark "`ID=       4` " indicates that this histogram has number 4 (the program was very small), but if this number is generated dinamically (as in the case of the C++ interface), one have to analyse the program flow to determine the erroneous call. Next, the inscription `YMIN.GE.YMAX` indicates that either `YMIN` is wrong, or `YMAX`, or both, but one cannot determine this from this message. To find an error it needs either to analyse the source code or to add debug output before the call of HBOOK2. Why we do not insert a more detailed output directly to the functions like HBOOK2? Of course, we cannot modify library subroutines suh as HBOOK2, so here we have to deal with it as is, but we can at least inbuild more complete debug utilities in our new programs. Finally, the last line of the lising above indicates that another subroutine from HBOOK has detected, that the histogram was not initialized. None of these errors triggered the program termination, but this can be determined only from the continuation of the listing and from the other results. This is also inconvenient.

Let us now pass to C++. Let us assume that we need to guarantee or just to check that a variable does not exceed some limit. Then the ordinary way to express this in a program would be:

```
#ifdef DO_CHECKS   // \textsf{to bypass unnecessary checks after debug}
if( variable > limit )
{
  mcerr<<"error in function ...\n"
       <<"File is "<<__FILE__<<" , line number is "<<__LINE__<<'\n'
       <<" variable > limit, variable=" << variable << " limit = "<<limit
       <<'\n';
  abort();
```

```
}
#endif
```

In practice such a style is not satisfactory by two reasons:

- Such rudimentary checks are too long for printing and distracting at reading the prgram text. The essence of this expression is forbidding the case when "`variable > limit`" , which is not seen instantly in this code. The inbuild `assert()` statement looks shorter and more clear, but it does not print the values of the compared variables. In most of the cases this means that the programmer should either edit the program, include printing in its source, compile and link it, then run it and reproduce the same situation; or he should run the program under debugger and reproduce the same situation as well. If this situation appears after, say, a week of calculations, the situation gets not too funny.
- Printing the function name, the file name, and line number is not sufficient to localize the error in those cases when the function with error is called from a long sequense of other functions and the problem occurs in them.

The problem of the error localization is solved with the help of printing a stack of names of functions currently involved. The function name (or any other name) can be put in the stack by macro

```
mfunname("name to put in stack");
```

This macro is replaced by very simple text:

```
#define mfunnamep(string) static char* FunNameIIII=string;  \
FunNameWatch funnw(FunNameIIII)
```

The name `FunNameIIII` is choosen just as unusual name which can scarcely be met in programs. The initialization of object of type `FunNameWatch` triggers inclusion of the pointer to string into the stack. The destruction of such object triggers removal of the corresponding pointer to string from the stack. This does not take any significant time, since the character array itself is not copied. This does not work however in inlined functions. But we would not use this in very fast inlined functions anyway. For them more quick methods are applied. For many geometrical functions older and faster method is used, see sect. 13.

So if you write a function

```
void my_function(void)
{
  mfunname("void my_function(void)");
  // at this moment the function name is remembered
  ...
}
```

the program will remember its name while this function is involved. The macro `mfunname` is switched on only if the macro `FUNNAMESTACK` is defined at compilation. The macro `mfunnamep` is permanent.

This stack is printed by macro `stackprt(stream)` which is defined in the file `FunNameStack.h` as

```
#define stackprt(stream)  stream<<FunNameStack::instance()
```

This means that the class `FunNameStack`, which is keeping the stack, is a "singleton" class. It is initialized in a single exemplar at the first call by a static function `instance()`

```
class FunNameStack
{
public:
  static FunNameStack& instance()
```

```
  {
    static FunNameStack inst;
    return inst;
  }
  ...
};
```

but it can be copied into another exemplar in order to pass the location of the error through the exception treatment:

```
class FunNameStack
{
public:
  ...
  FunNameStack( const FunNameStack& f);
  FunNameStack& operator=(const FunNameStack& f);
  ...
};
```

This class has two data members which can have it inactive, or make him printing names of functions at entring/exiting:

```
class FunNameStack
{
public:
  ...
  int s_act;          // 1 - active, 0 - not active
                      //( to switch without recompilation)
public:
  int s_print;        // works only is s_act=1;
                      // 0 - no print
                      // 1 - at each entry print the name of current function
                      // 2 - at each entry and each exit
                      //    print the name of current function
                      // 3 - at each entry print all stack
                      // 4 - at each entry and exit print all stack
  void set_parameters(int fs_act=1, int fs_print=0);
};
```

If user want to change the default parameters before invoking main, he can initialize a global object with any name of a special class:

```
class FunNameStack_Assist
{
public:
  FunNameStack_Assist(int fs_act=1, int fs_print=0)
  {
    FunNameStack::instance().set_parameters(fs_act=1, fs_print);
  }
};
```

With this system the problem of brief recording of error condition is solved like this:,

```
check_econd12( variable , > , limit , mcerr);
```

This means “checking error condition”, the first number after “`econd`” means that there is one condition, and the second number means that both variables have to be printed in the case of error. This macro is defined in the file `FunNameStack.h`, and looks like this:

```
#define check_econd12(a, sign, b, stream) if(a  sign  b) \
{ funnw.ehdr(stream); stream<<'\n'<<#a<<#sign<<#b<<'\n'; \
stream<<#a<<'='<<(a)<<' '<<#b<<'='<<(b)<<'\n'; spexit(stream); }
```

The call `funnw.ehdr(stream)` prints "error notification" and the name of the current function. The macro `spexit(stream)` looks like

```
#define spexit(stream) {stackprt(stream); \
stream<<"File is "<<__FILE__<<" , line number is "<<__LINE__<<'\n'; \
spexit_action(stream); }
```

We see here `stackprt(stream)`, which outputs the stack of the functions currently involved. Then this macro further localizes the error by printing the file name and the line number. But both are available only of the program is compiled with `-g` option. Finnally, the function `spexit_action`, called at the end of `spexit`, is a very simple function, whose body is defined in the file `FunNameStack.c`:

```
void spexit_action(ostream& file)
{
  file<<"spexit_action: the streams will be now flushed\n";
  file.flush(); mcout.flush(); mcerr.flush();
  if(s_throw_exception_in_spexit != 1)
  {
    file<<"spexit_action: the abort function is called\n";
    abort();
  }
  else
  {
    file<<"spexit_action: an exception is now called\n";
    throw ExcFromSpexit();
  }
}
```

With these tools the error condition "`i < 0`" would be written in a program by

```
check_econd12( i , < , 0 , mcerr);
```

It is not possible to avoid these commas. In the case of `i=-1`, the program should write

```
myfun10(int i): ERROR:

i<0
i=-1 0=0
FunNameStack: s_init=1 qname=3
  0  void main(void)
  1  myfun3(int i)
  2  myfun10(int i)
File is test.c , line number is 39
spexit_action: the streams will be now flushed
spexit_action: the abort function is called
Abort (core dumped)
```

This looks nice except of the sentence "`0=0`", which is rather strange (although perfectly correct). Such things would be distracting in output listing, and experience suggests that it is reasonable to avoid them. Unfortunately, it is not easy to do in standard C++. The only feasible solution is to multiply the `check_econd` macros. For given case it needs to have macro

```
#define check_econd11(a, signb, stream) if(a  signb) \
{ stream<<"ERROR:\n"; stream<<'\n'<<#a<<#signb<<'\n'; \
stream<<#a<<'='<<(a)<<'\n'; spexit(stream); }
```

with one condition and one value to print. Its call will be written like this:

```
check_econd11( i , <  0 , mcerr);
```

and output will be

```
myfun10(int i): ERROR:

i<0
i=-1
FunNameStack: s_init=1 qname=3
  0  void main(void)
  1  myfun3(int i)
  2  myfun10(int i)
File is test.c , line number is 39
spexit_action: the streams will be now flushed
spexit_action: the abort function is called
Abort (core dumped)
```

The condition `i < 0 || i > 10` would be written as

```
check_econd21( i , <  0 || , > 10 , mcerr);
```

The condition `i < j1 || i > j2` (which requires printing `i`, `j1`, `j2`) would be written as

```
check_econd23( i , < , j1 , || ,  > j2 , mcerr);
```

The condition `i1 < j1 || i2 > j2` (which requires printing `i1`, `i2`, `j1`, `j2`) would be written as

```
check_econd23( i1 , < , j1 , || , i2 , > j2 , mcerr);
```

and so on. There are some more similar macros defined in the file `FunNameStack.h`. But there is no cover for all cases. In a generic case one can use generic macro

```
#define check_econd(condition, add, stream) if(condition) \
{ funnw.ehdr(stream); stream<<'\n'<<#condition<<'\n'; \
stream<<add; spexit(stream); }
```

which checks any condition and prints any string (`add`). The last condition would look like this:

```
check_econd( i1 < j1 || i2 > j2 ,
             "i1="<<i1<<" i2="<<i2<<" j1="<<j1<<" j2="<<j2 ,
             mcerr);
```

The classes and macros supporting these actions are described in subdirectory `wcpplib/util` in files `FunNameStack.*`, where `*` stands for `c`, `ic`, and `h`.

10 Handling point-wise functions

We consider here functions of type $y = f(x)$ determined by a range of points y_i on a mesh of x_i. The function can either be a smooth function or a histogram. The same array of y-points can be interpreted as a smooth function in one context of the program or as a histogram in another context. This function/histogram can be determined on a different meshes: on an arbitrary mesh determined by an array, or by an equal-step mesh determined by minimum, maximum, and step size (or the number of steps). Sometimes it is not possible or not reasonable to convert the latter case to the first (by considerations of efficiency or memory expenses). If we interpret the y-points as values of a smooth function, we would understand them as heights of the function precisely at x-points. When we consider a histogram, the x-coordinates are usually understood as the left edges of the intervals. If the mesh is defined by a point-wise array, it should have by one point more than the array of the heights. The last y-point gives just the end of the last bin and does not have counterpart among the heights.

Moreover, the data can be of different type, such as `float` or `double`. The arrays can be of different type as well, starting from plain C-arrays and up to various templates. Again, it is may be inconvenient to reduce one to another.

The tasks may be ranged from trivial restoring the functions height in a given point to various integrations, histogram filling, or random number generation.

The edge conditions can be varied from considering the first and the last point as the absolute limits (it usually happens for histograms), to extrapolation outside them. The latter can be either infinite or to zero heights. The extrapolation, as well as interpolation, can be computed by straight lines or by more sophisticated curves.

Each such task taken alone needs carefulness, but is not very complicated. If we try to cover all the described options developing a separate program for each, their total number will be product of all these options, and we surely finish with complete nonsence. For example, if we find error or make improvement on one function, the question arises whether we should edit and debug tens or hundreds of similar others. It will be impossible to manage this system and to sustain it in a coherent state.

Therefore we needed to generalize the problems and to write generic, but still reasonably efficient algorithms. The file `tline.h` is an attempt to do so.

The two last options, tasks and the edge conditions, cannot be generalized. So we need a separate function for each task and this function may have auxiliary arguments govering its behaviour outside edges.

Switching between `float`, `double`, or, may be, longer floating point types is easily implemented by templates, if this type is a template parameter. We need only to remark that we currently assume that x, y, and results of calculations according to the given task (e.g. integration etc.) belong to one type. Their discrimination looks like unnecessary so far (Too intensive use of templates can reportedly lead to a drastic increase of the size of the executable program, as well as the time of compilation due to imperfectness of compilers.) Switching between types of arrays can also be implemented by templates, if the array type is a template parameter.

And the last problem, switching between the mesh types, can also be implemented with either templates or by a system of base and derived classes. Indeed, there are only a few operations which any mesh class should support. The type of mesh affects only the implementation of these functions. Since we need to invoke templates anyway, it is perhaps more reasonable to use templates here as well. So there will be two separate template classes, one for point-wise mesh and another for equal-step mesh:

```
template<class T> class EqualStepCoorMesh
template<class T, class D> class PointCoorMesh
```

Here `T` is the type of the data and `D` is the array type (plain or templated). The first class has the following data members:

```
template<class T> class EqualStepCoorMesh
{public:
  ...
private:
  long q;  // the number of steps;
  // Attention: if you count the number of points with the
  // last point of the last step there will be q+1 points.
  T xmin;
  T xmax;
  T step;
};
```

The second class has the other data members:

```
template<class T, class D> class PointCoorMesh
{public:
  ...
private:
  long q;  // the number of points
  // The number of intervals is q-1.
  // Therefore q has to be 2 or more
```

```
#ifndef TLINE_REDUCE_TO_RAW_ARR
  //PassivePtr< D > amesh;
  D* amesh;
#else
  T* amesh;
#endif
  T xmin;
  T xmax;
  // auxiliary thing to accelerate finding intervals
  mutable T x_old; // previous x for finding interval
  mutable long n_old; // -1 if there is nothing
};
```

First of all, note that the real data for mesh are assumed to be kept externally, in a different place. This is a reasonable decision. The mesh could be very large, it could be kept in another well suited container, and there might be no necessity to copy it to this local place only for integration tasks.

Note also that there is a macro `TLINE_REDUCE_TO_RAW_ARR` which allows to reduce any array `D` to plain array (see constructor `PointCoorMesh<T,D>(long rf, D* famesh)` in `tline.h`). This was assumed to increase the speed of calculations. But no significant effect was detected for real programs, and this option is not used.

But, whatever it is used or not, this class contains the *plain* pointer to the array. This is departure from the rule saying that any application class should not contain plain pointers. Otherwise, if the object pointed disappears, the pointer gets invalid and the program will lead to mess. Normally we would put here the passive smart pointer. But this pointer is intrusive and it cannot point to *plain* array. In many cases it is necessary to work with plain arrays.

So here we need to admit a compromise and allow a plain pointer as a class member. This departure is partially excused by the fact that this class is not fully application class, but it is semi-utility class, whereas we required the plain pointers to be eliminated only from the application classes. The programmer should strongly remember that the point-wise mesh is not designed for long keeping, but should be initialized immidiately before its use. In this context it could be regarded as a pointer inside a function, and this gives anothe excuse for its use.

A few examples of the member functions. The functions “`inline long get_qi(void)`” return the number of intervals. The mesh classes have many other functions like

```
  int get_interval(T x,
   long& n1, T& b1,
   long& n2, T& b2) const ;
```

and so an. Here `n1` and `n2` are the numbers of left and right point or the number of current and next bin of histogram. Correspondingly, `b1` and `b2` are the x-coordinates of left and right point or left and right edge of the current bin.

Such mesh classes allows to write more generic algorthms and reduce the product of all options. For example, the simplest function for obtaining an y-value of a histogram (“step array”) corresponding to x is

```
template<class T, class D, class M>
T t_value_step_ar(const M& mesh,              // dimension q
  const D& y,                  // dimension q-1
  T x,
  int s_include_last_point = 0)
                  // 0 - not include, 1 - include
{
  mfunname("double t_value_step_ar(...)");
  double xmin = mesh.get_xmin();
  double xmax = mesh.get_xmax();
  if(x < xmin) return 0.0;
```

```
  if(s_include_last_point == 0)
    if(x >= xmax) return 0.0;
  else
    if(x > xmax) return 0.0;
  long n1, n2;
  T b1, b2;
  int i_ret=0;
  i_ret = mesh.get_interval(x,
    n1, b1,
    n2, b2);
  check_econd11(i_ret , != 1 , mcerr);
  return y[n1];
}
```

As was explained, it can work with any mesh and array of any type.

And so on. There are many other functions, for whose description I don't have time. But their names and comments in `tline.t` can clarify their duties.

