Beam Simulation Tools for GEANT4 (BT-v1.50)
User's Guide

V. Daniel Elvira[1], P. Lebrun[1], P. Spentzouris[1]

[1] *Computational Physics Department, Computing Division, Fermi National Accelerator Laboratory*

1 March 2003

Contents

1 Introduction

Geant4 is a tool kit developed by a collaboration of physicists and computer professionals in the high energy physics field for simulation of the passage of particles through matter. The motivation for the development of the Beam Tools is to extend the Geant4 applications to accelerator physics. The Beam Tools are a set of C++ classes designed to facilitate the simulation of accelerator elements: r.f. cavities, magnets, absorbers, etc. These elements are constructed from Geant4 solid volumes like boxes, tubes, trapezoids, or spheers. In `BT-v1.50`, `1` refers to the tools version and will change every time there is a major upgrade, while `50` is the associated Geant4 version. In this case, we are using Geant4.5.0 with `patch01`.

There are many computer programs for beam physics simulations, but Geant4 is ideal to model a beam through a material or to integrate a beam line with a complex detector. There are many such examples in the current international High Energy Physics programs. For instance, an essencial part of the R&D associated with the Neutrino Source/Muon Collider accelerator is the ionization cooling channel, which is a section of the system aimed to reduce the size of the muon beam in phase space. The inonization cooling tecnique uses a combination of linacs and light absorbers to reduce the transverse momentum and size of the beam, while keeping the longitudinal momentum constant. The MuCool/MICE (muon cooling) experiments need acurate simulations of the beam transport through the cooling channel in addition to a detailed simulation of the detectors designed to measure the size of the beam. The accuracy of the models for physics processes associated with muon ionization and multiple scattering is critical in this type of applications. Another example is the simulation of the interaction region in future accelerators. The high luminosity and background environments expected in the Next Linear Collider (NLC) and the Very Large Hadron Collider (VLHC) pose great demand on the detectors, which may be optimized by means of a simulation of the detector-accelerator interface.

2 How to Use this Guide

We assume that the reader knows the basics of Unix, C++, and Geant4. Except in the case of complex applications which may need some code development, there is no need to be an expert in object oriented design or C++ to use Geant4 and the beam tools to put together a simulation. After summarizing the `BT-v1.50` features, we explain how to install the tools in a Linux system. The installation files include the `MuCool` package, a Geant4 based simulation of an accelerator section which makes use of the Beam Tools. The `MuCool` example is a simple version (first section) of one of the cooling channels proposed for the neutrino factory : the Double Flip (DF) channel, described in Ref. [1].

After installation, the user can inmediatelly run a few events, following the directions in Section 5, and analyze the generated data (see Section 6). This exercise will give you a glimpse at the capabilities of the tools and the `MuCool` analysis package.

If you are writting your own accelerator simulation, Section 7 will tell you how to incorporate the basic accelerator elements using `MuCool` as an example. Other examples are also included to illustrate additional capabilities. The header files with `MuCool` prefix correspond to the Geant4 user classes, or others which are typically modified by the user. They are described in Section 8. The header files with `BT` prefix define the `BT-v1.50` classes. They are not supposed to be modified, except to enhance the tools capabilities. They are available online in the beam tools reference guide [2], and briefly described in Section 9. In general, we do not describe the Geant4 classes (with prefix `G4`). except the user classes. Documentation on installation and usage of Geant4 is available in Ref. [3].

3 Summary of Features

The Beam Tools version 1.50 (`BT-v1.50`) allows the user to construct solenoid magnet objects, including the coil material, shielding, and associated magnetic field, which is computed analytically from a current density in volume. Pill Box r.f. cavities are also available, including the material, the field and the possibility of adding windows to increase the shunt impedance. Arbitrary fields, created with special tools from calculations or measurements, can also be constructed as magnetic or electric field map objects. Absorber objects of cylindrical and parabolic shape, including the material and its container can also be added to the simulation. The initial beam may be read from an `ASCII` file or generated following a Gaussian distribution for the average position, momentum and length of the beam. The relative phase of each r.f. cavity with respect to the others is tuned using a "reference" particle, which is processed automatically before the beam. (See Sections 7.2 and 7.4.6 for details). The reference particle is defined as a particle with velocity equal to the phase velocity of the r.f. wave. Typically, this particle takes kinematic parameter values that match the associated mean values for the beam.

Although the user is free to use any analysis package, the MuCool example utilizes `Root`, the high energy physics analysis tool developed at CERN [4]. The simulation creates a set of histograms and NTuples for diagnostic and analysis of the simulated data. A snapshop of the beam (mostly kinematic information) is stored in an NTuple one time per unit cell of the accelerator lattice. A skeleton of a `root` analysis package to process the NTuples is also provided.

The user may also choose from different available visualization packages, which will allow him/her to see a representation of the simulated apparatus on the screen. The MuCool example uses Open Inventor [5], which allows direct manipulation of the objects on the screen, plus perspective rendering via the use of light.

4 Installing the Beam Tools (and the MuCool Example)

Before installing `BT-v1.50`, you have to make Geant4 available in your computer. If you are working on the Computational Physics Department (CPD-CD-Fermilab) Linux computers, the Geant4 libraries become available by typing:

```
% setup geant4 v4_5_0 -f Linux+2.4 -q GCC_2_95_2
```

If you have access to a machine where the FNAL products database is available (ups/upd), ask your system administrator to add `geant4 v4_5_0` to your local database. Then, you should type the `setup` command as in the CPD machines. In both cases, the environmental variable `GEANT4_DIR` will be defined pointing to the directory which contains the Geant4 libraries. If you have not access to a CPD machine or the FNAL products database, you will need to install Geant4 in your system. Visit the Geant4 web page [3] and follow the link to the Installation Guide. You will find detailed information about the different computer platforms, operative systems and software supported or required by Geant4. The Geant4 source code and libraries are also available. Make sure you download the Linux2.4 version for the g++ gcc 2.95.2 compiler, which is the only compatible with `BT-v1.50`.

Once Geant4 is available in your system, a list of the `$GEANT4_DIR` directory must display the following:

```
% ls $GEANT4_DIR
CVS         ReleaseNotes    environments   include   source
Configure   config          examples       lib       ups
```

Then, you must create a work directory where you will edit the main program, write your simulation using the Geant4 user classes, and run the executable. For example, if your user name is `johndoe`, you may create the `~johndoe/work` directory, and download there the MuCool example and `BT-v1.50` from the Fermilab Geant4 web page [2]. You must download, un-zip, and un-tar the file `BT-v1.50.tar.gz`. The newly created `MuCool` directory must show the following contents:

```
% ls ~johndoe/work/MuCool
GNUmakefile   MuCool.cc   bin   include   set_env_g450.sh   src
```

The `MuCool.cc` file contains the `main` program for the MuCool example, while the `src` directory contains the source code for MuCool and the Beam Tools, and `include` the header files. Remember that files related to the Beam Tools start with the prefix `BT` and files associated with the example start with the prefix `MuCool`. Next, you should download `AuxLib.tar.gz` from the Fermilab Geant4 web page. It contains some libraries [6] needed mostly in the solenoidal field calculation: `Exceptions`, `SpecialFunctions`, `ZMtools`, and `ZMutility`. Un-zip and un-tar `AuxLib.tar.gz` in the area of your choice in your system. If you did not install Geant4 from the CPD or FNAL products databases, but from the Geant4 web page, you need to download `architecture.gmk` and `Linux-g++.gmk` from the Fermilab Geant4 web page to replace the file with the same name in `$GEANT4_DIR/config` and `$GEANT4_DIR/config/sys`. Make sure that `$GEANT4_DIR` points to the area in your system you selected for the Geant4 installation.

The `set_env_g450.sh` script must be run (`% source set_env_g450.sh`) before compiling, linking, executing the simulation package, or before building the Geant4 libraries. The user must previously edit the file and make sure some environmental variables point to the correct areas:

```
setenv SPECIAL_BASE_DIR /area-where-you-installed-the-AuxLibs/zoom2/
zoom/zoomdist/releases/base

setenv G4SYSTEM Linux-g++
setenv G4INSTALL $GEANT4_DIR
setenv G4LIB $G4INSTALL/lib
setenv G4WORKDIR ~johndoe/work/MuCool
setenv G4TMP $G4WORKDIR/tmp
setenv G4BIN $G4WORKDIR/bin
```

These variables point to the location of the Geant4 library (`G4LIB`), the user work directory (`G4WORKDIR`), the object files (`G4TMP`), and the executable (`G4BIN`). The `set_env_g450.sh` file is an example which runs on the CPD Linux machines. If you do not have access to the FNAL products database, you will have to install the packages which are `setup` in `set_env_g450.sh`. You do not need, however, to use `Root` or the visualization capability (packages) if you do not wish. `Root` is available on the web [4].

Now you are ready to compile/link the package and create a `MuCool` executable which will be located at `$G4WORKDIR/MuCool/bin/Linux-g++`.
In `$G4WORKDIR/MuCool`, just type:

```
% source set_env_g450.sh deb verb vis
% gmake
```

The parameters `deb` (for debug version), `verb` (for verbosity on), `vis` (for building a visualization version) are optional and can be written in any order or omitted. The time performance of Geant4 is seriously degraded when running executables with debug or verbosity on.

5 Running the MuCool Example

In this section, we will give only "operative" directions on how to run the MuCool example, for the user to test the installation and get a feeling on the package capabilities. You will be able to visualize the simulated apparatus, look at some diagnostic plots, and analyze the output data files. In `$G4WORKDIR/MuCool/bin/Linux-g++`, you will find the following files: `MuCool.in` (geometry, field, beam and global user defined input parameters), `InputBeam.dat` (kinematic variables for each one of the list of particles forming the input beam), `MuCoolSol.dat` (binary file with solenoidal field information), `prerun_vis.mac` (macro file for visualization setup), and `MuCool.mac` (macro file in case you are running in macro mode). The `MuCool.in` file is thoroughly commented. Details on the input parameters are discussed in succeeding sections.

There are three different run modes for `MuCool`, accessed by typing one of the following three options:

```
%MuCool MuCool.in VISUAL
%MuCool MuCool.in MACRO MuCool.mac
%MuCool MuCool.in HARD
```

5.1 The `VISUAL` *Mode*

The `VISUAL` mode opens an x-window with an image of the simulated apparatus. Figure 1 is an Open Inventor [5] view of a unit cooling cell of the MuCool example lattice. The viualization window and image was created by the macro `prerun_vis.mac` which was run from `main`, before any particle was propagated through the geometry. For information about macro commands for visualization, see the Geant4 User's Guide [3] under "How to Visualize the

Detector and Events". The image in Fig. 1 can be manipulated interactivelly with the mouse. Rotations are achieved by left-clicking on the left vertical and horizontal wheels and draging towards the desired direction. The vertical wheel on the right is for zooming. A right-click on any part of the image will bring the "examiners viewer" and then the "functions" menu: help, home (default image), set home (set new default image), view all (largest possible image which fits the screen), seek (sets new rotation center). These functions may also be accessed directly by left-clicking on the icons (question mark, house, house with arrow, etc). The icon with the hand allows to manually rotate the drawing by just clicking and dragging on the image. The "examiners viewer" then "draw style" menu contains options like "as is" (for solid images) and "wireframe" (for transparent images).

If you are ready to run some particles through the simulated apparatus, you should first left-click on the "File" menu and select "close". Then your terminal window will display the prompt `Idle>`, inviting you to interactivelly modify settings and run particles through the simulation using the built-in (or eventually user defined) commands provided by Geant4. To quit the program type `exit` on the `Idle>` prompt. The macro file `MuCool.mac` contains some examples of interactive Geant4 commands. For more information, see the chapter on "Communication and Control" in the Geant4 User's Guide [3].

5.2 The `MACRO` *Mode*

In `MACRO` mode, some settings, particle or beam parameters, and commands are read from a macro file. For example, `MuCool.mac` contains the following lines:

```
/gun/energy 100 MeV
/gun/direction 0 0 1
/gun/position 2 1 0 cm
/run/beamOn 1
```

The first three commands set the "gun", which "shoots" the particles. The kinetic energy is set to 100 MeV, the particle direction along the z axis and the initial position is $(2, 1, 0)$ cm from the global origin. The last command turns the beam on and orders to shoot one particle. For more on built-in commands or how to create user defined commands, see the chapter on "Communication and Control" in the Geant4 User's Guide [3].

5.3 The HARD(-wired) Mode

This mode should be used to run a large beam through a thoroughly debugged and stable simulation. Some user interfase commands are hardwired (embedded in the code), and the beam is either produced in the particle generation user's class or read from an ASCII file.

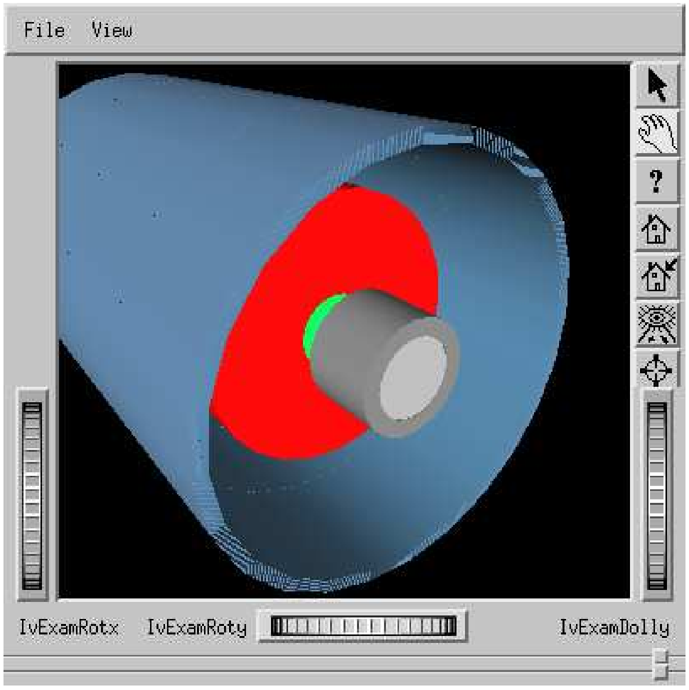


Fig. 1. Open Inventor window showing a unit cell of the MuCool example lattice. The blue cylinder is the solenoid, the red cylinder is the linac, and the grey cylinder is the liquid hydrogen absorber.

6 Analyzing the Simulated Data

There are three steps in a High Energy Physics simulation study: the writing of the simulation code, the production of "Monte Carlo" events, and the analysis of these events.

The MuCool example produces a set of Monte Carlo events, which can be analyzed with the `Root` software package. It creates a `Root` file (`MuCool.root`) which contains a set of histograms and NTuples with kinematic information of the individual beam particles. Directions on how to use `Root` are available in the `Root` user's guide [4]. The `MuCool.root` file contains one NTuple per lattice unit cell, which is a "snapshot" of the beam at that particular z location. Figure 2 is the `Root` browser window, which displays a set of NTuples and histograms: `RTTracRefPart1` is an NTuple containing complete trace information for the reference particle used to tune the cavity phases, `RTTraces0` contains trace information for the first particle in the list, `rf_NTuBegin` is a snapshot of the initial beam, and `rt_NTu0-19` are snapshots of the beam at every one of the 20 unit cells of the MuCool lattice. The histograms `TestBx,y,z` show the global magnetic field on axis. Figure 3 is the NTuple viewer for `rf_NTuBegin`. It displays the 11 variables: x, y, z, $x\prime$, $y\prime$, P (total momentum), Tof (time of flight), PiD (particle ID), $EventNum$, $ParentID$ (ID of the parent of the particle), $weight$ (particle weight in the distribution). As an illustration of the analysis of Monte Carlo events generated by the MuCool package, Fig. 4 shows the evolution of p_y versus x for a beam that propagates along the Double Flip (DF) cooling channel. The first histogram corresponds to the initial beam, and the next four are snapshots of the beam taken every 5 cooling cells.