As a real example we can consider the computing of the cross section in the class `SimpleTablePhotoAbsCS` of file `heed++/code/PhotoAbsCS.h`. This class contains two arrays

```
  DynLinArr< double > ener;   // MeV
  DynLinArr< double > cs;
```

with an arbitrary energy mesh (normally extracted from compilations of experimental data) and a cross section at these points. So this is understood as not a histogram, but as point-wise function, which is smooth, but has a threshold.
The function "`virtual double SimpleTablePhotoAbsCS::get_CS(double energy) const ;`" is expected to return the cross section at any energy. Below is this function step by step.

```
double SimpleTablePhotoAbsCS::get_CS(double energy) const
{
  mfunname("double SimpleTablePhotoAbsCS::get_CS(double energy) const");
```

putting function name in the stack, sect. 7.6 and 9.6.

```
  long q = ener.get_qel();
  if( q == 0) return 0.0;
  check_econd11(q , ==  1 , mcerr);
```

determining the number of points, sect. 9.1 and return if zero (assuming the cross section is zero). If only one point is given, it cannot determine how to interprete it, therefore it generates error, sect. 9.6.

```
  if(energy < threshold)
    return 0.0;
```

If energy less then threshold, nothing to do.

```
  else
  {
    if(energy <= ener[q-1])
    {
      PointCoorMesh< double, const DynLinArr< double > > pcmd( q, &ener);
```

If energy is in the interval of a mesh, initialize the mesh with `double` points kept in array `DynLinArr< double >`.

```
      double s;
      s = t_value_generic_point_ar< double, DynLinArr< double >,
        PointCoorMesh< double, const DynLinArr< double > >  >
        (pcmd, cs, &my_val_fun, energy,
         1, threshold, 0, DBL_MAX);
      return s;
```

Calculate and return the value. Note that the function `t_value_generic_point_ar` compute generic function. The function should be provided as an argument, here `&my_val_fun` (see below). In addition, note that the four last arguments having the following description

```
  ...
 int s_extrap_left, T left_bond,
 int s_extrap_right, T right_bond);
```

instruct `t_value_generic_point_ar` to extrapolate the point array to the left, if the requested energy is less than the position of the first point, but higher than `threshold`. This allows, in particular, to extrapolate the Henke tables below 10 eV, when an ionization potential is less than that.

```
    }
    else
    {
      if(energy == DBL_MAX)
        return 0.0;
      else
        return cs[q-1] * pow( energy , -2.75 ) / pow( ener[q-1] , -2.75 );
    }
  }
}
```

If energy is higher, use "generic" applorimate extrapolation $1/E^{2.75}$ similar to ordinary cross sections. The mentioned generic function is here:

```
double my_val_fun( double xp1, double yp1, double xp2, double yp2,
   double xmin, double xmax, double x)
{
  double res;
  if(sign_nonlinear_interpolation(xp1, yp1,
  xp2, yp2, xmin) == 1)
    res = t_value_power_2point<double>
      (xp1, yp1, xp2, yp2,
       x);
  else
    res = t_value_straight_2point<double>
      (xp1, yp1, xp2, yp2,
       x,
       1);
  return res;
}
```

This function calls another three functions. The function `sign_nonlinear_interpolation` is "local" function of HEED package. It allows the program to decide whether to use power interpolation E^r or straight interpolation. If the curve falls and the energy is far from the threshold, the power interpolation might be more reasonable. But only two last points of Marr and West table are far enough from each other to provide significant difference at 50 keV energies, which is not important. The linear interpolation is usually more safe regarding various odd fluctuations. So it is also worth considering. The paper [1] was computed with the linear interpolation (due to an oversight; despite all precations one cannot avoid oversights in large programs). So we provide both variants. The macro `ALWAYS_LINEAR_INTERPOLATION` is defined in file `heed++/code/PhotoAbsCS.h`. The function `sign_nonlinear_interpolation` is given below.

```
int sign_nonlinear_interpolation(double e1, double cs1,
 double e2, double cs2, double threshold)
{
#ifdef ALWAYS_LINEAR_INTERPOLATION
  return 0;
```

```
#else
  if(cs2 >= cs1 || cs2 <= 0
     || e1 < 300.0e-6 || e1 < 1.5 * threshold )
    return 0;
  else
  {
    double pw = log(cs1/cs2) / log(e1/e2);
    Iprintn(mcout, pw);
    if( pw >= -1.0 ||  // good case for linear interpolation
pw < -5.0)     // unclear odd case, power would be dangerous
      return 0;        // linear
    else
      return 1;        // non linear
  }
  return 0; // to make sure
#endif
}
```

The functions `t_value_straight_2point` and `t_value_power_2point` extract the value of the straight of power interpolation by two points:

```
template<class T>
T t_value_straight_2point
(T x1, T y1, T x2, T y2,
 T x,
 int s_ban_neg
)
{
  mfunname("double t_integ_straight_2point(...)");
  check_econd12(x1 , == , x2, mcerr);

  T a = (y2 - y1)/(x2 - x1);
  // More numerical precision (although it is not proved),
  // starting from what is closer to x
  T res;
  T dx1 = x - x1;
  T adx1 = (dx1 > 0) ? dx1 : -dx1;       // absolute value
  T dx2 = x - x2;
  T adx2 = (dx2 > 0) ? dx2 : -dx2;            // absolute value
  if(adx1 < adx2)  // x is closer to x1
    res = a * dx1 + y1;
  else
    res = a * dx2 + y2;
  if(s_ban_neg == 1 && res < 0.0) res = 0.0;
  return res;
}

template<class T>
T t_value_power_2point
(T x1, T y1, T x2, T y2,
 T x)
{
  mfunname("double t_value_straight_2point(...)");

  check_econd11(y1 , <= 0.0, mcerr);
  check_econd11(y2 , <= 0.0, mcerr);
  check_econd12(y1 , == , y2, mcerr);
  check_econd12(x1 , == , x2, mcerr);
  T pw = log(y1/y2) / log(x1/x2);
  check_econd11(pw , == -1.0 , mcerr);
```

```
  T res = y1 * pow( x , pw ) / pow( x1 , pw );
  return res;
}
```

11 Random Numbers Generators

The calls of random generators from C++ are performed through functions (by upper case characters):

- `inline void INITE_RANLUX(void)` – read the state of generator from file `ranluxsd.dat`
- `inline void SAVE_RANLUX(void)` – write the state of generator to file `ranluxsd.dat`
- `inline void EXTRACT_RANLUX(fint* ivec)` – extract the array of 25 integers determining the state from generator to the program
- `void RANLUX(ffloat* vec, fint len)` – extract the array of random numbers
- `inline ffloat SRANLUX(void)` — extract the single random number

All these functions are defined in the file `wcpplib/random/ranluxint.h`. This file and the total directory `wcpplib/random` contains also a large number of other obsolete functions. For example, file `wcpplib/random/ranluxint.h` contains macros

```
//#define RANF_INSTEAD_OF_RNDM
#define RANLUX_INSTEAD_OF_RNDM    // currently only this option works
```

which could allow to switch it to other `rndm` random number generator from `CERNLIB`. However, this will change the names of interface functions, i.e. `void RANLUX(...)` etc., and it would not work with HEED, because these names are imprinted in it. This file also contains the definitions of three functions

```
inline ffloat LRANOR(ffloat* arg1, ffloat* arg2)
inline void LSPOIS(ffloat* amu, fint& n, fint& ierror);
inline void HISRAN(ffloat* y, fint n, ffloat xlo, ffloat xwid,
      ffloat& xran);
```

which are not used now. But we mention that these functions call the Fortran functions `lrannor_`, `lspois_`, and `hisran_`, whose sources are currently in `wcpplib/random/ranluxint.f`. The first function is a modified copy of the GEANT3.21 routine `GRANOR` calling `ranlux`. The second function is corrected Poisson generator "`POISSN`' from CERNLIB. It calls `ranlux`. The third function is a modified `HISRAN` program from CERNLIB calling `ranlux` as well.

HEED actually calls C++-remake of `lspois`:

```
long pois (float AMU,int &IERROR);
```

defined in files `pois.h` and `pois.c` and calling `SRANLUX()`.

There is also C++-remake of `GRANOR`

```
void rnorm(float r1, float r2,    // flat random numbers
   float &x1, float &x2); // results
```

defined in files `rnorm.h` and `rnorm.c` and independent on flat numbers generators.

The direct analog of `hispre-hisran` is `chispre(...)` and `chisran(...)` with different arguments and return values, defined in files `chisran.h` and `chisran.c`. But these functions are considered old, althouth they are still used in some places. The replcement is template functions

```
template<class T, class D, class M>
T t_hispre_step_ar(const M& mesh,                // dimension q
   const D& y,                  // dimension q-1
   D& integ_y   // return integrated array
   );   // for power = 0 only
template<class T, class D, class M>
T t_hisran_step_ar(const M& mesh,                // dimension q
   const D& integ_y,                  // dimension q-1
   T rannum);   // for power = 0 only
```

from file `wcpplib/math/tline.h`.

12 Graphics

The interface class `histdef` is defined in file `wcpplib/graphics/histdef.h`.

The main histogram class is the class `histdef`. It contains the name of histogram, the histogram number, the number of dimensions, the numbers of bins, the boundaries and some other auxiliary information. The meaningfull content of histogram is not present in this class. At call of its function `histdef.fill(...)` the content is passed instantly to a histogram driver. Currently the main driver is the `HBOOK` package, but rather experimental possibilities to use OpenScientist (with filling according to AIDA interface and visualization probably through "opaw") and the author's GHist package (with visualization through ROOT) is also provided. There is also possibility to switch it to an alternative old histogram driver "Histog", but it doesn't have own UNIX-compatible graphics driver so far, so we will not discuss this.

12.1 HBOOK

In principle, the initialization of any object of `histdef` might lead to initialization of `HBOOK` histogram. But really it cannot always be done by this way, since prior to initialization of histograms we need to initialize the internals of `HBOOK` package by the call of `HLIMIT` with appropriate size of available memory as an argument. When the histograms are components of global objects, right sequence can be violated with unpredictable result. Therefore the initialization of `HBOOK` histogram is removed from constructor of `histdef` to its special member function `init()`. The initialization by `init()` is marked by a member variable `s_init = 1`. Then, a typical error is missing this function at all, which will normally result in termination. Another consequence of such order is that the copying of the `histdef` object to another place does not affect the associated histogram. The deletion of `histdef` also does not affect the histogram, except the loss of access to it from C++.

The huge problem in practical use of `HBOOK` is the notation of histograms by numbers. At the number of histograms more than ten the user cannot firmly remember the meaning of numbers. Worser, that at initialization of additional histograms there may be necessity to shift the numbers, which destroys the correctness of kumac files. However, `HBOOK` contains the mechanism of "alias" names, which can be used instead of histograms numbers in all, or at least in the most of the commands of `PAW`. This allows us to write kumac scripts with these symbolic names, which can have clear sence and there is no necessity to change them. To associate the names with numbers `PAW` requires a kumac file with alias commands. This file is generated in `void HistManager::write(void)`. To do this the program needs to know the name (or notation) of every histogram, its number, and provided that the numbers are not repeated. Therefore, the numbers are generated automatically in sequencial order starting from 1. The names are defined by the user at initialization of histdef. To allow `PAW` to function correctly, the names should not have blanks and underscores. Thus blanks are filled automatically in constructors of `histdef`, and the underscores are substituted to '-' as well, therefore the name appeared as alias and in inscription under the figure may have some different appearance with respect to the original name.

To allow correct generation of numbers the program should know names and numbers of all histograms inited in the system. This is implemented by the class `HistManager`. In addition, initialization of this class contains the call of `HLIMIT`. The constructor of this class also takes the root of file names, in which the histograms and the alias file will be written. The hbook file name is taken as is, but the alias file is named as hbook plus '.l'. The member function `write()` calls `HBOOK` function `HRPUT(...)`, which writes all histograms to disk file readable by PAW. Also the function `write()` writes the alias file.

A simple program filling two histograms looks like this:

```
#include "wcpplib/util/FunNameStack.h"
#include "wcpplib/graphics/histdef.h"

String taskname="simple_graph_prog";

#ifdef USE_HBOOK
#define PAWC_SIZE   1000000
float pawc_[PAWC_SIZE];
HistManager histmanager(taskname+".hbook",PAWC_SIZE);
#elif defined(USE_OPEN_SCIENTIST)
HistManager histmanager(taskname+".aida");
#elif defined(USE_GHIST)
HistManager histmanager(taskname+".ghist");
#else
HistManager histmanager(taskname+".hist");
#endif

int main(void)
{
  mfunnamep("void main(void)");
  long qx = 10;
  double xemin = 0.0;
  double xemax = 10.0;
  long qy = 20;
  double yemin = 0.0;
  double yemax = 20.0;

  histdef hist1("hist1", qx, xemin, xemax);
  hist1.init();
  histdef hist2("hist2", qx, xemin, xemax, qy, yemin, yemax);
  hist2.init();

  // just a few any ‘‘events’’
  hist1.fill(2.5);
  hist1.fill(3.5);
  hist1.fill(3.5);
  hist1.fill(4.5);
  hist2.fill(5.5, 4.5, 1.0);
  hist2.fill(5.5, 5.5, 2.0);
  hist2.fill(5.5, 6.5, 1.0);
  hist2.fill(6.5, 5.5, 1.0);
  hist2.fill(6.5, 6.5, 2.0);
  hist2.fill(6.5, 7.5, 1.0);

  histmanager.write();
}
```

The script for flexible (adjusted) compilation of this program would be much larger than it:

```
#! /usr/bin/env tcsh

echo run compiler

if ( $?USE_HBOOK ) then

echo "histdef with HBOOK will be linked"

setenv FORTLIBARIES "$SPACKLIB -L$CERNLIB -lkernlib"

$CPPLINKD -I$WCPPLIB -DFUNNAMESTACK -DDO_CHECKS \
-DUSE_HBOOK \
-o simple_graph_prog.e simple_graph_prog.c \
$WCPPLIBOBJ/graphics/histdef.o \
-L$WCPPLIBOBJ  -lsafetl -lstream -lutil -lmath -lgraph \
$FORTLIBARIES \
$CFLIBSERV

else if ( $?USE_OPEN_SCIENTIST ) then

echo "histdef with OPEN_SCIENTIST will be linked"

echo "aida-config --cppflags:"
aida-config --cppflags
if ($status != 0) then
    echo "aida-config is not seen"
    echo "Have you run?:"
    echo "source <install_path>/OpenScientist/<version>/aida-setup.csh"
    exit
endif

$CPPLINKD -I$WCPPLIB -DFUNNAMESTACK -DDO_CHECKS \
-DUSE_OPEN_SCIENTIST \
`aida-config --cppflags` \
-o simple_graph_prog.e simple_graph_prog.c \
$WCPPLIBOBJ/graphics/histdef.o \
-L$WCPPLIBOBJ  -lsafetl -lstream -lutil -lgraph \
`aida-config --libs`

else if ( $?USE_GHIST ) then

echo "histdef with GHIST will be linked"

$CPPLINKD -I$WCPPLIB -DFUNNAMESTACK -DDO_CHECKS \
-DUSE_GHIST \
-o simple_graph_prog.e simple_graph_prog.c \
$WCPPLIBOBJ/graphics/histdef.o \
-L$WCPPLIBOBJ  -lsafetl -lstream -lutil -lgraph

else if ( $?USE_HISTOG ) then

echo "histdef with Histog will be linked"

$CPPLINKD -I$WCPPLIB -DFUNNAMESTACK -DDO_CHECKS \
-DUSE_HISTOG \
-o simple_graph_prog.e simple_graph_prog.c \
$WCPPLIBOBJ/graphics/histdef.o \
-L$WCPPLIBOBJ  -lsafetl -lstream -lutil -lgraph
```

```
else

echo "ERROR: not defined graphics driver."

endif
```

If the environment variable USE_HBOOK is set and also a macro variable with the same name is set as well, this program will generate two files simple_graph_prog.hbook and simple_graph_prog.hbook.l. The hbook-file is binary and we cannot look in it. The hbook.l file is text file, which looks like:

```
alias/create hist1                                1
alias/create hist2                                2
```

After execution of this file the histogram number 1 can be accessed also by alias name hist1, and similarly for histogram number 2. The typical kumac to load these files to PAW looks like this (file simple_graph_prog.kumac)

```
hist/file 40 simple_graph_prog.hbook
*
opt *
opt stat
opt grid
opt nbox
set *
set fit 101
set stat 1111111
igset *
opt fit
exe simple_graph_prog.hbook.l
return

* This is kumac file for program "paw" (sometimes also called as pawX11).
* example of session:
* shell> paw    (or pawX11)
* PAW > exe simple_graph_prog
* PAW > h/pl 1
* PAW > h/pl 1
* PAW > h/pl 2 box
* PAW > exit
```

At the end of this file you can see commented-off example of interactive session.