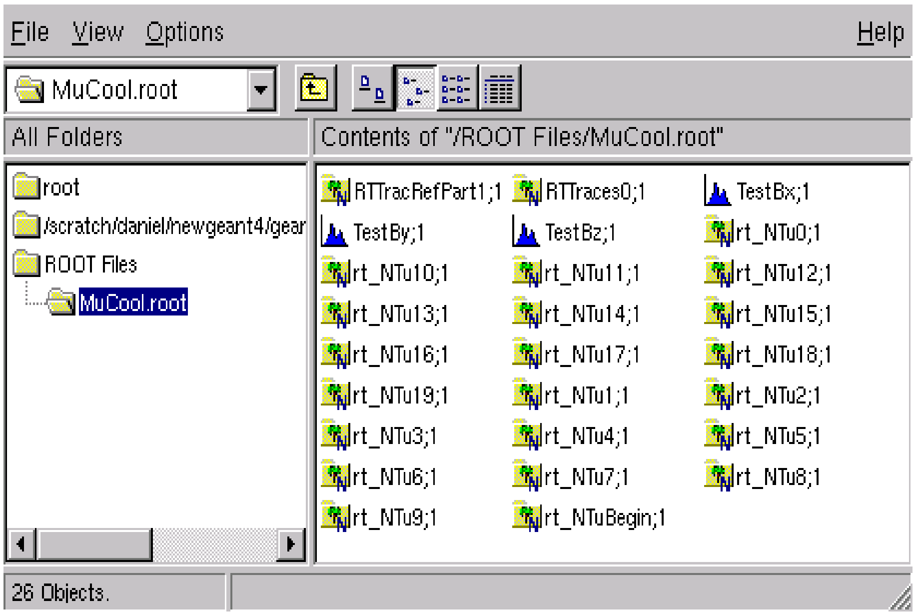


Fig. 2. Browser window, showing a set of NTuples and histograms.

`Root` may be turned off by following directions under "Setup Root" in the `set_up.sh` file. You can search your `cc` (source) and `hh` (header) files in the `MuCool`, `src`, and `include` areas under `work` for the key word: `ROOTFLAG`. This flag controls a set of `#ifdef` statements which you may inactivate by unsetting the flag. You may also remove/replace these blocks in case you wish to use a different analysis tool, or change the contents of histograms and NTuples.

In the `$G4WORKDIR/MuCool/bin/Linux-g++` directory you will find three files: `MuCool.C`, `Analysis.h`, and `Analysis.C`. `MuCool.C` is the main `Root` macro:

```
gROOT->LoadMacro("Analysis.C");
Analysis *m = new Analysis();
m->Loop();
```

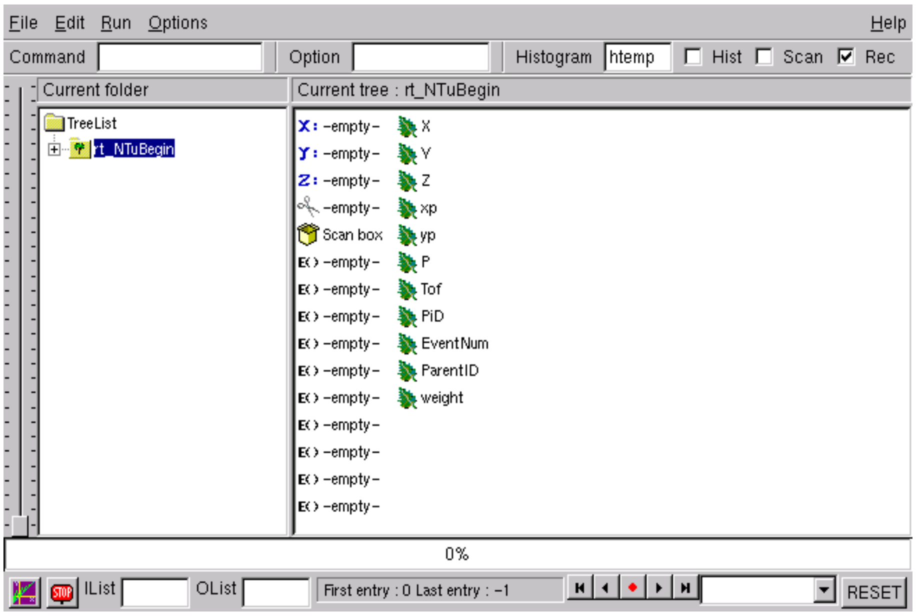


Fig. 3. NTuple viewer for `rf_NTuBegin` showing the 11 variables: x, y, z, $x\prime$, $y\prime$, P (total momentum), Tof (time of flight), PiD (particle ID), $EventNum$, $ParentID$ (ID of parent's particle), $weight$ (particle weight in the distribution).

`Analysis.C` includes a C++ method, `Loop()`, which performs the event by event analysis (see Fig. 5). `Analysis.h` defines a C++ class with the NTuple variables as data members (not seen in Fig. 6) and a set of methods. In the `Analysis(TTree *tree)` constructor, you should make sure to use the name of the `Root` file and NTuple you wish to analyze. In the case illustrated in Fig. 6, we are studying `rt_NTuBegin` from `MuCool.root`.

If you need to change the structure (number and/or name of variables) of the NTuples, you will have to remake the `Analysis` class using the `MakeClass` method of the `TFile` class (see `Root` documentation [4]).

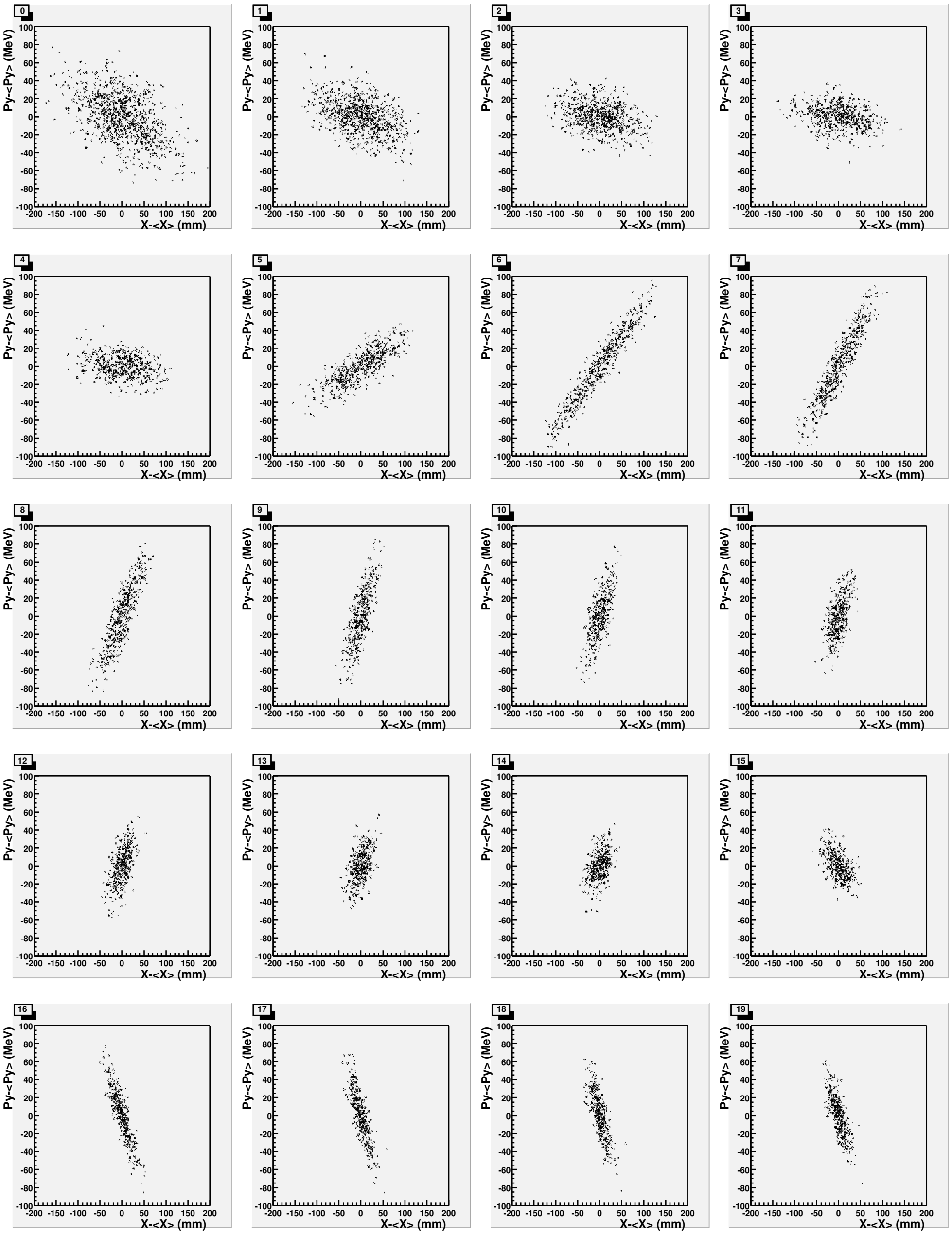


Fig. 4. Evolution of p_y versus x for a realistic beam along the Double Flip (DF) cooling channel (`MuCool` example). The first histogram corresponds to the initial beam, and the next five are snapshots every 5 cells of the first section of the DF.

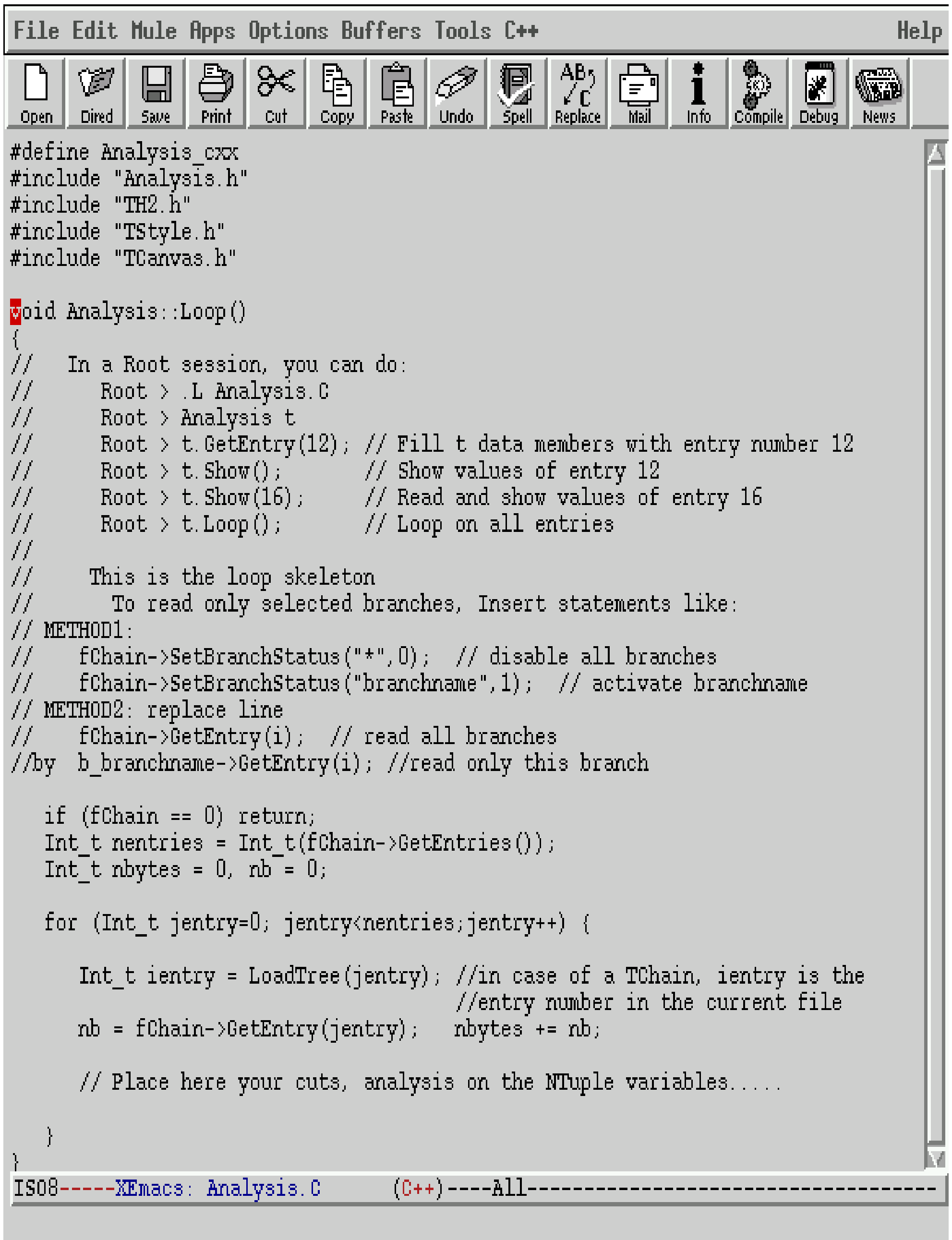
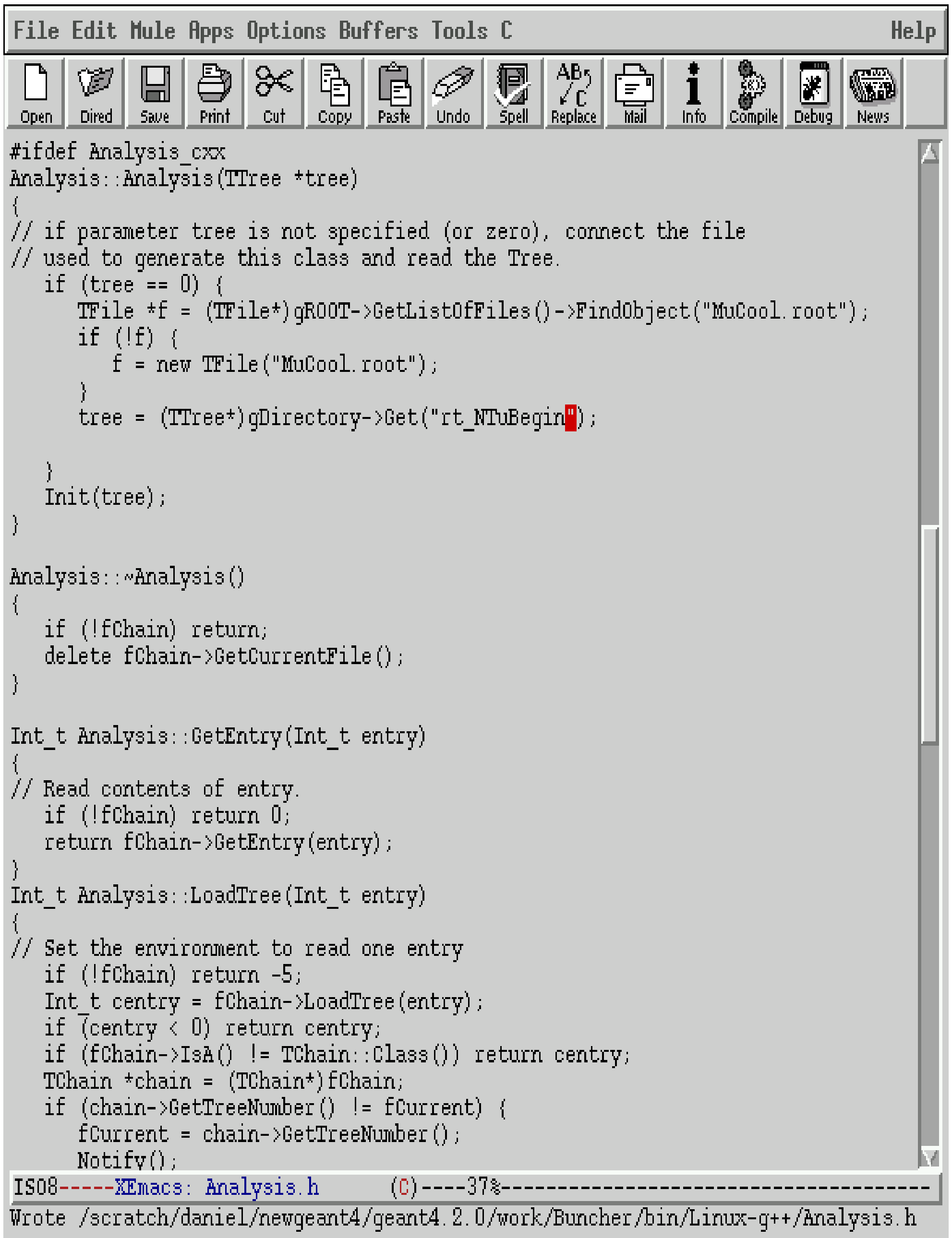