12.2 AIDA and OpenScientist

If the environment variable USE_OPEN_SCIENTIST is set and also a macro variable with the same name is set as well, this program will generate file simple_graph_prog.alias. This is the text file in XML-format. It is quite difficult to understand why do we need this XML format in our physics studies. Please see manuals of OpenScientist for instructions how to visualize it in opaw of onX.

Note that as histdef objects contain only references to HBOOK histograms (so at copying of histdef the new instance of HBOOK histogram is not created, but instead the both objects refere to the same histogram), by the same way histdef objects contain only references to AIDA histograms. So logic of program is not affected by this switch.

12.3 GHist and ROOT

12.3.1 Rationale of GHist

The package GHist was designed to support copies, but not references. GHist object can be copyed and this creates new histogram. This design concept is based again on object management methodology, according to which the objects are as a rule copyable, and copying leads to cloning of all parts. One specific difficulty regarding histogram is that usually histogram is assigned a name, which serves for double purpose. Firstly the name is printed below or above it. Second the name might be used to identify it uniquely and to access it in graphics interface. There may be also not a single name, but a couple: a name and a notation. Both are printed, but only notation is used in dialog. For example, in ordinary HBOOK there is a name and notation in the form of histogram number. The name (and the notation) is naturally to keep as a part of histogram. But the name (and the notation) kept in histogram object cannot serve for the second purpose, since after copying it may not remain unique. Therefore currently the histogram objects of class GHist do not keep the literal names at all. Instead the names are assigned to references to histograms, which will be described just below. Only just a few words to finish this issue: if you want to have two or many objects, which fill the same common histogram, just access this histogram by passive pointers.

In order to be output in centralized manner, the GHist object could be registered in class `ManagerGHist`. This could be registration through passive pointer. The histogram is granted a name only at registration. The names and the notations are kept in the special lists kept in the objects of class `ManagerGHist`. Naturally, these lists have the same number of entries, as the "main" list keeping the passive references to objects of GHist. Both the names and notations are literal names in this approach. In fact, this design is very reasonable.

Thus, normally the class `ManagerGHist` does not delete histograms at the deletion of self. The same class is convenient to use for reading histograms in ROOT interactive sessions. In order to clean the memory at the end of this session (i.e. to provide functioning of memory leak checkers) it now needs to delete the objects at the deletion of this manager. This is provided by some switches of passive pointer, which allow it to behave like ordinary shared pointers and delete the objects at dissapearance of the last pointer. We mention it here because it is a rare exception from the common rule saying that shared pointers are not necessary.

One else rationale of creation of this package is that all the known packages are not enough flexible regarding the types of filled data (which may be from "short int" to "long double"), weights (the same variation), possible number of entries, the types of coordinate scale (which can be normal equal-step, logarithmic, generic point-wise). The known packages also do not provide all necessary statistics related to data filled to histogram, such as moments of the all filled data, moments of data accepted (in the borders of given scale), moments of data accepted in each bin, moments of weights. The known packages do not usually provide possibility to save filled entries in order to re-fill histogram with changed scale. The known packages might also have deficiencies in providing "profile" histograms as alernative for many-dimensional histograms. The class GHist is an attempt to include in a single class all these possibilities. This class is compiled and works slower than ordinary histogram classes, but in real scientific programs (such as HEED) this is not noticable, whereas its wide possibilities are assumed to be very useful. But with the histdef package most of these is not accessable and not used.

12.3.2 Interface of histdef with GHist

In fact, the same functionality is convenient to use for interface with histdef. (This would be one more exception). Indeed, we just allocate GHist objects in free memory, register their addresses in class `ManagerGHist`, also memorize them in objects of histdef, and again switch passive pointers to delete these objects at the deletion of the last pointer. By this way we reach the same reference semantic of histdef, thus providing that the logic of the program previously used HBOOK is not affected by the switch from HBOOK to GHist.

After run the program produces two files. For the example above it would be `simple_graph_prog.ghist` and `simple_graph_prog.ghist.l`. The both files are (currently) textual and could be viewed. The beginning of the first file looks like this:

```
GHist file, format= 2
```

```
Next histogram type is 1
histname is hist1
histnota is hist1
GHist<TC, TB, TW, TQ>::print(ostream& file, int l)( l=1 s_short_output=1):
GHist<TC, TB, TW, TQ>:
  1 1
  dimname=DynLinArr<T>: qel=0

  4 1 0 0 1
  mesh=DynLinArr<T>: qel=1
    ActivePtr<X,C>:  ptr!=NULL: VirtEqualStepCoorMesh<T>:
      EqualStepCoorMesh<T>:
        Type of T is (in internal notations) f
        q=10, xmin=0, xmax=10, step=1
  hist=DynArr<T>: qdim=1
    DynLinArr<T>: qel=1
      10
    0
    0
    1
    2
    1
    0
    0
    0
    0
    0
  0
  s_profile_hist=DynLinArr<T>: qel=1
    0
  ncoor_profile_hist=DynLinArr<T>: qel=0
  profile_hist=DynLinArr<T>: qel=0
  0
  4
  4
  1
  n=0, entries_missed_hist[n][0]=0, entries_missed_hist[n][1]=0
  0 0 0
  attem_moments_at_fill_hist=DynLinArr<T>: qel=0
  saved_moments_at_fill_hist=DynLinArr<T>: qel=0
  0
  weight_moments_hist=DynArr<T>: qdim=0
    DynLinArr<T>: qel=0
  0 0 0 0
  ntuple: coordinates and weight
  0 0 0
  attem_moments_ntuple=DynLinArr<T>: qel=0
  saved_moments_ntuple=DynLinArr<T>: qel=0

Next histogram type is 1
histname is hist2
histnota is hist2
...
```

We see content of this histogram and various control and auxiliary numbers. This file can be read back to program and visualized in the form of plots.

The second file is designed in order to see it in a separate window and to use histogram notations in exchange with ROOT:

```
In total there are 2 histograms
```

```
Below it will be printed the following:
the histogram sequencial number, the notation, the name, qdim
0    hist1    hist1    1
1    hist2    hist2    2
```

Since histdef, designed initially for HBOOK contains only one histogram name, the histogram notation and name appear to be the same at the output of Ghist when it is run through histdef.

12.3.3 Interface of GHist to ROOT

What should you do with this? In ideal life one could create dinamic library from class `ManagerGHist`, to load it in ROOT, to write a simple function which converts histograms of GHist to histograms of ROOT, and then to do anything one wants. All the internal information of GHist (which may keep a lot of auxiliary histograms and statistical data, which hopefully will be very useful at physical-program debug) would be available from ROOT, as from ordinary C++ program. But our real life is not ideal. One of the confirmations of that is the fact that class `ManagerGHist`, as well as class `GHist` and a lot of other classes from wcpplib, on which they depend, is *not correctly compiled* by C++ compiler of ROOT known as CINT. Even some simplest things are not compiled, although they were used (and, of course, were compiled) by the author during last 10 years at every computer and every operating system he tried. The problems are quite numerous and it does not seem to be useful to try to adapt wcpplib or to search a single error in CINT. Therefore, at least as a temporary solution another aproach have been used. A special medium class has been created, which combines ROOT and GHist components while does not fully expose them to each other. This class is called `RootMGHist`. Today it has some limited possibilities, but possibly it can and will be developed and possibly will allow to access and to use directly or indirectly all information from GHist. Thus, despite of problems, this interface GHist-CINT-ROOT seems to be the most perspective and convenient among all other possible methods of application of GHist.

So, what should you do with this? After building `wcpplib` (and proper installing ROOT system, see its manuals) run script `wcpplib/graphics/build_rootm.b`, which should create dynamic library `libRootMGHist.so`. In order to execute this successfully, you have to have ROOT installed and in UNIX-like systems you should have 3 related environment variables set. You might want to create a file which will set then after its "soursing" in tcsh:

```
setenv ROOTSYS .../root
setenv PATH "$ROOTSYS/bin:${PATH}"
setenv LD_LIBRARY_PATH "${LD_LIBRARY_PATH}:$ROOTSYS/lib"
```

If at execution of `wcpplib/graphics/build_rootm.b` some diagnostics is printed about error and recommendation to compile with `-fPIC`, you are advised to do the following. First you are advised to add this `-fPIC` to definition of compilation and linking commands `CPPCOMPILE`, `CPPCOMPILED`, `CPPLINK`, and `CPPLINKD`. Then you are recommended to clean and recompile everything. Finally you are advised to add `-fPIC` to compilation commands in `wcpplib/graphics/build_rootm.b`.

Then, after this is done and `wcpplib/graphics/build_rootm.b` has successfully created `libRootMGHist.so`, you have to run "root" and execute the following:

```
.L libRootMGHist.so
```

Then you have to execute either of the following. Either this:

```
RootMGHist r("simple_graph_prog.ghist"); // do everything, what is mentioned in comments below
```

or this:

```
RootMGHist r;  // instantiate object which can be mediator between
// root and GHist.
r.open_file("simple_graph_prog.ghist");  // creates ManagerGHist object,
```

```
// reads given file and creates all GHist objects of appropriate
// template types.
r.make_all();  // Corverts Ghist objects to ROOT HF** objects with names,
// corresponding to histogram notations.
```

You have loaded and initialized everything. Now you can enjoy by the show:

```
r.list();       // to see on screen what GHist objects have been created
hist1.Draw();  // To draw HF** object
hist2.Draw();  // You will see some points
hist2.Draw("box"); // You will see the same histogram by ‘‘boxes’’.
.q
```

12.4 Histogram borders and steps

A small but important for HEED function is a function to adapt the histogram boundaries to the discrete data. When we convert the number of conduction electrons to energy by multiplying the number by the mean work per pair production w, we receive a floating point number, but it remains to be discrete with some step equal to the value of w. If the histogram step is not divisible by w, the different bins of histogram will receive different number of possible values. If the width of histogram bin is not larger than w by the order of magnitude, the bin height will visibly fluctuate. In order to eliminate this detrimental effect, we adjust the histogram edges in order to make the bin divisible, but closest to recommended. This is done in function

```
void choose_lim(long hist_q, double hist_min, double hist_max,
// above are recommented values (hist_q is required)
                double data_step,
                double& hist_minm, double& hist_maxm,
                double& hist_stepm, long& n_step_in_bin,
                int s_print);
```

The upper line is what is reccommended. The `data_step` is w in our case. The adjusted values are below it.

13 Geometry

The geometrical modeling is quite interesting issue itself. The system used in HEED++ is described in ref. [8]. The system is located in the directory `wcpplib/geometry`.

The geometry is dimension-less. No cm, no mm etc., just and abstract lengths and sizes of type `vfloat` defined in `wcpplib/geometry/vfloat.h` as “`typedef double vfloat` ”. It was assumed also that the user could be able to switch from `double` to `float` if he needs faster and less precise calculations, but whether the program works in this mode was not tested up to now.

The main file introducing vectors (the direction and the length in space, `class vec`), points (the position in space, `class point`), basises (three ortogonal unit vectors, `class basis`), and systems of coordinate (point and basis, `class abssyscoor`, `class fixsyscoor`) is defined in `wcpplib/geometry/vec.h`. This file also defines `class absref`, which allows to convert all geometrical objects between different systems of coordinates without manual encoding the converting procedures. This file defines a special global variable `int vecerror`, whose non-zero value indicate an error in geometrical calculations. It is also used a return signature of irregular situations in geometrical, such as zero-length vector or product, when it should not be such, and many other various cases. When such return is expected, the calling function checks the value of `vecerror`, clears it and acts accordingly. If the problem is unexpected, `vecerror` may be left unchecked till the call of the next geometrical function. Almost all functions (except very short ones) check `vecerror` at the first line, and trigger termination (or exeption throwing) sequence if it is not zero.

For example, a function computing sinus of angle between two vectors from file `wcpplib/geometry/vec.c`:

```
vfloat sin2vec(const vec& r1, const vec& r2)
  // sinus of angle between vectors
{
  pvecerror("vfloat sin2vec(const vec& r1, const vec& r2)");
  vfloat lr1=length2(r1);  vfloat lr2=length2(r2);
  if(lr1 == 0 || lr2 == 0){ vecerror=1; return 0; }
  vfloat sn=length(r1||r2);
  sn=sn*sn;
  sn = sqrt(sn/(lr1*lr2));
  return sn;
}
```

The first line of this function means a macro-call substituted by

```
#define pvecerror(string)   mfunname(string); \
  if(vecerror!=0)                           \
  { mcerr<<"vecerror is not zero, program is terminated\n"            \
        <<" function detected error is "<<string<<'\n'; spexit(mcerr);}
```

, where `spexit(mcerr)` was explained in sect. 9.6. The second line computes the lengths of two supplied vectors. The third line checks if one of them is zero, and if it so, it assigns `vecerror=1` and returns. Indeed one cannot determine the vector angle for the zero-length vector. But this function cannot determine whether it is a fatal error that should lead to program termination. If the calling function ready to react on this, that might be a routine situation. Then the called function should check `vecerror`, make it zero again, and switch algoritm accordingly. If it does not do this, at the next call of any geometrical function the program will be terminated.

Considering the rest of this example we have to remark, that the operation `||`, when applied to vectors, means "vector product". The operation `*` with vectors indicate scalar product:

```
  friend vfloat operator*(const vec& r1, const vec& r2)
    {   return r1.x*r2.x+r1.y*r2.y+r1.z*r2.z; }
  friend vec operator||(const vec& r1, const vec& r2)
    {    return vec(r1.y*r2.z-r1.z*r2.y,
       r1.z*r2.x-r1.x*r2.z,
       r1.x*r2.y-r1.y*r2.x); }
```

From this the reader also see how to access the vector "coordinates". The vector is declared naturally and without data encapsulation:

```
class vec: public absref
{public:
  vfloat x,y,z;
  ...
};
```

The point is declared through the vector:

```
class point: public absref virt_common_base_pcomma
{public:
  vec v;  //
  ...
};
```

We remark here that the point and the vector are different entities in this system, since their numerical representation is changed differently at passing to different system of coordinates or at translations and rotations, see ref. [8].

A range of other files from the same directory, `straight.h`, `plane.h`, `circumf.h`, `polyline.h` define more complicated geometrical objects than vector, but still quite trivial.

The simplest objects like `vec` or `point` cannot be referred by passive pointers, since they are not derived from `class RegPassivePtr` (which declares the necessary "intrusive" ingredients for working with them). This class occupies some memory, and this would be too much overhead for it. Therefore an auxiliary objects are provided for the cases, when the vector etc. should be referred. They have just a siffix `Reg`, like this:

```
class vecReg: public vec, public RegPassivePtr
{public:
  vecReg(vfloat xx, vfloat yy, vfloat zz ): vec(xx, yy, zz) {}
  vecReg(void): vec() {}
};
```

File volume.h defines class absvol, a key sbstract class from which all volumes are derived. It defines functions like

```
  virtual int check_point_inside(const point& fpt, const vec& dir) const  =0;
  virtual int find_embed_vol(const point& fpt, const vec& dir,
        manip_absvol_treeid* atid)const;
  virtual int range(trajestep& fts, int s_ext,
       int& sb, manip_absvol_eid* faeid) const ;
  virtual int range_ext(trajestep& fts, int s_ext) const =0;
```

and other key functions participating in trajectory tracing through volumes. It also defines manip_absvol, "manipulator" of volumes, a class enveloping a system of coordinates and a volume defined in it.

File surface.h allows to create volumes of complicated shape by intersection of semi-spaces determined by the surfaces.

Files vol_subtraction.h, vol_intersection.h, and vol_union.h determines "boolean" operations.

A range of other classes box.h, tube.h, cylinder.h, tube_box_its.h define some most frequently used shapes.