Fig. 5. `Analysis.C` includes a C++ method, `Loop()`, which performs the event by event analysis.




```
#ifdef Analysis_cxx
Analysis::Analysis(TTree *tree)
{
// if parameter tree is not specified (or zero), connect the file
// used to generate this class and read the Tree.
   if (tree == 0) {
      TFile *f = (TFile*)gROOT->GetListOfFiles()->FindObject("MuCool.root");
      if (!f) {
         f = new TFile("MuCool.root");
      }
      tree = (TTree*)gDirectory->Get("rt_NTuBegin");

   }
   Init(tree);
}

Analysis::~Analysis()
{
   if (!fChain) return;
   delete fChain->GetCurrentFile();
}

Int_t Analysis::GetEntry(Int_t entry)
{
// Read contents of entry.
   if (!fChain) return 0;
   return fChain->GetEntry(entry);
}
Int_t Analysis::LoadTree(Int_t entry)
{
// Set the environment to read one entry
   if (!fChain) return -5;
   Int_t centry = fChain->LoadTree(entry);
   if (centry < 0) return centry;
   if (fChain->IsA() != TChain::Class()) return centry;
   TChain *chain = (TChain*)fChain;
   if (chain->GetTreeNumber() != fCurrent) {
      fCurrent = chain->GetTreeNumber();
      Notify();
```


Fig. 6. **Analysis.h** defines a C++ class with the NTuple variables as data members (not seen here) and a set of methods.

7 Usage of the Beam Tools by Examples

This Section is devoted to explain how to construct beams and accelerator elements using the Beam Tools. We will only discuss the usage of the Beam Tools classes. See Section 9 for a list and a description of the `BT-v1.50` classes.

Geant4 provides a set of user classes, or "hooks", for the user to provide the geometry, field, and beam, or access information at different stages of the simulation process. You can find the header and the implementation files in the `include` and `src` subdirectories below `$G4WORKDIR/MuCool`. The three fundamental user classes for accelerator simulations are:
`MuCoolPrimaryGeneratorAction` (beam construction), `MuCoolConstruction` (geometry, e.m. field construction), `MuCoolSteppingAction` (diagnostic and actions at the end of every step in the simulation). The first rule in programing with `BT-v1.50` is to include the corresponding header file, each time a Beam Tools class is utilized.

7.1 Data Cards for Input Parameters

`BT-v1.50` uses a native input parameter handler instead of the Geant4 messenger classes. The `MuCooldataCards` class allows the user to create a `MuCool.in` file containing parameters which may be passed to the simulation at run time. No compilation or linking is needed upon a modification of the parameter values in `MuCool.in`. To add a parameter, you should first edit `MuCooldataCards.cc` and add a new line following the appropriate syntax, as in the example below:

```
MuCooldataCards::MuCooldataCards()
{
cd["numEvts"] = 1.;
cd["NumberOfTraces"] = 0.;
cs["ROOTFileName"] = "MuCool.root";
cs["InputBeamFile"] = "InputBeam.dat";
.
.
.
}
```

`ROOTFileName` is the name of the output file containing histograms and NTuples, and `InputBeamFile` the name of the input beam file. `numEvts` is the total number of particles to be processed, and `NumberOfTraces` the number of particles for which the user wants to create a trace NTuple. Note that the only types allowed to the parameters are `string` and `double`, although they may be casted to any other type in the user's code. The statement `cs[‘‘....’’]` is used to add a parameter of type `string`, and `cd[‘‘....’’]` to add a `double`. It is necessary to compile and link `MuCool` (using `gmake`) after making a modification to `MuCooldataCards.cc`. The file `MuCool.in` in the `Linux-g++` area is meant for the user to modify the parameter values without the need of re-compilation/linking. For example, the following lines in `MuCool.in` would overide the hard-wired values in `MuCooldataCards.cc`:

```
numEvts 100.
NumberOfTraces 5.
ROOTFileName MuCool2.root
InputBeamFile InputBeam2.dat
```

Other parameters to control global run conditions are: `ChannelType` (only `MuCool` available), `rfCellType` (either `PillBox`, `rfmap`, or `none` for no r.f. system), `BeamMode` (`gaussian` beam or read from `file`), `verboseTrackingLevel` (`0` for no tracking information on screen, higher numbers to get more and more information), `NoStochastics` is `1` if no multiple scattering or straggling is modelled in the physics processes, `0` to turn on these processes). `MaxRadiusInChannel` is the maximum radius of any element in the simulation, `StopAtRadius` and `StopAtZ` are the thresholds in r-z space beyond which a particle is no longer propagated. `KineticEnergyCut` is the minumum allowed kinetic energy of a particle before it is killed.

The syntax to read a `string` or `double` parameter from the user code is:

```
std::string fileROOTout =
MyDataCards.fetchValueString("ROOTFileName");
int vL = (int) MyDataCards.fetchValueDouble("verboseTrackingLevel");
```

Note that the `verboseTrackingLevel double` has been casted to an `int vL`.

7.2 The MuCool `main` *Program*

The `main` function is implemented by the user in `MuCool.cc`, and controls the flow of the program. It takes two or more arguments, as explained in Section 5: the input file, the run mode, and eventually a macro file. The handling of the input arguments and the selection of the run mode is done in the first and the last few blocks of `main`, using a `switch` statement.

The core of `main` starts with the construction of the `runManager`. This object of type `G4RunManager*` actually controls the flow of the program and manages the event loop within a run. The are some mandatory user classes which must be set next:

```
// set mandatory initialization classes

MuCoolConstruct *detector = new MuCoolConstruct();
runManager->SetUserInitialization(detector);
cout << " Processing MuCool Example (Double Flip Channel) " << endl;
MuCoolPhysicsList *physList = new MuCoolPhysicsList();
runManager->SetUserInitialization(physList);

// set mandatory user action class

runManager->SetUserAction(new MuCoolPrimaryGeneratorAction);
runManager->SetUserAction(new MuCoolTrackingAction);
MuCoolSteppingAction* stepAct = new MuCoolSteppingAction;
runManager->SetUserAction(stepAct);
runManager->SetUserAction(new MuCoolEventAction);
```

These objects of the user classes contain the information related to the geometry of the apparatus, the fields, the beam, and actions taken by the user at different times during the simulation. They will be described in the following sections.

Geant4 allows to set a production cut on secondary particles, like gammas and electors in the MuCool example. Given a primary particle going through matter (for example a muon), a secondary particle will not be generated if the current energy of the primary particle is low enough so that it would come to rest in a given range (cut value by range). Cuts could also be set directly by energy. More details on production cuts are available in the Geant4 user's guide [3]. In `main`, we explicitly set the cut values for gammas and electrons by range to the default value of 2 mm. For this, we use interface commands defined by the user in the `MuCoolPhysicsListMessenger` class.

```
UI->ApplyCommand("/range/cutG 2 mm");
UI->ApplyCommand("/range/cutE 2 mm");
```

The `runManager->Initialize()` statement initializes the Geant4 kernel. If the accelerator has an r.f. system, like in the MuCool example, `main` must contain a block where parameters are set to run a reference particle. As illustrated in Fig. 7, the global e.m. field is retrieved and the reference particle mode set. Then, all stochastic processes are turned off: delta rays, multiple scattering, and straggling. Next, the reference particle is processed with a call to `runManager->BeamOn(1)`. At the end, the normal running mode is set in preparation for processing the beam (see Fig. 8).

The last important block in `main` is a `switch` statement for the three cases associated with the run modes: VISUAL, MACRO, and HARD, which were described in Section 5.

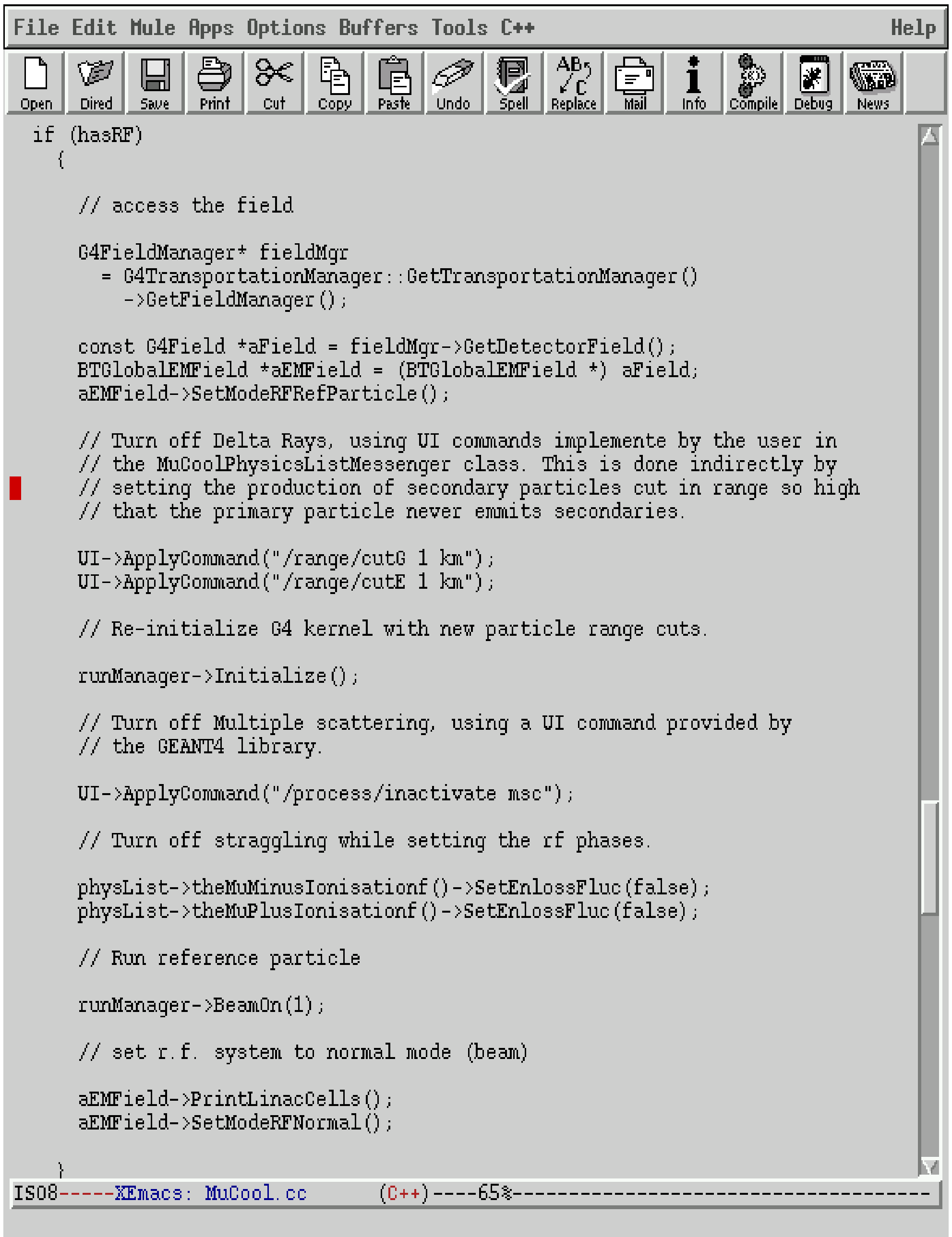


Fig. 7. Use of a reference particle for r.f. phase tunning in `main`.

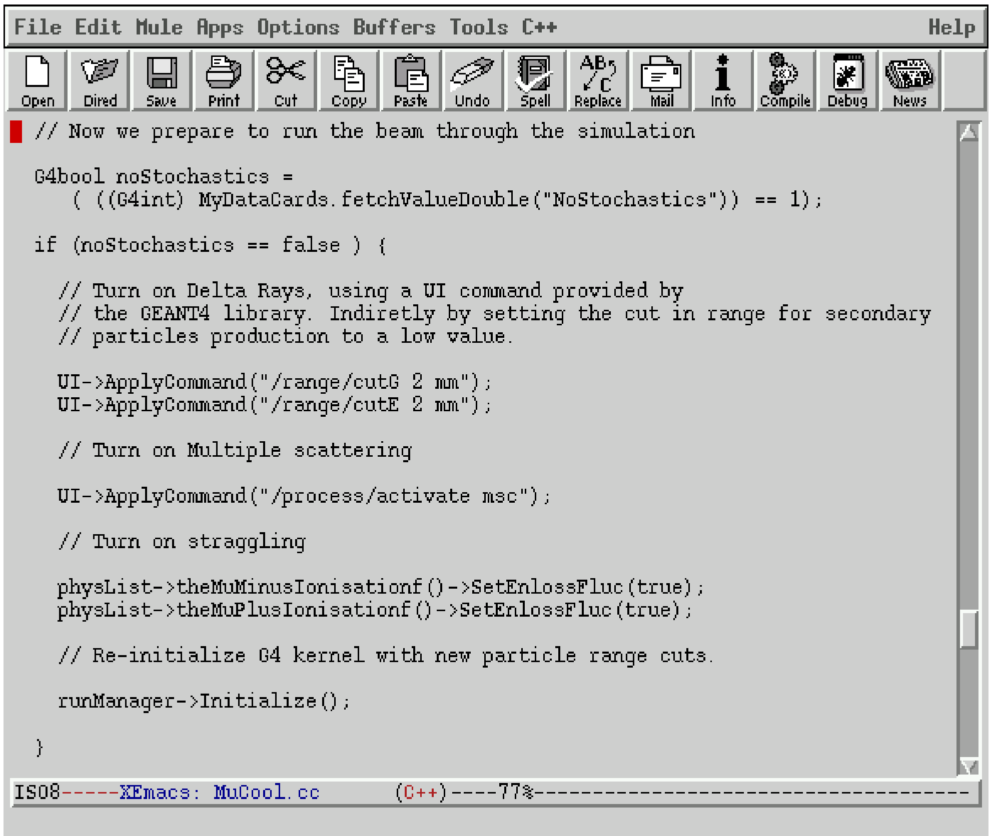


Fig. 8. Preparation for normal run in `main`.

7.3 The Beam

The user constructs the beam by writting his/her own code in methods provided by the `MuCoolPrimaryGeneratorAction` user class. In the MuCool example, there are two modes of beam operation, controlled from `MuCool.in`: either the beam is read from `InputBeamFile` (`BeamMode file` option) or it is generated from `AverageKineticEnergy`, `BetaFunc`, `SigmaX`, `BunchLength`, and `DeltaEoverE`, which are parameters of a Gaussian beam (`BeamMode gaussian` option). The beam in the example can be injected at an arbitrary instant and position with respect to the global coordinate system (controlled by `ZOffsetStart`, `TimeOffsetStart`).

```
#
# Beam information
#
MuonCharge 1.
# 1 means positive muons
muDoDecay 0
# 0 means the muon does not decay
AverageKineticEnergy 94.35
```

```
# ave kinetic energy of the initial Gaussian beam (in MeV)
ReferenceKineticEnergy 94.35
# energy of the reference particle (in MeV)
BetaFunc 380.301
# initial value of the beta_perp function (in mm)
SigmaX 54.6
# sigma of the beam X distribution (in mm)
BunchLength 0.33356
# sigma of the time distribution (in ns)
DeltaEoverE 0.2119
# E means kinetic energy here.
# 10% of total 200MeV
ZOffsetStart 0.0
# distance from z-origin for the initial beam (in mm)
TimeOffsetStart 0.
# initial time (in ns)
```

The user also controls the kinetic energy of the reference particle utilized to tune the phase of each r.f. cavity (`ReferenceKineticEnergy`), the charge of the muon (`MuonCharge`), and whether it is allowed to decay or not (`muDoDecay`).