File trajestep.h is very important for determination of step length at directory propagation according to the required precision. The class gparticle.h and derived need a global object, defined as

```
extern trajestep_limit gtrajlim;
```

This object is expected to be initialised in the header user program, for example like this:

```
extern trajestep_limit gtrajlim;
trajestep_limit gtrajlim(100.0*cm, 1000.0*cm, 0.1*rad, 0.2*rad);
```

(We use here dimensional units from CLHEP). The arguments of this contructor say the following.

The maximal "geometrical" range of the particle is allowed to be 100.0*cm. This means that there is no reason for variable curvature of the trajectory. That is there is no magnetic (or significant electric) field. The curvature is variable if the shape of the trajectory differs from circle. So the constant electric field causes variable curvature. The constant magnetic field gives constant curvature only when normal to the trajectory. If curvature is variable, this parameter should be possibly much less.

The threshold radius of curvature, at which the program switches between the propagation of the trajectory by circle (at less radius) and simplier and faster propagation by straight steps is 1000.0*cm, quite a lot.

If there is slight curvature (possibly due to fields), and propagation is performed by straight steps, the program limits the angle at which the step is seen from the center of the circle. This leads to restriction of step length. In our example this limit is 0.1*rad.

Similarly, if the propagation is performed by segments of circles, the program limits the angle as well. In particluar, this is convenient for dealing with magnetic fields, where particles go by spilars, and too long steps by parts of circles can reduce precision. On our example, the limit is 0.2*rad.

Another example, taken from package which reconstructed the particle trajectories and momenta in a magnetic spectrometer whose magnet had poles of 1x1 m^2 and half a meter gap, is

```
trajestep_limit gtrajlim(10.0*cm, 1000.0*cm, 0.1*rad, 0.2*rad);
```

It differs only by lower first argument, the maximal "geometrical" range. Note that this limit is rather broad comparing with the magnet size and taking into consideration that the field is sharply reduced at the exit from the gap. But this does not spoil the momentum resolution, since at the end of each step the program corrects the particle speed both by value (for non-normal fields) and by direction according to the mean field which is obtained as average of the field at the beginning of the step and that at the end. This complicated algorithm is in function `void mparticle::new_speed(void)`. This gives very precise results at a modest number of steps.

File `gparticle.h` defines an abstract point in space-time-velocity, `class stvpoint`, and generic mass-less and energy-less abstract particle `class qparticle`. The latter keeps the previous, current and next `stvpoint` and provides step-by-step track tracing.

The mass, and therefore the kinetic energy, appears in `class mparticle`, file `mparticle.h`. The class `mparticle` is of course derived from `class gparticle`. To the `mparticle` object a force can be applied and the trajectory can be curved or even inversed backward. The `mparticle` can accumulate the kinetic energy falling in a potential field, or loss it rising against it. The `mparticle` can fly in non-potential fiels, keeping the energy constant and the trajectory curved. In particular this was used for the momentum reconstruction in the mentioned magnetic spectrometer, and it is proved to be quite precise even for spectrometrical applications. The class `mparticle.h` is also dimension-less, but contains `static double speed_of_light` member which should defined in a choosen system of units.

The last class and file to be mentioned here is `wcpplib/particle/eparticle.h`, which is derived from `class mparticle` and from `class particle_type` (see below) and defines charged particle, capable to react for electric and magnetic fields according to science. In particular, the classes from HEED `HeedParticle` (the incident particle from HEED) and `HeedDeltaElectron` (δ-electron) are derived from `eparticle`.

14 Particles

The file `wcpplib/particle/particle_def.h` defines a range of known particles like proton, electron etc. Each "`particle_def` " object denotes type of particle rather than an individual particle. The predefined particles are defined in global objects. The user can add another particles as any kind of objects.

In order to indicate that a given object belongs to a particles of a certain sort, one is expected to use passive pointer to the corresponding `particle_def` object. Otherwise one can create a class derived from `class particle_type`, which contain this reference. This class is also defined in the file `wcpplib/particle/particle_def.h`. In particular the mentioned `class eparticle` is build by this way.

The following particles are currently defined:

```
extern particle_def electron_def;
extern particle_def positron_def;
extern particle_def proton_def;
extern particle_def anti_proton_def;
extern particle_def neutron_def;
extern particle_def anti_neutron_def;
extern particle_def P11_def;
extern particle_def D13_def;
extern particle_def S11_def;

// light unflavored mesons
extern particle_def pi_plus_meson_def;
extern particle_def pi_0_meson_def;
```

```
extern particle_def pi_minus_meson_def;
extern particle_def eta_meson_def;
extern particle_def K_plus_meson_def;

extern particle_def deuteron_def;
extern particle_def alpha_particle_def;
```

15 Matter, general definitions

The matters (or materials or substanses) are defined in directory `wcpplib/matter`.

15.1 Atoms

The file `wcpplib/matter/AtomDef.h` defines atoms and atom mixtures. For the atoms we specify only the basic information: the name, the notation, the atomic weight and charge. The principle of definitions of atoms is a dictionary or a database: the atoms are not repreated, each atom is presented in the total system no more than one time. (It is similar to definition of particles, but for particles the ban to define two the same particles is not implemented so far. So we do not recommend to refere to the particles by notations.) The system knows each atom presented in it. The atom characteristics can be obtained by literal notation. Therefore, the system does not allow the secondary initialization of the atom with already registered name or notaion. But the copying is not forbidden, since it is sometimes necessary by technical reasons and since it is just a general approach of this program, that any object can be copied. The the copy constructor and the assignment operator are allowed. There is a special function `AtomDef::verify()`, which checks that there is no two atoms with the same name or notation in the system. When the user program wants to refere to atom, it has to use either char* (String) notation, or pointer (or reference) to one of these objects. (It is a contrversial point, whether the system with search by notations and ban of secondary initialization, while alloing copying where necessary, but it did not procude any harm so far.) As usually, in the case of pointers I recommend to use protected pointers to external objects PassivePtr. The user pogram can initialize the new atoms. The standard atoms are inited in files `GasLib.h` and `GasLib.c`. To provide initialization of standard atoms prior to their use it might be necessary not to link the program to the object file `GasLib.o`, but to include the file `GasLib.c` in user program.

15.2 Mixtures

To handle uniformly with atomic gases, molecular gases, and various solids, we need to known the answers at questions like what is relative weight by quantity of weight by mass of a given atom in this mixture? To answer such questions we use a special class `AtomMixDef`. The molecule and material classes will be derived from it.

15.3 Molecules

The molecules are defined in the file `wcpplib/matter/MoleculeDef.h`. It is organized like `AtomDef`: a dictionary or database. The molecula can be initialized with parameters of Van-der-Vaals (not sure in correct spelling) parameters, used for calculation of gas density.

15.4 Matter

The materials (assumed solid) are defined in file `wcpplib/matter/MatterDef.h`. The density should be supplied at initialization.

15.5 Gases

The gases and defined in `wcpplib/matter/GasDef.h`.

The characteristic feature of the Gas with respect to any matter is that this matter consists of molecules. Additional feature is that the density can be calculated by temperature and pressure. But this is not always, and therefore it is not characteristic feature. Then with only one this feature (constisting of molecules) we can describe as gas also some other substances, for example, the liquids.

Only the basic information: the data of matter, plus the pressure. Note that the class AtomMixDef undirectly appear twice. It is the base class of matter and molecula. Therefore it is indirectly the base class of GasDef, and the base class of its external elements - DynLinArr¡ PassivePtr¡MoleculeDef¿ ¿molech.

As the base class of GasDef, the class AtomMixDef determines only the relative weights of atoms of different sorts. Also note that the atoms of the same sorts participated in different molecula, included in AtomMixDef as different atoms.

As the base class of MoleculeDef, the class AtomMixDef determines also only the relative weights of atoms of different sorts in the given molecule, since the class AtomMixDef don't have space to encapsulate the number of atoms. But the latter number is also necessary: consider H2, the relative weight of H is 1, and nothing says that there are two atoms. Therefore in the class MoleculeDef there is additional array, which gives the numbers of atoms of each sort, and also there is another parameter giving the total number of atoms in molecule.

There is a special constructor which allows to compute density by Van-der-Waals. But be careful: it is not very good. First the author is not sure that he understood how to mix different molecules with different parameters. Now they are mixed as if they are in different volumes, which is true for ideal gas but may be not true for more precise approach, where molecules interact with each other (which is taken into account in Van-der-Waals). Second, the horizontal segment is currently positioned at the peak, which is wrong: in theory it is located by such a way that the areas enveloped below and above it are equal. It is more difficult to compute, and currently it is not implemented. So the reliable results of Van-der-Waals are expected only quite far from the phase transitions, which is by the way true in general: Van-der-Waals is only a second approximation (if the first is ideal gas). But usually, in the proportional chambers, the gases are such that their density is very well described by the law of ideal gas.

15.6 Atomic library

In the file `GasLib.h` all predefined atoms, molecules, and the simplest gases are gathered. They are global objects.

This all is gathered in a single file, since molecules needs atoms, gases need molecules and atoms, and everything should be inited in a certain order. C++ compilers provide this only if they are gathered in a single file. If one needs to use this to initialize other global objects, he needs to include the file `GasLib.c` in its source file instead of linking with the compiled `GasLib.o`.

Regarding atoms, in principle I am going to inite all atoms from Mendeleev's table, but I haven't finished yet. Only its first half is filled at the moment.

The attension has to be paid to the temperature and pressure, defined in `GasLib.c` and used to compute gas density:

```
double todays_temperature=293.0*kelvin; // room temterature,
                                        // 20 degrees Celsium
//double todays_temperature=293.15*kelvin; // room temterature,
//                                          // 20 degrees Celsium
//double todays_temperature=296.15*kelvin; // anoter room temterature,
                                        // 23 degrees Celsium

double todays_pressure=1.0 * atmosphere;   // sea level
//double todays_pressure=1.0 * atmosphere * 960.0/1000.0;   // level of CERN
```

```
// By different information the altitude is about 400 meters and
// the slope of pressure is about 1 bar per 10 m (somebody says - 12 m).
// But the actual pressure depends on weather, of course!

//double todays_pressure=730.0/760.0 * atmosphere;
// level for one experiment (looks like in CERN as well)
```

16 Atomic database of HEED

The location is `heed++/database`.

- Henke photoabsorption cross sections are in directory `heed++/database/henke`. Each atom is described in separate file named by this atom with extension `.dat`. The total content of this directory is

```
Ac.dat  Bi.dat  Cu.dat  H.dat   La.dat  Ne.dat  Pr.dat  Sc.dat  Th.dat  Zn.dat
Ag.dat  Br.dat  Dy.dat  He.dat  Li.dat  Ni.dat  Pt.dat  Se.dat  Ti.dat  Zr.dat
Al.dat  C.dat   Er.dat  Hf.dat  Lu.dat  O.dat   Ra.dat  Si.dat  Tl.dat
Ar.dat  Ca.dat  Eu.dat  Hg.dat  Mg.dat  Os.dat  Rb.dat  Sm.dat  Tm.dat
As.dat  Cd.dat  F.dat   Ho.dat  Mn.dat  P.dat   Re.dat  Sn.dat  U.dat
At.dat  Ce.dat  Fe.dat  I.dat   Mo.dat  Pa.dat  Rh.dat  Sr.dat  V.dat
Au.dat  Cl.dat  Fr.dat  In.dat  N.dat   Pb.dat  Rn.dat  Ta.dat  W.dat
B.dat   Co.dat  Ga.dat  Ir.dat  Na.dat  Pd.dat  Ru.dat  Tb.dat  Xe.dat
Ba.dat  Cr.dat  Gd.dat  K.dat   Nb.dat  Pm.dat  S.dat   Tc.dat  Y.dat
Be.dat  Cs.dat  Ge.dat  Kr.dat  Nd.dat  Po.dat  Sb.dat  Te.dat  Yb.dat
```

  The data are converted from other variable which as far as I remember, called optical strench. The format is the energy in eV vs. cross section in Mbarns. The fragment for `Ar.dat`:

```
...
       15.9     32.42
    15.9249     32.48
    16.1825     33.04
    16.4442     33.61
    16.7102     34.18
    16.9805     34.77
    17.2551     35.31
    17.5342     35.85
    17.8178     36.39
     18.106     36.95
    18.3989     37.51
    18.6964     37.66
...
```

  The tables are started from 10 eV. The program extrapolate the table linearly to the left till the actual threshold, when it is lower than 10 eV.

- The file `shelllist.dat` is converted table of Gvin Williams [9] with added fluorescent rates extracted from [10]. For argon one can see there

```
  Atomic charge
    The number of shells
       The name of element
                The bound energy of given shell
                        The number of electrons at given shell
                                The probability of the fluorescence
                                        The comment mark (but it is also read)
```

```
                                          The shell principal quantum number
...
# 18 7  Ar
                3205.9  2       0.118   ! 1s
                326.3   2       0.00018 ! 2s
                250.6   2       0.00022 ! 2p1/2
                248.4   4       0.00022 ! 2p3/2
                29.3    2       0       ! 3s
                15.9    2       0       ! 3p1/2
                15.7    4       0       ! 3p3/2
...
```

Obviously, the sign # is used by computer for searching the specific atomic charge at initialization of the atom. The data after "!" were intended to be comments, but now the first number is also used and interpreted as the principal quantum number of the shell. So, please be careful when adding there anything.

- Theoretical calculations by Trzhaskovskaya for argon, krypton and xenon are in files ftb18.dat, ftb36.dat, and ftb54.dat respectively. The files ftbf18.dat, ftbf36.dat, and ftbf54.dat are the same files with added fluorescence rate from [10]. For example, the beginning of the file ftbf18.dat now looks like this:

```
Editted this file, added main shell numbers in shell names. (IS)
Editted this file again, added fluorescence rate
copying it from shellist.dat

 COMPLEX OF COMPUTER PROGRAMS RAINE BY BAND AND TRZHASKOVSKAYA
 PHOTOIONIZATION CROSS SECTION IN A DIPOLE APPROACH
 DIRAC-FOCK WAVE FUNCTIONS OF BINDING AND FREE ELECTRONS ARE USED
 HOLE-EFFECT IS NOT TAKEN INTO ACCOUNT

 INPUTE INFORMATION
 NUCLEAR CHARGE =  18
 K1SHELL
 BINDING ENERGY IS THEOR EBIND =     3.2416 KEV
 PHOTON ENERGIES (IN KEV) ARE

     3.2436    3.8924    4.6708    5.6050    6.7260    8.0712    9.6854
    11.6225   13.9470


         PHOTOIONIZATION CROSS SECTION
               DIPOLE APPROACH
     HOLE-EFFECT IS NOT TAKEN INTO ACCOUNT
  TERM DEPENDENT IS TAKEN INTO ACCOUNT
         Z = 18   1K1 SHELL EBIND =    3.2416keV

    PHOTON ENERGY (keV)   CROSS SECTION (Mbarn)

$
9 3.2416    0.118
          3.2436              0.4694E-01
          3.8924              0.4926E-01
          4.6708              0.3122E-01
          5.6050              0.1932E-01
          6.7260              0.1173E-01
          8.0712              0.7034E-02
          9.6854              0.4175E-02
         11.6225              0.2452E-02
```

```
         13.9470            0.1427E-02

 INPUTE INFORMATION
 NUCLEAR CHARGE =  18
 L1SHELL
 BINDING ENERGY IS THEOR EBIND =     0.3377 KEV
 PHOTON ENERGIES (IN KEV) ARE

     0.3397    0.4077    0.4892    0.5871    0.7045    0.8454    1.0145
     1.2173    1.4608    1.7530    2.1036    2.5243    3.0291    3.6350
     4.3620    5.2344    6.2812    7.5375    9.0450   10.8540   13.0248
    15.6297


         PHOTOIONIZATION CROSS SECTION
               DIPOLE APPROACH
...
```

From this one can infer that the "$" mark is used by computer to pass the preamble and to start reading at the number of points in the following table (9), the threshold energy (3.2416, it is computed in the `RAINE` program), the fluorescent rate (added manually) and two column table of 9 (in this case) cross section. Then the program looks for the next $ mark and reads data of next shell and so on.

- The argon data compiled by Marr and West [16] are given in the file mw3.dat. It is in the same format as the Henke converted files, so it is read by the same constructors. The data are very close to Henke's data except for M-shell of argon, for which the Henke data are slightly higher.
- The file `shelltsc.dat` contains the fit parameters of theoretical data computed (and fitted) by Band, Trzhaskovskaya at el. `shelltscf.dat` only the fluorescence rate is added at the last colunm.
- The numerical data for elastic scattering cross section [17] are given in file `cbdel.dat`.
- The file `elastic_disp.dat` contains results of calculations by `ElElasticScat::fill_hist_low_scat(void)` and allows to automatize the low angle elastic scatterings.
- The file `delta_path.dat` contains the distribution, which is used to calculate E_c from ref. [1]. This is just this distribution in a table form.

17 Matter, HEED definitions

17.1 Atoms and molecules

HEED needs photoabsorption and photoionization cross sections, which are defined in the file `heed++/code/PhotoAbsCS.h`.

The literature data on photoabsorption cross section are fragmentar and not always consistent. This class hierarchy is designed to gather them in a consistent library. The principle is ordinary: definition of an abstract class which defines the interface available for the rest of program, and definition of derived classes with this or that realization. To the contrary with `wcpplib/matter`, there is no any global "database" and no formal ban to duplicate these definitions (also there would not be sense in duplication). So these are simple classes determining photoabsorption cross sections for atomic shells, for atoms, and for moleculas. Also the atomic relaxation cascades are defined. The system requires some memory for keeping data, and some disk files with input information. It takes some time for initializations, so it is not intended to be used in a loops, but only for initial or interactive initializations. Interesting that in the Fortran version of HEED, the equivalent data structure kept in a common block was depending on energy mesh, which was fixed and constant for all data. Since the data are initially given with different meshes or sometimes without them at all, they need to be converted to this fixed mesh. But in this C++ version it was found possible to avoid this conversion. The data are kept and handled as is, without unnecessary conversions. If they are initially determined on a mesh, they are kept with this mesh. This improves precision of handling energies near ionization threshoulds.

In the interface the energy is assumed to be in MeV and the cross sections are given in Mbarns.

Class `PhotoAbsCS` determines just any cross section with a single threshold related to Z electrons. It is mainly used as shell cross section of a particular atomic shell. Sometimes it is also used to input the atomic cross section before its separation to single shells. The principal functions are

```
inline double get_threshold() const {return threshold;}
virtual double get_CS(double energy) const = 0;// energy in MeV, CS in Mbarns
virtual double get_integral_CS(double energy1, double energy2) const = 0;
                               // energy in MeV, CS in Mbarns * MeV
```

This class does not specify whether it is ionization of absorption cross section.

The following classes are derived from it.

Class `OveragePhotoAbsCS` is designed in order to check how strong the results are sensitive for the details of the cross sections. It is derived from `PhotoAbsCS` and can replace it. It computes the overage cross section as a mean of integral of the real cross section over the interval `width`. Then the problem is to compute the integral for `OveragePhotoAbsCS::get_integral_CS()`. This is done numerically with `step`, supplied in constructor.

Class `HydrogenPhotoAbsCS` represents the cross section of Hydrogen by one of the formulas [12].

Class `SimpleTablePhotoAbsCS` describes cross section determined by a table of a mesh. So there are really two tables. It has two different constructors. One reads two column tables from disk. The first colunm is energy in eV. The second is cross section in Mbarns. The second constructor reads the fit parameters published in one of the old papers by Band and Trzhaskovskaya, and computes the table currently with the local mesh

```
long q = 1000;
ener = make_log_mesh_ec(2.0e-6, 2.0e-1, q);
```

The leading zeros are then removed.

Class `PhenoPhotoAbsCS` described cross section determined by a phenomenological power formula for $E > T$

$$\sigma = f/E^p \tag{1}$$

where E is photon energy, T is threshold, usually the binding energy of this shell, p is power supplied at initialization and f is normalization factor computed according to sum rule:

$$f = T^p C_t Z(p-1) \tag{2}$$

where C_t is sum rule for one electron:

$$C_t = \frac{2\pi^2}{\alpha m_e}. \tag{3}$$

Now we consider other classes not derived from `PhotoAbsCS`.

Class `AtomicSecondaryProducts` defines the decay channels for atomic relaxation.

Class `AtomPhotoAbsCS` defines atom with many shells and thresholds. It defines a lot of functions like

```
virtual double get_ACS(int nshell, double energy) const = 0;
virtual double get_integral_ACS(int nshell,
double energy1, double energy2) const = 0;
```

and so on. Functions with suffix "ACS" are related to absorption cross section. Similar functions with suffix "ICS" give ionization cross section.

The following classes are derived from AtomPhotoAbsCS.

Class SimpleAtomPhotoAbsCS defines the atom with ionization only (and absorption equal to ionization). This class has an array of active pointers "DynLinArr< ActivePtr< PhotoAbsCS > > acs", which allows to use any shell cross sections derived from class PhotoAbsCS. Due to ownership semantic of DynLinArr and ActivePtr it holds own copies of them.

Class ExAtomPhotoAbsCS is similar but adds excitation below atomic ionization potential. Most of the atoms are represented by this class. The integral of excitation cross section is computed according to sum rule. But the form is unknown. It is assumed that the bulk of the excitation should be close to ionization potential, since the most of the excitation levels should be near it. Then the excitation is assumed to be distributed uniformly from $0.7I_{min}$ till I_{nim}. This may be very arbitrary and rough, but perhaps it is better than nothing. This class has a lot of constructors and able to deal with Henke, Marr, and Trzhaskovskaya data, as well as with combinations of them. Reading Henke or Marr tables, in which the shells are not separated, it separates the shells. The algorithm of separation is quite complicated and one should think twice before starting to edit it.

The simple example of initialization of the cross sections and their ploting is given in the program PhotoAbsCS_allAr_ts.c. It computes four different cross sections for Argon: by Henke [11], by Marr and West [16], calculated theoretically by Trzhaskovskaya [14, 15], and analitic fit of theopretical calculations by Verner et al., ref. [14]. As was explained earlier, it produces the histogram file photoabscs_allar_ts.hbook and alias file photoabscs_allar_ts.hbook.l. Kumac file photoabscs_allar_ts.kumac produces the encapsulted PostScript figure photo_allar.eps, which shows all these curves in different scales (the region around K-shell edge is increased). This curve is shown here at fig. 1.

Another plot of photoionization cross section by Henke and Trzhaskovskaya is shown in fig. 2

Figs. 3 and 4 illustrate the shell separation. The idea of separation is based on an assumption that the partical cross sections fall with constant rate (which was assumed to be $1/E^{2.75}$). This allows to subtract them one by another passing from low energies to higher ones. The ionization potentials and the amount of electrons are also taken into account. For example, if the ionization potensials are close, the "weights" of sub-shells can be assumed to be proportional to the number of electrons. But sometimes this algorithm fails to separate subshells. For example, it interference effects destoy this simple model, as occurs for M-shell of argon, the sub-shells are not separated well enough. But the shells are always separated, and this is probably enough for our purposes.

These two figures show also the assumed excitation contribution as a flat additions at low energies .

Class MolecPhotoAbsCS defines the cross section for moleculas. It contains array DynLinArr< PassivePtr< AtomPhotoAbsCS > > atom, which allows to handls any atoms.

The library of the ready cross sections are defined in files heed++/code/PhotoAbsCSLib.h and heed++/code/PhotoAbsCSLib.c. As was explained the .c file first includes wcpplib/matter/GasLib.c to provide prior initialization of general atomic information. Then it initializes the necessary names:

```
char* a_internal_HDB;
String shelllist_dir_name =
String((a_internal_HDB = getenv("HEED_DATABASE")) == NULL ? "" :
       a_internal_HDB ) + "/";
```

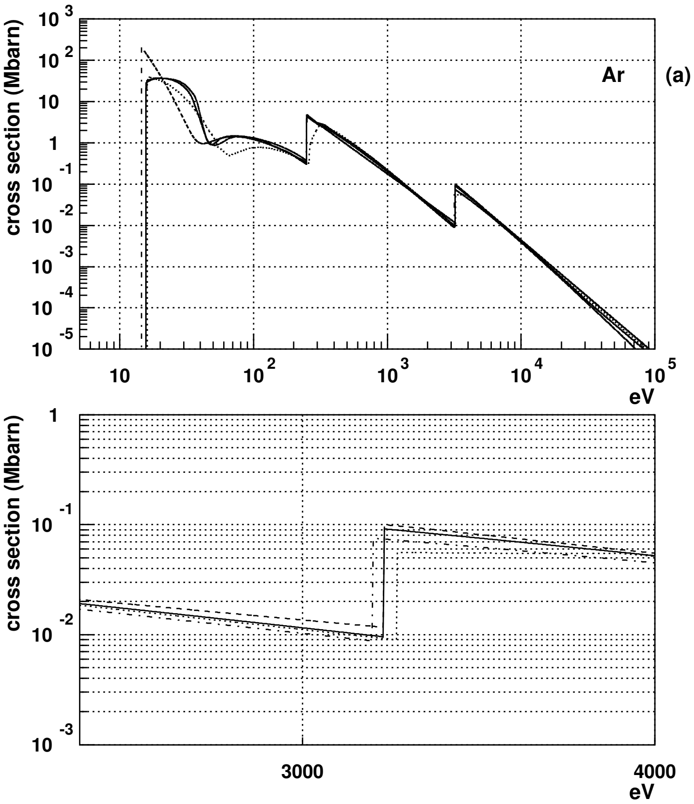

Figure 1: The photoionization cross section for argon(a) according to from Henke tables (solid lines), Marr and West (dashed lines), and calculated theoretically by Trzhaskovskaya [14, 15] (dotted lines) and analytic fit by Verner et al[14] .

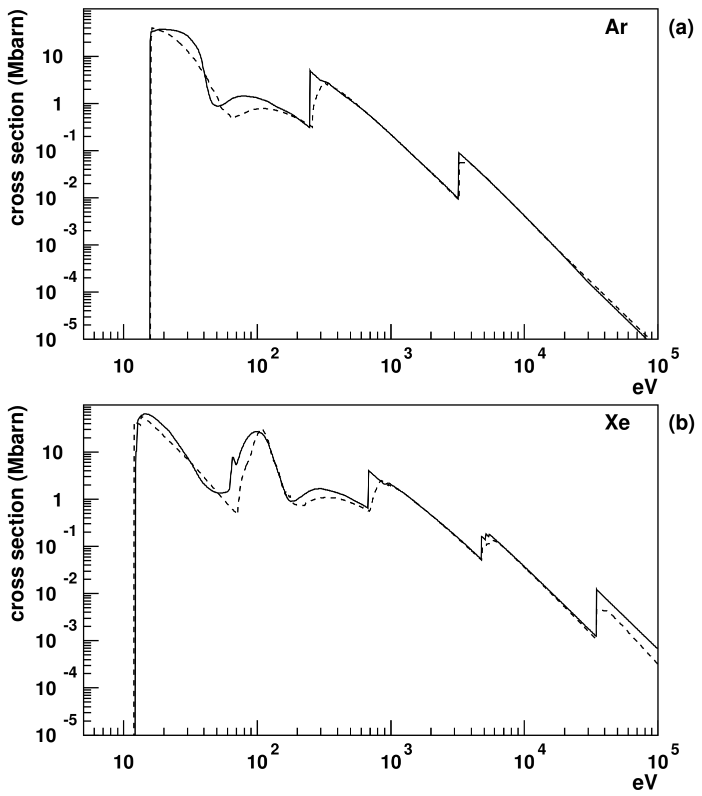

Figure 2: The photoionization cross section for argon(a) and for xenon(b) from Henke tables (solid lines), and calculated theoretically by Trzhaskovskaya [14, 15] (dashed lines).

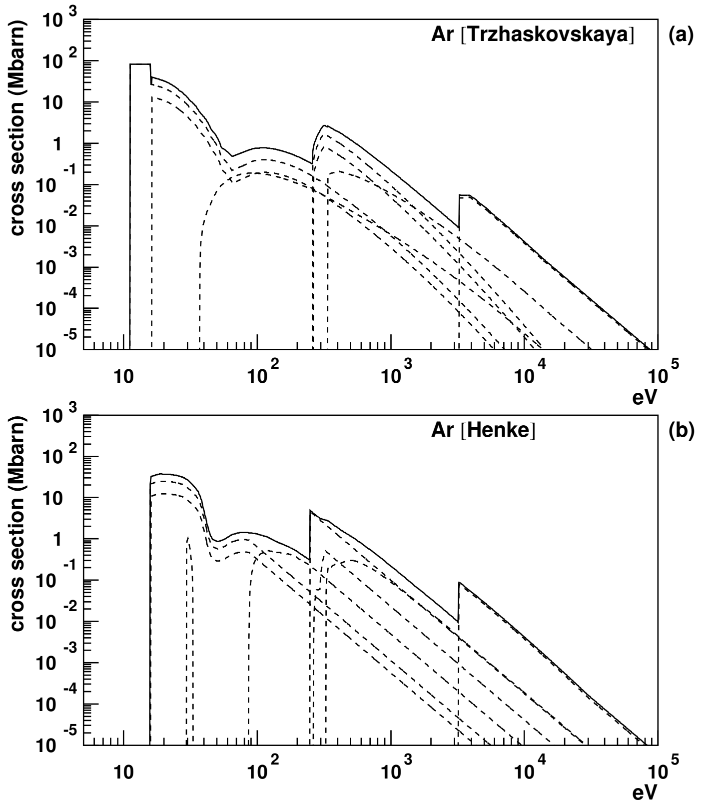

Figure 3: The photoabsorption cross section of argon theoretical from [15] (a) and compilation of Henke [11]. The solid lines are the total cross sections, the dashed are partial ones.

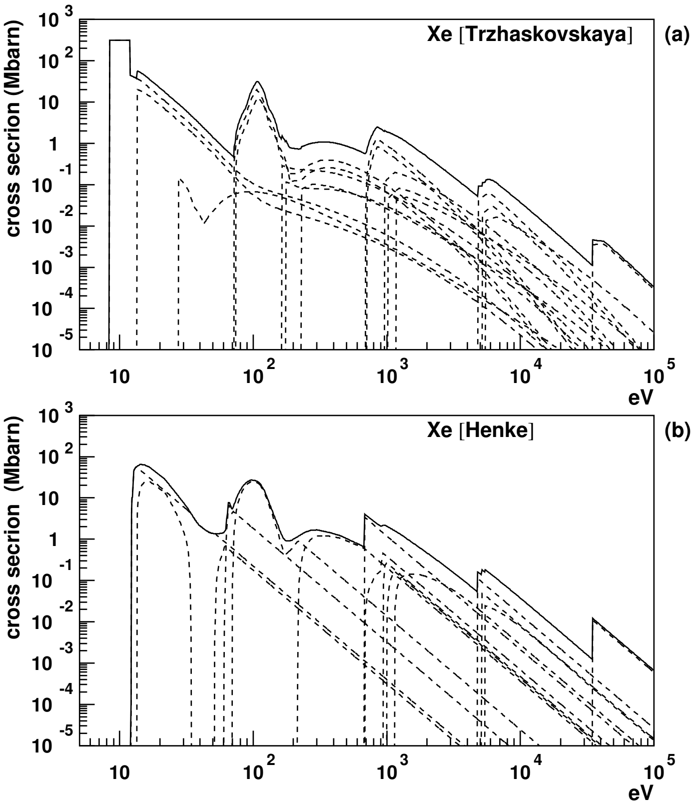

Figure 4: The photoabsorption cross section of xenon theoretical from [15] (a) and compilation of Henke [11]. The solid lines are the total cross sections, the dashed are partial ones.

```
String pacs_table_dir_name =
String((a_internal_HDB = getenv("HEED_DATABASE")) == NULL ? "" :
       a_internal_HDB) + "/henke/";
```

And then it initializes cross sections of many atoms like this:

```
ExAtomPhotoAbsCS Argon_PACS(18,
    shelllist_dir_name + "shelllist.dat",
    pacs_table_dir_name + "Ar.dat");
MolecPhotoAbsCS Ar_MPACS(Argon_PACS, 1, 26.4e-6);
```

Some atoms have alternative commented options. The preferred by the opinion of the author variants are uncommented. Of course, the user can change this, but it is not recommended to do off-hand: this may change strongly the results of calculations.