The file `MuCoolPrimaryGeneratorAction.cc` contains the `MuCoolPrimaryGeneratorAction::MuCoolPrimaryGeneratorAction()` constructor, and the `GeneratePrimaries` method. The constructor typically contains the actions which need to be performed only once. It should not be modified unless you want to change the particle type, using `FindParticle(particleName="...")`, create new beam modes, or in general add/remove data members to the class (defined in the header file with the prefix `my`). In MuCool, the `MuCoolPrimaryGeneratorAction` constructor creates the `rt_NTuBegin` NTuple and initializes the injection coordinates of the beam. It also verifies the presence of an r.f. system to decide whether or not to process a reference particle to tune the cavity phases.

Each particle of the beam is generated after the previous one has been processed through the simulation. This is done in the `GeneratePrimaries` method, also implemented in `MuCoolPrimaryGeneratorAction.cc`. If the system has r.f., the code enters in reference particle mode, a reference particle is processed, and the times when the particle goes through the center of the r.f. cavities are internally stored for tuning the cavity phases to the desired synchronous phase. The code below illustrates the particle generation process which starts by setting its initial position, kinetic energy, and direction:

```
if (myhasRF && aEMField->isModeRFRefParticlef())
{
```

```
mymuonGun->SetParticlePosition(G4ThreeVector(0.,0., myzStart));
mymuonGun->SetParticleEnergy(
MyDataCards.fetchValueDouble("ReferenceKineticEnergy"));
mymuonGun->SetParticleMomentumDirection(G4ThreeVector(0.,0., 1.));
mymuonGun->GeneratePrimaryVertex(anEvent);
return;
}
```

If the `BeamMode` is `gaussian`, the `GeneratePrimaries` method generates one particle at a time with Gaussian distributed p_x, p_y, $time$, and $energy$. The user may add/remove correlations between the particle coordinates. For example, MuCool includes an x-p_y correlation to account for the angular momentum of the beam in a 3 T solenoidal field, as well as a transverse-longitudinal correlation to optimize the Double Flip channel performance. In case the `BeamMode` is `file`, the particle information is read out from an `ASCII` file and stored in the array `beam`. The format of the file is:

`Ptcle#` x`(cm)` p_x`(MeV)` y`(cm)` p_y`(MeV)` z`(cm) E(MeV) T(sec) Weight`

The user will most probably have to change formats when writting a different application. This is a trivial exercise achieved by just changing the assignments of the `beam` array to the kinematic variables. Be aware of the units.

```
else if (myBeamMode == "file")
{
float beam[9] = 0.,0.,0.,0.,0.,0.,0.,0.,0.;

// read file with input beam information

myinputbeam >> beam[0] >> beam[1] >> beam[2] >> beam[3]
>> beam[4] >> beam[5] >> beam[6] >> beam[7] >> beam[8];

// Fill up arrays with kinematic information

position[0]=beam[1]*10.; // from cm to mm
position[1]=beam[3]*10.; // from cm to mm
position[2]=0.0;
Enow = beam[6]; // in MeV
momentum[0]=beam[2]; // in MeV
momentum[1]=beam[4]; // in MeV
momentum[2]=sqrt(Enow*Enow-momentum[0]*momentum[0]-
momentum[1]*momentum[1]-mp*mp); // in MeV
Tnow = beam[5]/29.99792458; // in ns
weight = beam[8]; // particle weight
```

}

At the end of GeneratePrimaries, rt_NTuBegin is filled and the "gun" to "shoot" the particle is set in the same way as for the reference particle. More information on how to "generate primaries" is available in the Geant4 user's guide [3].

7.4 The Accelerator

We will describe here how to use BT-v1.50 to model the different pieces of the accelerator system: magnets, r.f. cavities, absorbers, detectors. The tools use Geant4 library classes, which start with the prefix G4. Information on the Geant4 classes is available in Ref. [3].

The simulation code is implemented in the Construct() method of MuCoolConstruct.cc. The first part of this method does the initialization of the magnetic and full electromagnetic fields: BTGlobalMagField* magFF and BTGlobalEMField *fullEMField. The equation of motion is initialized, the stepper selected, and the accuracy parameters set. In the MuCool example, we selected the G4ClassicalRK4 (Runge-Kutta) stepper, which is accurate to $O(l^4)$, with l the integration step size in real space. The user should select the stepper that better suits the needs of the application, from those available in the Geant4 library [3]. Note that Runge-Kutta integrators are not symplectic, which means that in certain applications long term stability problems might arise. The most common example of such an application is long term tracking of particles in a circular channel. The net effect of a non-symplectic integrator is the same as a slight non-conservation of phase space area. In the case of a circular ionization cooling channel this should not be a problem, since the effect of the absorbers will be much larger than the numerical inaccuracies of the integrator. Of course, a non-symplectic integrator will limit the ability to understand the dynamic properties of such a channel in the absence of the absorbers. It is worth emphasizing that the limitations of the Runge-Kutta integrators are inherent to the method, and have nothing to do with their implementation in Geant4. The same problem exists in Geant3 and its derivatives, like DPGeant [7], with its double precision implementation of the integrator.

The MuCool example, in Construct(), also illustrates on how to construct materials for use in the magnets, cavities, and absorbers. See the Geant4 user's guide for more information on these choices, as well as sintax issues associated with the classes with prefix G4 [3]. By contrast, the Beam Tool classes start with a prefix BT. Information on the BT-v1.50 classes is available in Section 9.

7.4.1 The "world" and the MuCool Lattice

The MuCool example is the first section of the Double Flip cooling channel [1], which consists of a periodic structure of 20 unit cells like the one shown in Fig. 1. First, the user must setup the "world", which will contain the simulated apparatus. The "world" volume is defined as an empty "experimental hall" box, filled with vacuum, in the MuCool example.

```
G4Box* experimentalHall_box
= new G4Box("expHall_box",expHall_x,expHall_y, lzAll);
G4LogicalVolume* experimentalHall_log
= new G4LogicalVolume(experimentalHall_box, Vacuum,"expHall_log",0,0,0);
G4ThreeVector expHallPos(0.,0.,0.);
G4VPhysicalVolume* expPhys = new G4PVPlacement(0,expHallPos,"expHall",
experimentalHall_log,0,false,0);
```

Every element in Geant4 has three componets: the solid or shape (a `G4Box` object for the experimental hall), a logical volume of type `G4LogicalVolume` which includes the material the solid is made of, and a physical volume of `G4PhysicalVolume` type which includes the position of the object in global coordinates. The vector `(expHall_x,expHall_y, lzAll)` gives the dimensions of the box (half lengths of the three sides). The geometric center of the experimental hall is located at the origin (`expHallPos=(0,0,0)`). Each logical volume must be assigned a maximum step size. This association of step and volume assumes that properties and parameter values of volumes are fairly uniform. If the field changes abruptly inside a given volume, it may be necessary to introduce a daughter volume with a different step size. Geant4 calculates the step size following accuracy criteria related to the fluctuations in fields and volume properties, but the user sets the maximum allowed step size for a logical volume by typing:

```
G4double maxStep = MyDataCards.fetchValueDouble("MaxStepSizeDefault");
experimentalHall_log->SetUserLimits(new G4UserLimits(maxStep));
```

The magnetic field of an accelerator section does not typically fall abruptly at the beginning and the end. Real systems are matched to similar preceeding and succeeding lattices to avoid sudden field changes which would affect the motion of the beam. In MuCool, we have added a pre and a post section to ensure a smooth field at the edges. These two sections consist of a magnetic lattice identical to that in the cooling channel (section one), but without an r.f. system or absorber. The longitudinal size of the experimental hall will therefore have to be larger than the sum of the three sections (pre, first, post). To be safe, `lzAll` (half the full length) is slightly larger than two times the sum of the half lengths of each section:

```
G4double lzAll = 2. * (numCellPre * myperiodLengthPre +
numCell1 * myperiodLength1 + numCellPost * myperiodLengthPost) + 1000.;
```

where `numCell...` and `myperiodLength...` are the number of cells and half the length of each cell.