17.2 Enegy mesh

Internal calculations in HEED is convenient to perform with some standard energy mesh. This mesh should be logarithmic or similar, with dense points at little energies and sparse at large energies. The class `EnergyMesh` determines such mesh and some standard functions, namely getting center of energy interval `get_ec(long n)`, left edge of interval `get_e(long n)` (right edge is left one for the next bin), and some other simple functions. There may be many meshes in program (but currently only one mesh is used in a single program, different mashes may be used for totally different matters and cross sections). The pointer to mesh should be given as the parameter of class constructor, when a class depends on mesh.

The class keeps the left sides of bins and their centers in arrays. Since the right side of interval is the left of the next one, the array keeping the left sides should be by one larger than the dimension of the mesh.

For speed the internal arrays of the class `EnergyMesh` keeping the right sides of intervals and their centers are defined as simple plain arrays with dimension const int pqener = 1001. The actual size of mesh can be 1000 or less. The violation of boundaries triggers error and termination.

The energy mesh in HEED is defined in MeV.

The typical initialization looks like this:

```
EnergyMesh energy_mesh(2.0e-6, 2.0e-1, 1000);
```

The argumants are the minimum, maximum and the quantity. For good precision the quantity may be from 200 to 1000.

17.3 Matters

Class `HeedMatterDef` decribes the materials (gases) for HEED. It contains the passive references to matter from wcpplib, to `AtomPhotoAbsCS` objects and to `EnergyMesh`:

```
PassivePtr< MatterDef > matter;   // MatterType is not used for
                                  // similarity with the next line
DynLinArr< PassivePtr<AtomPhotoAbsCS> > apacs;
// Each element of this array corresponds to component of matter
PassivePtr< EnergyMesh > energy_mesh;
```

It computes the total absorption and ionization cross sections, the dielectric permeabilities

```
DynLinArr< double > epsi1;  // real part of dielectric constant (e_1 - 1)
DynLinArr< double > epsi2;  // imaginary part of dielectric constant.
```

The example is given in fig. 5.

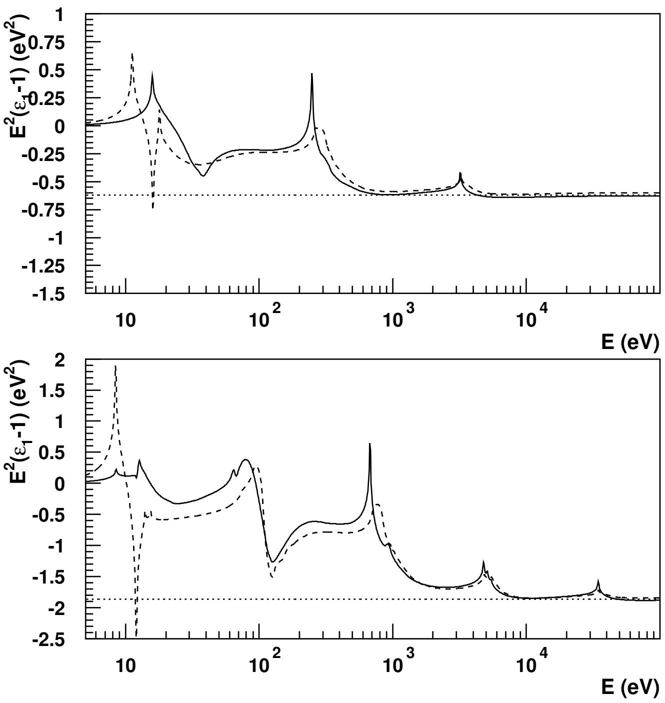

Figure 5: The dielectric permeability for argon (a) and kcenon (b) ϵ_1, shown in form $E^2(\epsilon_1 - 1)$, computed by cross sections of Henke [11], fig. 3(b) and 4(b) (solid lines) and by theoretical cross sections [15] fig. 3(a) and 4(a) (dash lines) and plasm dielectric permeability (dotted lines).

17.4 Cross section of the energy transfer

Class `EnTransfCS` computes the cross section for given incident particle mass and velocity and for given `HeedMatterDef`, to which it has passive reference.

```
PassivePtr< HeedMatterDef > hmd;
```

Thus the particle is not expected to be slowed down. It would be of course important to provide this, but now it is not still provided. It is assumed that the energy loss of fast particle in a gas volume in negligible compared with its initial energy. This is reasonable approximation for most of the gaseous detectors. The constructor of this class look by the following way:

```
EnTransfCS(double fparticle_mass, double fgamma_1,
           int fs_primary_electron,
           HeedMatterDef* fhmd, long fparticle_charge=1);
```

So the user has to provide the particle mass, its $\gamma - 1$ ("-1" is used in order to avoid the loss of precision at low speed), signature whether it is electron (then the Rutherford term would be different, see ref. [1])), and the pointer to the matter. If the particle charge differs from ± 1, the user can also provide its value. The cross section is assumed to be scaled as z^2. (This is precisely so according to PAI, but there are messages in the literature that this may be perfectable).

The example of the cross section for the energy transfer is given in fig. 6. An example program `EnTransfCS_sim_ts.c` prints the number of primary clusters at ionization minimum.

The cross section if used in class `HeedParticle` in order to compute by random sampling the number of interactions with each shell and the energy transferred for each interactions. The internals of the class `EnTransfCS` remind the old Fortran HEED. There is some redundance in kept data: the most of internal arrays are necessary for debug only. The main useful members are the following. The class member

```
DynLinArr< DynLinArr< double > > quan;  // per 1 cm, used for path length
```

gives the mean number of interactions with each shell, which is to be supplied to Poisson distribution. Here the first index denotes the atom and the second the shell. The class member

```
  DynLinArr< DynLinArr< DynLinArr< double > > > fadda; // integral
```

is the integrated cross section of energy transfer. It is used as argument of the function

```
    double r = t_hisran_step_ar< double, DynLinArr < double >,
      PointCoorMesh< double, const double* > >
      (pcm_e, aetcs->fadda[na][ns], rn);
```

(`t_hisran_step_ar` is defined in `wcpplib/math/tline.h` and the citation of from `heed++/code/HeedParticle.c`) which gives the random energy in MeV. The integration is performed by the sentence

```
val_fadda[na][ns] = t_hispre_step_ar< double, DynLinArr < double >,
  PointCoorMesh< double, const double*  > >
  (pcm_e, fadda[na][ns], fadda[na][ns]);
```

from `heed++/code/EnTransfCS.c`.

In order to use this class for Monte-Carlo calculations in which the gamma of initial particle can be random, it needs to be able to calculate the objects of this class for a frequent mesh of incident particle's $\beta\gamma$. This is however difficult to arrange if to compute and to keep all the data which was originally put in it.

I note that this program was studied and used in the mode with a single $\beta\gamma$. If it needs to simulate for example an experiment with $\beta\gamma$-dependence of ionization, these $\beta\gamma$ are simulated in queue, one by one. So just one object of class `EnTransfCS` was kept each given moment. This object was initialized once for and before, say, a million events, when the particle with this $\beta\gamma$ run through the detector a million times. Thus both the space and time expenses are negligible. This is not the case if you want to use this program for generic Monte-Carlo calculation, in which each next particle has different $\beta\gamma$ (and even different type). Then you have either to compute the cross section object for each new paricle, which will be time expensive, or to compute a mesh of cross section and, say, to interpolate between points. Interpolation is OK, but the computation of the mesh could be not fast enough and the mest could consume too much memory. The measures discussed below address these issues.

To reduce the size of these objects four macros are included. Activating them leads to removal from the object unecessary for MC data and calculations and makes them appropriate for the general MC application.

```
#define EXCLUDE_A_VALUES  // exclude absorption values
#define EXCLUDE_VAL_FADDA  // exclude unecessary for MC values
#define EXCLUDE_MEAN  // exclude calculations of means, which are
// unnecessary for MC

#define CLEAR_ARRAYS
```

In order to refere to particular cross section we will use a simple class `EnTransfCSType`, which is defined by the following simple way:

```
class EnTransfCSType
{public:
  PassivePtr<EnTranfCS> etcs;
  EnTransfCSType(void): etcs() {;}
  EnTransfCSType(EnTranfCS* md): etcs(md) {;}
};
```

If the user wants this cross section to be associated with a particular volume, he should define the volume as derived from, in particular, this class `EnTransfCSType`. Obviously, the object of this class "knows" the address and allocation of cross section object `EnTranfCS`. The

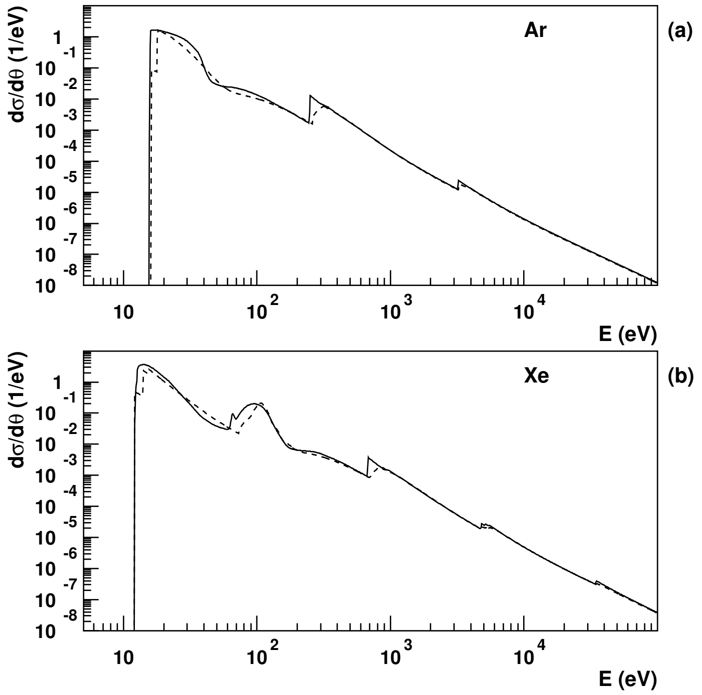

Figure 6: The cross section of the energy transfer in th nimimum of ionization for argom(a) and xenon (b) calculated with photoabsorption cross section of Henke [11] (solid lines), and by [15] (dashed lines).

latter can be allocated anywhere in the program. I usually define and initialize it simply in the main program. The incident particle type should be `HeedParticle`. The user can check that `void HeedParticle::physics(void)` tries to convert dynamically the type of the pointer to volume object to the pointer to an object of class `EnTransfCSType`. If this convertion is successful, this means that this particle should interact with the matter of the volume as HEED assumes. Otherwise this is empty or some non-significant volume interaction with which are not simulated:

```
void HeedParticle::physics(void)
{
  ...
  const absvol* av = currpos.G_lavol(); // get least address of volume
  const EnTransfCSType* etcst = dynamic_cast< const EnTransfCSType*  >(av);
  if(etcst == NULL)
  {
    return;
  }
  else
  {
    EnTransfCS* aetcs = etcst->etcs.getver();
    HeedMatterDef* ahmd = aetcs->hmd.getver();
    MatterDef* amatter = ahmd->matter.getver();
    EnergyMesh* a_energy_mesh = ahmd->energy_mesh.getver();
    ...
```

17.5 Cross section of the energy transfer for mesh of incident $\beta\gamma$

The considerations outlined in previous section justified the special class which contains an array of cross-sections each for single $\beta\gamma$. The class is named `EnTransfCS_BGM`. The "BGM" means

Beta-Gamma Mesh. This class has (passive) pointers to matter (PassivePtr< HeedMatterDef > hmd) and $\beta\gamma$-mesh (PassivePtr< BGMesh > mesh).

In order to provide linear interpolation of cross-sections, it needed to modify the alrorithm of interactions. Therefore a special type of particle was introduced: HeedParticle_BGM. The user can look into their codes and see that the classes HeedParticle and HeedParticle_BGM are very similar in many respects.

17.6 δ-electron elastic scattering

The physics of this modeling is to be described later. Here we just suggest that for correct functining the user should add in his program the following lines:

```
String database_dir_name =
  String((a_internal_HDB = getenv("HEED_DATABASE")) == NULL ? "" :
 a_internal_HDB ) + "/";
...
String file_name= database_dir_name + "cbdel.dat";
ElElasticScat ees(file_name);   // large angle scattering
ElElasticScatLowSigma eesls(&ees, database_dir_name + "elastic_disp.dat");
// ^ low angle scattering
```

17.7 Pair production by δ-electrons

We postone the detailed description as well. For correct functioning the program should contain the following sentences:

```
String database_dir_name =
  String((a_internal_HDB = getenv("HEED_DATABASE")) == NULL ? "" :
 a_internal_HDB ) + "/";
...
String pairprod_file_name = database_dir_name + "delta_path.dat";
PairProd pairprod(pairprod_file_name, W*1.0e6, F);
```

17.8 Full δ-electron physics

For the total initialization of δ-electron physics one has to issue the following commands:

```
HeedDeltaElectronCS hdecs(&HeedGas, &ees, &eesls, &pairprod);
```

where HeedGas is parameter of HeedMatterDef type, and the other three parameters are defined above in two previous sections.

We will need also a tool to refere to class HeedDeltaElectronCS:

```
class HeedDeltaElectronCSType
{public:
  PassivePtr< HeedDeltaElectronCS > hdecs;
  HeedDeltaElectronCSType(void): hdecs() {;}
  HeedDeltaElectronCSType(HeedDeltaElectronCS* md): hdecs(md) {;}
};
```

18 The real volumes

When particle traces particles, it should know all these cross sections for each geometrical volume. To let the program know all this information we gather it like this:

```
class Chamber: public sh_manip_absvol, public box,
       public EnTranfCSType,
       public HeedDeltaElectronCSType,
       public SensitiveVolume
```

```
{public:
  virtual absvol* Gavol(void) const {return (box*) this; };
protected:
  virtual void get_components(ActivePtr<absref_transmit>& aref_tran)
    {
      sh_manip_absvol::get_components(aref_tran);
    }
public:
  Chamber(void) {;}
  Chamber(const abssyscoor& fcsys,
  const EnTranfCSType etcst,
  const HeedDeltaElectronCSType hdecst);

  macro_copy_total(Chamber);
  virtual void print(ostream& file, int l) const ;

};
```

This is a declaration of a class, an object of which will know the cross sections for incident particles and for δ-electrons. How it is explained in ref.[8], the geometrical system allows to gather the geometrical information and the material data in a single object.

The shape of this volume is box, which is naturally described by class `box`. The box, as well as any other shape, is defined in own system of coordinates. Therefore it should always be accompanied by class `sh_manip_absvol`, which contains the particular system and functions to deal with it (or by `manip_absvol` if the system is defined somewhere else). The member function `get_components` is a standard function of all geometrical objects needed for provision of their defauld transformations at the passage between different system of coordinates, shifts or rotations. Here we have to geometrical classes: `sh_manip_absvol` and `box`. The both classes have this function as well, and for successful compilation we have to determine it in `Chamber` as well. We also remark that since the `box` class cannot be transformed, its instance of `get_components` issues an error message and terminates the program. Then only `sh_manip_absvol` is true transformated class here. The transformation of the system of coordinates, which it contains, will lead to correct indirect transformation of the box. Therefore the only this which `get_components` is doing here, is to call `sh_manip_absvol::get_components(...)`.

The function

```
  virtual absvol* Gavol(void) const {return (box*) this; };
```

is defined in `manip_absvol` (`sh_manip_absvol` is derived from `manip_absvol`) and is used by the functions of `manip_absvol` to access the real volume. Not wondering that here it just returns the address of the total object, which is derived from `box`, and thus *is* `box`. Note that it is not necessary that the real volume description were right here. It can be allocated independently, which is reasonable to do, if there are very many volumes of the same shape and with the same internals.

The classes `EnTranfCSType` and `HeedDeltaElectronCSType` are described above.

The class `SensitiveVolume` is a class, which means that the program should remember the conduction electrons deposited in it. It is located in file `heed++/code/HeedCondElectron.h`, and so short that we can reproduce it right here:

```
class SensitiveVolume
{public:
  BlkArr< HeedCondElectron > conduction_electron_bank;
  SensitiveVolume(void) {;}
};
```

The blocked array `conduction_electron_bank` is a place where the program will safe the coordinates of conduction electrons, and where the user can later take them.

The notation `macro_copy_total(Chamber)` means a macro defined in `wcpplib/satetl/AbsPtr.h`:

```
#ifndef COPY_RETURNS_COMMON_BASE
...
#define macro_copy_total(type)  virtual type* copy(void) const \
{return new type(*this);}
...
#else
...
#define macro_copy_total(type)  virtual CommonBase* copy(void) const \
{return new type(*this);}
...
#endif
```

This is a cloning function, which is necessary in order to address this object by active pointer. The function definition is synchronized with the method of bypassing the "covariant return" problem at old GNU compilers. There are many other variants of this function, see this file. So if we copy active pointer to object of class `Chamber`, the object will be cloned and the new instance of "`Chamber`" will be created.

Such Chamber can also be copied by assignment of copy constructor. The new object will contain new `sh_manip_absvol`, new `box`, new reference to `EnTranfCS` and `HeedDeltaElectronCS`, and new `SensitiveVolume`.

The real example is in file `Spectr_ts.c`. In order to initialize the object of this type, the user can issue following commands

```
  fixsyscoor prim_sys(point(0,0,0), basis("primary"), "primary");
  Chamber ch(prim_sys, &etcs, &hdecs);  // chamber
  ch.print(mcout, 1);   // just print to see what happen
```

It is interesting that this system differs conceptually from that of GEANT3 and, as far as the author is informed, GEANT4. In them the volumes are initialized by a function calls and stored somewhere inside the internal data-banks. Here the volumes are declared as classes and initialized at initialization of objects of these classes. Therefore here volumes are directly accessible. Any data can be readout, any their virtual function reloaded etc. This seems to be more "object-oriented" system, then the approach of GEANTs. However the initialization is perhaps more long in this approach.

In addition, in order to allow the user to initialize the volumes from Fortran, a method of indirect initializations through functions was also developed. The necessary declarations are in files `wcpplib_int.h` and `wcpplib_int.c`. Here the freedom of user is restricted. The class is however quite complicated, see sect. 22.2.

19 The real example of volumes initialization

In the following example we initialize a volume consisting of 128 adjacent sub-volumes of 6 cm width each. First we initialize sub-volume without embraced volumes, called "cell":

```
const long q_cell = 128;
const double cell_width = 6 * cm;

class Cell: public sh_manip_absvol, public box,
    public EnTransfCSType,
    public HeedDeltaElectronCSType,
    public SensitiveVolume
{public:
  long ncell; // number of this cell
  virtual absvol* Gavol(void) const {return (box*) this; };
protected:
  virtual void get_components(ActivePtr<absref_transmit>& aref_tran)
    {
```

```
      sh_manip_absvol::get_components(aref_tran);
    }
public:
  Cell(void) {;}
  Cell(const abssyscoor& fcsys,
       const EnTransfCSType etcst,
       const HeedDeltaElectronCSType hdecst, long fncell);

  macro_copy_total(Cell);
  virtual void print(ostream& file, int l) const ;
};
Cell::Cell(const abssyscoor& fcsys, const EnTransfCSType etcst,
   const HeedDeltaElectronCSType hdecst, long fncell):
  sh_manip_absvol(fcsys),
  box(60.0*mm, 1000*mm, cell_width, String("cell_") + long_to_String(fncell)),
  EnTransfCSType(etcst),
  HeedDeltaElectronCSType(hdecst),
  ncell(fncell)
{
  mfunname("Cell::Cell(const abssyscoor& fcsys, const EnTransfCSType* etcst)");
}
```

Then we define the chamber from many cells:

```
class Chamber: public sh_manip_absvol, public box,
       public EnTransfCSType,
       public HeedDeltaElectronCSType
{public:
  Cell cell[q_cell];
  virtual absvol* Gavol(void) const {return (box*) this; };
protected:
  virtual void get_components(ActivePtr<absref_transmit>& aref_tran)
    {
      sh_manip_absvol::get_components(aref_tran);
    }
public:
  virtual DynLinArr< manip_absvol * > Gamanip_embed(void) const ;
  Chamber(void) {;}
  Chamber(const abssyscoor& fcsys,
  const EnTransfCSType etcst,
  const HeedDeltaElectronCSType hdecst);

  macro_copy_total(Chamber);
  virtual void print(ostream& file, int l) const ;

};
Chamber::Chamber(const abssyscoor& fcsys, const EnTransfCSType etcst,
 const HeedDeltaElectronCSType hdecst):
  sh_manip_absvol(fcsys),
  box(60.0*mm, 1000*mm, q_cell * cell_width, "chamber"),
  EnTransfCSType(etcst),
  HeedDeltaElectronCSType(hdecst)
{
  mfunname("Chamber::Chamber(const abssyscoor& fcsys, const EnTransfCSType* etcst)");
  long n;
  for(n=0; n<q_cell; n++)
  {
    double z =  - 0.5 * q_cell * cell_width  + 0.5 * cell_width +
      n * cell_width;
    cell[n] = Cell
```

```
      (fixsyscoor(point(0.0, 0.0, z),
                  basis(),
                  String("cell_") + long_to_String(n)),
       etcst, hdecst, n);
  }
}
DynLinArr< manip_absvol * > Chamber::Gamanip_embed(void) const
{
  DynLinArr< manip_absvol * > internal(q_cell);
  long n;
  for(n=0; n<q_cell; n++)
  {
    internal[n] = (manip_absvol*) (&(cell[n]));
  }
  return internal;
}
```

The function `Gamanip_embed` returns the pointers to embraced volumes, that is the cells in the given case. The constructor of `Chamber` shows how the `Cell`s are positioned with shift relatively the primary center of coordinates.

Then we can trivially access the cells from the program. For example, to extract the number of cunduction electrons, deposited in them in a given event, we use

```
      ...
      Chamber ch(prim_sys, &etcs, &hdecs);
      ...
      for( nev=0; nev<qev; nev++)
      {
        ...
        for(ncell=0; ncell<q_cell; ncell++)
        {
          qcel[ncell] = ch.cell[ncell].conduction_electron_bank.get_qel();
        }
        ...
        }
```

We can also copy the total `Chamber` to another place, and all the internals will be also correctly copyied. So this is very consistent object-oriented system.

20 The incident particle

The incident particle is a class `HeedParticle` from file `heed++/code/HeedParticle.h`. Its constructor shows, what the user need to supply to it:

```
  HeedParticle(manip_absvol* primvol, const point& pt,
       const vec& vel, vfloat time, particle_def* fpardef,
       int fs_loss_only = 0,
       int fs_print_listing = 0);
```

It is the volume manipulator (`manip_absvol* primvol`, see sect. 13 and ref. [8]), the initial point (`const point& pt`), the vector of velocity (`const vec& vel`), the current time (`vfloat time`), the particle type definition (`particle_def* fpardef`, see sect. 14), the signature whether it needs to model the primary interactions only (`int fs_loss_only = 0`), and the sign of debug printing (`int fs_print_listing = 0`).

The class `HeedParticle` is derived from class `eparticle`. The main function of them is function `fly()`, which makes the particle moving and interacting.

The secondary particles are saved in

```
extern AbsList< ActivePtr< gparticle > > particle_bank;
```

All the latter can be treated by the function

```
void treat_particle_bank(void);
```

declared in the same file.

21 The full program

We have described all the key components. Now it is nothing to add becides referring to a simple exemple of the full program: `heed++/code/Spectr_ts.c`.

This program generates the ionization distributions in a layer of a given width in a given gas for a given particle. Currently it is Argon(93%) and CH_4(7%). Chamber width is 1.5 cm. Mass of the particle is 140 MeV. Momentum of 3000.0 MeV/c. Cross section is computed with assumption that it is not electron. The charge of the particle is set to +1. The user-physicist can guess what sort of the particle is actually meant under these conditions, but the actual type is not important for the ionization loss calculations.

The program can take into account the device resolution. Currently the resolution is set as 0.36 keV at 2.5 keV. (These numbers are taken from one particular experiment.)

The program generates the PAIR dirtributions as well as PAI ones. The PAI energy-loss distribution without the device resolution is accumulated in a histogram denoted in the program as `hist_total_eloss`. The same with device resolution is `hist_total_seloss`. The PAIR dirtributions plotted in energy-loss units (each event is obtained by multiplication of quantity-of-conduction-electrons by work-per-pair-production w) are respectively `hist_meas_eloss` and `hist_meas_seloss`.

Number of conduction electrons on event-by-event basis are filled in `hist_qcel`.

The program generates also the longinudinal space distributions and transverse distributions of the "center of gravity" of conduction electrons. Space distributions of these centers are accumulated in `hist_space_x(y,z)`.

The package should have pre-computed files

- `spectr_ts.hbook` – PAW-formatted histograms.
- `spectr_ts.hbook.l` – listing of paw macroses allowing to operate with histogram names.
- `spectr_ts.ghist` – GHist-formatted histograms.
- `spectr_ts.ghist.l` – listing of histograms for conveniency of the user.
- `Spectr_ts.eout` – textual output.

As explained somewhere earlier in this manual the user can quickly check the functioning of the newly installed program by changing

```
String taskname="spectr_ts";
```

in `Spectr_ts.c` to something like

```
String taskname="spectr_ts1";
```

Then one can run

```
shell>Spectr_ts.e > ttt.txt;
shell>diff ttt.txt Spectr_ts.eout.
```

Normally one should not see any difference except the difference in file names (particularly those depending on HBOOK or GHist mode). But in general it is not guaranteed that subtle differences in floating point operations implemented in particular processor (usually it can expected to have different precision of calculations inside mathematical co-processor and rounding at output from mathematical co-processor at different stages of calculations) would lead to shift of random number sequence and changing of all statistical fluctuations.

Note two lines at the end of this listing:

```
mean_total_eloss/MeV=0.00342878
mean_total_eloss/MeV/W=129.752 mean_qcel=120.026
```

This `mean_total_eloss` is mean of energy transfers (in MeV), computed in the limits defined by energy mesh:

```
EnergyMesh energy_mesh(2.0e-6, 2.0e-1, 200);
```

Increasing the upper linit (now $2.0 \cdot 10^{-1}$ MeV) would increase `mean_total_eloss`. This should not affect strongly at the mean number of conduction electrons `mean_qcel`, since the latter corresponds to absorbed energy, whereas the right tail of the energy transfer distribution has a trend to leak from thin gas volumes. The amount of this leakage can be guessed from comparison of `mean_total_eloss/MeV/W=129.752`, which is expected (but not realised) mean number of conduction electrons computed as simple ratio of transferred energy and work per pair production, and the actual mean number of conduction electrons `mean_qcel`, occured in detailed simulations.

To proceed with checks in the case of HBOOK-PAW mode of operation one could run

```
diff spectr_ts1.hbook spectr_ts.hbook
diff spectr_ts1.hbook.l spectr_ts.hbook.l
```

In the case of GHist:

```
diff spectr_ts1.ghist spectr_ts.ghist
diff spectr_ts1.ghist.l spectr_ts.ghist.l
```

To see plots do the following.

In HBOOK-PAW mode run pawX11. In its shell run

```
exe spectr_ts.kumac
h/many meas_seloss_GasAr_93_CH4_07 meas_eloss_GasAr_93_CH4_07
```

And you will see fig. 7.

In GHist-ROOT mode run root. In its shell run

```
.L libRootMGHist.so
RootMGHist r("spectr_ts.ghist");
meas_seloss_GasAr_93_CH4_07.Draw();
meas_eloss_GasAr_93_CH4_07.SetLineStyle(2);
meas_eloss_GasAr_93_CH4_07.Draw("same");
```

And you will see fig. 8.

Please see manuals of PAW and ROOT for mamy commands and options allowing user to change the style of figures.

The program `Spectr_ts.c` shows how to define and initialize detectors and everything what were described (or missed) in this manual. It might be useful for many applications as is. It may also be used as a “fish” to develop more complicated programs.

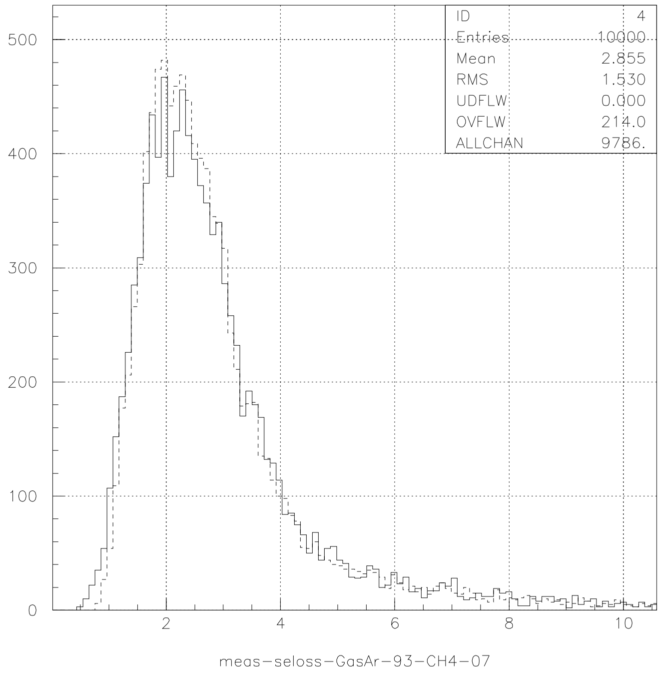

Figure 7: The example of ionization effect of Ar (93%) and CH_4 mixture with (solid line) and without (dashed line) assumed device resolution. Produced in HBOOK-PAW environment.

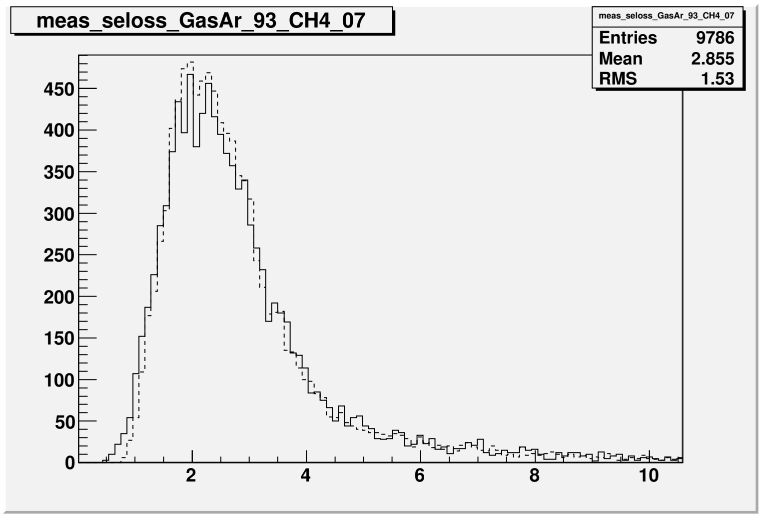

Figure 8: The same as at the previous plot, but produced in GHist-ROOT environment.

22 The interface with Fortran-77

Since most of the users of old HEED have called it through Garfield written in Fortran-77, the author has studied whether it is possible to call the new program by the same or similar way as the old program was called, having in mind its possible connection to Garfield as well and replacement the old program by this new. The users, who are not interested in this matter, can skip this chapter totally.

We will consider here such an interface which may be used in an interactive session. So the C++ program does not have any a-priory data about the task to be solved and receive all the information from Fortran. (Otherwise it would be rather strange to use this interface, rather then just encode the task in C++.)

The strategy of this package is the provision of all main initializations, deletions, and actions by the calls of pseudo-Fortran subroutines. Actually these subroutines are written not in Fortran but in C++. However, they are made callable from Fortran by some special declarations and allocation of information.

To provide correct link and work of this package, the main program should be written in C++ and the link should be performed through the C++ linker. (The case of Fortran linker plus Fortran main would not work, whereas the mixed cases were not tested). Usually, in any Fortran program the main program unit can be declared as subroutine, and called from short "formal" C++ function int main(void). This assures correct link and execution of all global initializations in C++, whereas it does not make any harm to functioning of Fortran algorithms.

All the files of this package reside in directory `heed++/forinterf`.

22.1 Allocation and identification of data

When the user manipulating with Fortran program requests any initializations of C++ objects that were not allocated previously, they can be allocated in the heap memory and the pointers can be saved in global variables. Then, the return of the control from C++ to Fortran does not do any harm to these objects. They are kept and can be used later at the next calls of C++, but they can not be accessible from Fortran directly.

Then there is a problem how to denote, to mark, or to refere this or that C++ object from Fortran. The C++ identifiers would not work in Fortran. Quite many objects in HEED-C++ are uniqiuly identified by literal names. But this is also not very useful feature for providing interface since the exchange of characters between Fortran and C++ is troublesome and may be not portable enough. The most reasonable option is the use of indexes. Therefore in the interface package all the objects that are kept in C++ and could be referred in Fortran are enumerated. Technically, it is performed by putting the pointers to objects in an array. The objects are then made accessible from Fortran through such arrays of pointers.

Many C++ objects are defined as static global objects. The others need to be initialized dynamically in heap memory during interactive dialog, and the pointers are expected to appear in the same array. The dynamic objects can be deleted also dynamically, and the same indexes can be assigned to newer objects. This creates difference in manipulating with pointers, since the static objects can not be deleted. Therefore the class containing this array should know which object is "heap", and which is global. The first could be deleted, the second is not. The whole functionality of this class is somewhat tedious, therefore it was written once as a template class with type corresponding to the type of handled objects. It is applicable to any types of objects. This template class have long name "`InterfaceExtensibleContainer`" and is widely used in the interface. Its first `q_static_elements` are static. The value `q_all_elements >= q_static_elements`. The additional elements are not obliged to have sequential indexes. When requesting initialization, the user is expected to supply the index of the new element. If this index is already occupped, the program generates exception and returns the control to Fortran. If this index is more than the current size of this array denoted by `q_all_elements`, the array is extended.

The data member of this class are clear:

```
template<class X>class InterfaceExtensibleContainer
{public:
  long q_static_elements;  // number of static elements,
  // these elements cannot be deleted.

  long q_all_elements;  // should be equal to dimensions of arrays below
  // Number of all elements including the static ones.

  DynLinArr<String> element_name;  // character name of elements
  DynLinArr< PassivePtr<X> > element; // pointers to elements
  ...
};
```

Actually, as we see, this is not one array, but two arrays. The first contains literal names, and the second pointers. If the given element is static and have the type whose objects are identified in HEED by their literal names, the pointer to this element may be not initialized at all, since the object can be located by literal name. On the other hand, all the pointers to heap objects must be initialized so as to assure their deletion, even if they are identified by names. For the objects having no literal names the content of the first array with names may be not initialized, but then the second array should be initialized in whole. The `NULL` pointer indicates that the corresponding index is vacant. In addition, so as to debug his interactive script or program the user can print to output stream all elements of particular container or any element with given index. The objects of class InterfaceExtensibleContainer cannot be copied.

The elements that are a-priory initialized, such as the most atoms and some moleculas, can be identified in fortran through so-called parameters or named constants with simbolic names. It is strictly recommended, since the list of these elements can be extended and their numerical indexes changed in the next versions, whereas the literal names are expected to be stable. The values of Fortran parameters with these simbolic names are strictly correspond to C++ indexes, with which these elements are registered in `InterfaceExtensibleContainer` objects. Namely the Fortran definitions are given in the file `interfdata.inc`. C++ definitions are in `interfdata.h`. Currently the initialization of new atoms and moleculas is not provided through this interface (since it is perhaps easier to inbuilt them in C++). The list of currently available atoms is given in file `interfdata.h`:

```
const int q_atom_name = 52;
const char* atom_name[q_atom_name]=
{"H", "He", "Li", "Be", "B", "C", "N", "O", "F",
 "Ne", "Na", "Mg", "Al", "Si", "P", "S", "Cl", "Ar",
 "K", "Ca", "Sc", "Ti", "V", "Cr", "Mn", "Fe", "Co", "Ni", "Cu", "Zn", "Ga",
 "Ge", "As", "Se", "Br",
"Kr",
"Rb", "Sr", "Y", "Zr", "Nb", "Mo", "Ru", "Rh", "Pd", "Ag", "I",
"Xe",
"W",  "Bi", "U", "Pu"};

const int q_molecules_name = 18;
const char* molecules_name[q_molecules_name]=
{"He", "Ne", "Ar", "Kr", "Xe", "H2", "N2",
 "O2", "NH3", "N2O", "CO2", "CF4",
 "CH4","C2H2", "C2H4", "C2H6", "C3H8",
 "iC4H10"};
```

Their notation in Fortran is provided through the parameters, which are defined in the file `interfdata.inc`. The brief citation from this tedious file:

```
integer nat_H
integer nat_He
```

```
integer nat_Li
integer nat_Be
integer nat_B
integer nat_C
        ...
parameter (nat_H  = 1  )
parameter (nat_He = 2  )
parameter (nat_Li = 3  )
parameter (nat_Be = 4  )
parameter (nat_B  = 5  )
parameter (nat_C  = 6  )
        ...
```

With these tools we can initialize new matters and gases by functions defined in `interffun.h`:

```
extern "C" void inimatter__(fint* nmat,
    fint* pnatom, ffloat* pwatom, fint* pqatom,
    ffloat* pres, ffloat* temp);
extern "C" void primatter_(fint* p); // p < 0 : print all matters
// p == 0 : print nothing
// p > 0 : print matter with this number

// gas from wcpplib, base parameters
extern "C" void inigas_(fint* nmat, fint* pqmole,
 fint* pnmole, ffloat* pwmole,
 ffloat* pres, ffloat* temp);
```

The argument `nmat` is number of matter. `pnatom` is array of atom numbers, `pwatom` is array of fractions, `pqatom` is the number of elements in these arrays. Then goes pressure and temperature.

For gas `pqmole` is the number of molecules in mixture, `pnmole` and `pwmole` are the arrays of the numbers and of fractions of the given moleculas.

There is a range of pre-defined matters:

```
const int q_matter_name = 20;
const char* matter_name[q_matter_name]=
{"He", "Ne", "Ar", "Kr", "Xe", "H2", "N2",
 "O2", "NH3", "N2O", "CO2", "CF4",
 "CH4","C2H2", "C2H4", "C2H6", "C3H8",
 "iC4H10", "Air", "GasHydrogen2"};
```

So the new matters should have number more than 20. The pre-defined matters are registered in `InterfaceExtensibleContainer`:

```
InterfaceExtensibleContainer<MatterDef> IMattersCont(q_matter_name,
       matter_name);
```

These matters are from `wcpplib`. The photoabsorption cross sections for HEED are defined by similar way. We will not reproduce here all mentioned files, since they are too large and tedious.

Similarly the geometrical primitives are allocated. For example, a citation from `interfdata.h`:

```
// ordinary unit and zero vectors to which one can refere by protected pointer
extern vecReg deregx;  // unit along x
extern vecReg deregy;  // unit along y
extern vecReg deregz;  // unit along z
extern vecReg dvreg0;  // zero
const int q_vecReg = 4;
const vecReg* list_vecReg[q_vecReg] =
{
  &deregx,
```

```
  &deregy,
  &deregz,
  &dvreg0
};
// Extensible container, you can add other vectors from Fortran.
extern InterfaceExtensibleContainer<vecReg> IvecRegCont;
```

and corresponding piece of `interfdata.h`:

```
vecReg deregx(1,0,0);
vecReg deregy(0,1,0);
vecReg deregz(0,0,1);
vecReg dvreg0(0,0,0);

InterfaceExtensibleContainer<vecReg> IvecRegCont(q_vecReg, list_vecReg);
```

22.2 Allocation and identification of volumes

The mentioned class `InterfaceExtensibleContainer` handles any data except volumes. The geometrical library, on which HEED is based, allowes the volumes to be the directly allocated one in another. If the external volume is copied or deleted, all the embraced volumes are also assumed to be copied or deleted respectively. This is logically consistent, since this supports the integrity of objects and integrity of the program as well. But `InterfaceExtensibleContainer` does not assume any operations on the other objects at operation with one of them. All the objects are registered in it and if one of them silently disappears, it will be a problem. So it is not suitable for handling volumes.

On the other hand, the standard approach of geometric library does not allow us to create the volumes interactively. Indeed, it is based in class declarations. But in C++ there is no way to change the class definitions interactively. Therefore we need to define a special class which can have varied form and embrace any volumes of possibly different forms.

Such class was designed and called `absvol_with_internals`, file `wcpplib/geometry/volume_int.h`. This class defines an abstract volume with having a number of embraced volumes of the same base type, `absvol_with_internals`. The latter can also have embraced volumes. Each concrete volume in an interactive session should be a class derived from `absvol_with_internals` and also from a class defining concrete shape and from some other ingredients.

There are classes `box_with_internals`, `cylinder_with_internals`, and `tube_with_internals` in file `heed++/code/volume_spec.h`. The initial or world wolume is expected to be kept in smart pointer with container behaviour with identifier `mother_volume`. Each `absvol_with_internals` has an element `long id`, which is treated as index. It is assumed that the value of this variable should be unique for each volume in the volume tree. There is function to verify this. There is function which finds the volume by the index and returns its address. This allows to provide the identification of volumes in Fortran by indexes. Thus, the volumes are allocated differently from all the other objects, but identified similarly, by indexes.

This system is not trivial, but it works.

22.3 The indexes

The front-end C++ functions callable from Fortran accept Fortran-style indexes starting from 1, and convert them instantly to C++ indexes starting from 0. That is from each input Fortran index they subtract unity. In the output listings produced by printing front-end C++ functions the indexes appear as in C++, i.e. starting from 0. This inconsistency may be somewhat confusing, but it seems to be inevitable. Although for the volumes their indexes are not indexes but rather identifiers, for example, they can be negative, they are also reduced by one at passing to C++ for similarlity with the other indexes.

22.4 The base types

See sect. 7.8 for definition of `fint`, `ffloat`, and `fdouble`.

Most of the floats passed through the interface are defined as `ffloat`, except the geometrical data. The latter are passed as `fdouble`. In particular that means the user should not write

```
call inivec(nvec, 0.0, 0.0, 0.0, ierr)
```

since these nulls will be interpreted in fortran as `real*4`, that is floats, whereas the C++ will interprete them as doubles, and at the result the user will receive nonsense. Instead the user has to write something like this:

```
real*8 xp, yp, zp
...
xp = 0.0
yp = 0.0
zp = 0.0
call inivec(nvec, xp, yp, zp, ierr)
```

22.5 Histograms

HEED-C++ with Fortran interface does not use histograms. But the program is still linked with one of histogram packages. At the moment I am not sure whether it is inavidable or not.

22.6 Run-time error handling

The system, described in sect. 7.6 and 9.6 is good for batch mode. In the interactive mode the diagnostic message would also be desiable, but the program termination would be rather an unpleasant surprise for the user who might have already typed manually a lot of preliminary initializations. Therefore, for support of the interactive mode the abort command is replaced by the call of exception. This is done in one place, in a function responsible for final actions during error handling, `void spexit_action(ostream& file)`, see sect. 9.6 and file `wcpplib/util/FunNameStack.c`. Note that abort and exception throwing can be switched dynamically, during the program execution. For this end the main program determined in the file `interffun.c` is designed by the following way:

```
int main(void)
{
  mfunnamep("int main(void)");
  s_throw_exception_in_spexit = 1;  // to return the control
  // to fortran program  instead of abort

  fortmain_();
}
```

Every function callable from Fortran catches the exceptions just at its end, sets appropriate value of special signature of error, that is an ordinary variable which is always declared as the last parameter of this function and returns the control to Fortran. At the beginning of each function callable from Fortran this signature is checked and if it indicates the error (!=0), the control is immidiately returned to Fortran without execution of the rest of this function. This is designed so as not to multiply the errors after the first one occurs. Thus, after each call of HEED function from Fortran the user (that is the person writting the interface functions from the Fortran side) can check the value of this variable and to act accordingly. The example of Fortran-callable function must be clear after all these explanations:

```
void inientranfcs_(fint* netcs,
  ffloat* particle_mass,
  ffloat* gamma_1,
  fint* s_primary_electron,
  fint* nhmd, // number of HeedMatter
  fint* ierr)
```

```
{
  mfunnamep("void inientranfcs(...)");
  if(*ierr != 0 ) return;
  try
  {
    HeedMatterDef* ahmd = IHeedMatterDefCont.get_element(*nhmd - 1);
    IEnTranfCSCont.add_element
      (*netcs - 1,
       new EnTranfCS(double(*particle_mass),  // accepts mass in MeV
     double(*gamma_1),
     *s_primary_electron, ahmd));
  }
  catch (ExcFromSpexit& efs)
  {
    mcout<<"inientranfcs: exception is detected, the call is not executed\n";
    *ierr = 1;
  }

}
```

And how this function called from fortran is shown below (file `for_ts.f`):

```
call inientranfcs(1, 1000.0, 4.0, 0, 1, ierr);
```

22.7 The interface files

- `interfdata.inc` – Fortran included file. Interface details for Fortran. In particular, there are named constants which should strictly correspond to content of `interfdata.h`.
- `interfdata.h` — C++ included file. Interface details used in C++.
- `interffun.h` – Declarations of pseudo-Fortran C++ front-end functions, used in C++.
- `interffun.c` – Definitions of these functions, very large file. In particular it includes C++ main program.
- `volume_spec.h` and `volume_spec.c` – classes `box_with_internals`, `tube_with_internals`, and `cylinder_with_internals`
- `build.b` – compilation script for it.
- `for_ts.f` – test program written in Fortran.
- `for_ts.b` – script to compile and link the test program.

23 Conclusion

Good luck!

References

[1] I.B. Smirnov, Modeling of ionization produced by fast charged particles in gases. Nucl. Instr. and Meth. A 554 (2005) 474–493.

[2] CERN Program Library, http://www.cern.ch/cernlib.

[3] OpenScientist web site, http://openscientist.lal.in2p3.fr.

[4] ROOT web site, http://root.cern.ch.

[5] http://en.wikipedia.org/wiki/ROOT

[6] I.B. Smirnov, LikelihoodLib — Fitting, Function Maximization, and Numerical Analysis. Preprint PNPI-2421, 2001.

[7] I. B. Smirnov, Raw Pointers in Application Classes of C++ Considered Harmful. ACM SIGPLAN Notices, Vol 42, No. 4, April 2007.

[8] I.B. Smirnov, SpaceMetricLib, Geometrical Class Library for Detector Modeling in HEP. Preprint PNPI num. 2373, 2000.

[9] G. Williams, Electron binding energies, X-Ray WWW Server of the Uppsala University, Sweden, http://xray.uu.se/

[10] Pathikrit Bandyopadhyay, The program for generation of fluorescence yield, X-Ray WWW Server of the Uppsala University, Sweden, http://xray.uu.se/ .

[11] B.L. Henke, E.M. Gullikson, and J.C. Davis, At. Data Nucl. Data Tables 54 (1993) No. 2. X-Ray WWW Server of the Uppsala University, Sweden, http://xray.uu.se/.

[12] E.L. Kosarev, E.P. Podoliak, Opt. & Spectrosk, 56 (1984) 643 (in Russian).

[13] G.V. Marr and J.B. West, At. Data Nucl. Data Tables, 18 (1976) 497.

[14] D. A. Verner, D. G. Yakovlev, I. M. Band and M. B. Trzhaskovskaya, At. Data Nucl. Data Tables, 55 (1993) 233;

[15] M.B. Trzhaskovskaya, private communication, calculations by program package RAINE [14], 1995.

[16] G.V.Marr and J.B.West, Atomic Data and Nuclear Data Tables, 18 (1976) 497.

[17] *M.E.Riley, C.J.MacCallum, F.Biggs* . Theoretical Electron-Atom Elastic Scattering Cross Sections At. Data and Nucl. Data Tables 15 (1975) 443, 16 (1975) 379.