The user decides on the visualization properties of the logical volumes. The line `experimentalHall_log->SetVisAttributes (G4VisAttributes::Invisible)` would make the experimental hall invisible. To make if visible, replace that line by:

```
G4VisAttributes * dbVisAttHall
= new G4VisAttributes(G4Colour(1.0,1.0,1.0));
experimentalHall_log->SetVisAttributes(dbVisAttHall);
```

The experimental hall will be drawn white. Consult the Geant4 guide for color codes [3]. If you search the files with `MuCool` prefix in the `src` area for the key word `G4VIS_USE`, you will find all the code blocks associated with visualization statements. You can then change settings that suit your needs.

To finish with the general description of the MuCool structure, we will go over a set of geometry input parameters. In `MuCool.in`, there is a geometry block:

```
#
# Geometry
#
MoveZorigin 0.0
# offset of the channel origin with repect to the global origin (mm)
NumCellPeriodPre 5.0
# some solenoid cells to have a uniform field in channel
GapSectionPre 1.0;
# 1 mm gap between the four coils forming a cell
LatticePeriodPre 2420.0
# length of pre-section cell (in mm)
NumCellPeriod1 20.0
# number of cells in first section
GapSection1 1.0;
# 1mm gap between the four coils forming a cell (in sec 1)
LatticePeriod1 2420.0
# length of first-section cell (in mm)
NumCellPeriodPost 5.0
# some solenoid cells to have a uniform field in channel
```

```
GapSectionPost 1.0;
# 1mm gap between the four coils forming a cell
LatticePeriodPost 2420.0
# length of post-section cell (in mm)
#
# Maximum step sizes for lattice unit cell logical volumes
#
MaxStepSizePre 100.0
# max step size in mm for pre-section (for the RK integrator)
MaxStepSizeSec1 100.0
# max step size for first section (in mm)
MaxStepSizePost 100.0
# max step size for post section (in mm)
MaxStepSizeDefault 100.0
# max step size in the channel (in mm)
```

Although the user can change the length of the cooling channel (section one) or that of the pre and post sections, the origin of the global coordinate system will automatically be adjusted to be at the begining of the channel (section one). `MoveZorigin` can be used to shift the channel position with respect to the global coordinate system. The solenoid covers the full length of a unit cell (`LatticePeriod`...), but we have divided it in four solenoid units separated by a 1 mm `GapSection`... The gap values are negligible in the MuCool example, but the feature exists to allow the user some flexibility for introducing one or up to four solenoids per unit cell, as well as controling the size of the gaps between them. Although the unit cell is an abstract concept, it is defined as a solid volume `G4Tubs* aCellTubeSec1`, with associated logical (`G4LogicalVolume *aCellLogSec1`) and physical (`G4VPhysicalVolume *aCellPhysSec1`) objects in MuCool. Magnets, r.f. cavities and absorbers are placed inside this volume, as will be illustrated in subsequent sections.

```
G4Tubs* aCellTubeSec1 =
new G4Tubs("CellSection1",0., radInner+radBlock+5.0*extraR,
myperiodLength1, startZeroAngle, spanningAll360);

G4LogicalVolume *aCellLogSec1 =
new G4LogicalVolume(aCellTubeSec1, Vacuum, ostCellLogSec1.str() ,
0,0,0);

G4VPhysicalVolume *aCellPhysSec1 =
new G4PVPlacement(0, V3, ostCellPhysSec1.str(), aCellLogSec1,
expPhys , false, iCell);
```

In the MuCool example, the lattice unit cell solid is a cylinder (`G4Tubs* aCellTubeSec1`) filled with vacuum, with a radius slightly larger than the maximum radius of any accelerator element, and length equal to the lattice period. There is also a `G4LogicalVolume *aCellLogSec1` associated with `aCellTubeSec1`. The physical volume is placed many times (`iCell` is a loop control parameter over all cells) at the position `G4ThreeVector V3` of each cell with respect to the global coordinate system. The first parameter in the `G4PVPlacement` constructor is a pointer to a `G4RotationMatrix` object which allows to rotate the lattice unit cell with respect to the global coordinate system. (See Geant4 user's guide for usage [3]). These feature is useful, for example, to simulate closed orbit accelerators or, in general, any accelerator which is not straight.

7.4.2 The Solenoids

Here we will tell you how to simulate realistic solenoids from current distributions. `BT-v1.50` provides the user with classes to construct `BTSheet`, `BTSolenoid`, `BTSolenoidLogicVol` and `BTSolenoidPhysVol` objects. `BT-v1.50` also provides a class to read a generic magnetic field produced or calculated externally (see Section 7.4.4 for details). For information on these `BT` classes, see the reference guide in Section 9.

`BTSheet` objects are a set of parameters necessary to generate analytically the magnetic field for an infinitecimally thin solenoidal current sheet. `BTSolenoid` objects are field maps in the form of a grid in r-z space. They correspond to solenoid coils of finite thickness made of a set of concentric acurrent sheets `BTSheet`. The `BTSolenoidLogicVol` class defines objects containing the material and physical size of the coil system which generates the `BTsolenoid` field from the `BTSheets`. The `BTSolenoidPhysVol` class is the placed version of the `BTSolenoidLogicVol` object.

The parameters necessary to construct the solenoids are provided by the user through the sheet/solenoid block in `MuCool.in`:

```
#
# Sheet/Solenoid information
#
SheetLength 602.5
# length of sheet (4 per cell) in mm
NumberofSheets 2.0
# number of concentric sheets to form a thick coil
InnerRadius 710.0
# sheet inner radius (in mm)
BlockRadius 20.0
```

```
# block thickness (in mm)
BlockCurrent 120.4
# block current (in A/mm**2)
MakeVolumeSheet 1
# to draw each of the sheets
SolDataFiles useFiles
# other than ''none'' means the field is read from a file
#SolDataFiles none
# none means the field is constructed
GridLengthR 710.0
# radial length of the field grid in mm
GridLengthZ 10.0
# longitudinal length of the field grid (in number of radii)
NumberNodesRGrid 100.0
# number of nodes of the grid in R
NumberNodesZGrid 1000.0
# number of nodes of the grids
StepSizeInSol 100.0
# maximum step size for the Runge-Kutta integrator (in mm) in solenoid
```

The process of simulating a solenoid starts with the construction of the `BTSheets` in `MuCoolConstruct.cc`:

```
for (G4int l=0; l<solnsheets; ++l)
{
radSheet = radInner + (((G4double) l) + 0.5) *radBlock/solnsheets;
idSheet = 1;
vsheets.push_back(BTSheet(G4ThreeVector(0.,0.,0.), idSheet,
typeSheet, thicksheet, radSheet, lenSheet, -curSheet));
}
```

The loop is performed over a number `solnsheets` of infinitecimally thin and concentric `BTSheets`. The sheets radii `radSheet` are equally spaced to model a magnet coil of thickness `radBlock`. The `BTSheet` constructor takes the sheet position in local coordinates of the solenoid (`G4ThreeVector(0.,0.,0.)` for concentric sheets), the `idSheet` and `typeSheet` (irrelevant in `BT-v1.50`), the sheet thickness `thicksheet` (only zero is supported), the radius (`radSheet`), the full length (`lenSheet`), and the one dimensional current density in Ampere/mm^2 (`curSheet`). The `push_back` method stores one by one the sheets into a `vsheets` vector of `BTSheets` which will make the solenoid. Figure 9 shows an image of a sheet system.

If the parameter `SolDataFiles` is set to `useFiles`, then a `BTSolenoid` magnetic field map object is constructed from a binary file `MuCoolSol.dat` in `$G4WORKDIR/MuCool/bin/Linux-g++` using:

```
solSH = new BTSolenoid("MuCoolSol.dat");
```

If it is set to `none`, the solenoid map is constructed from the vector of `BTSheets` and written into a `MuCoolSol.dat` file for subsequent use. The object `solSH` is just a field map with no associated volume or material.

```
solSH = new BTSolenoid(0., solmaxrxy, solnumptrxy,-solmaxz, solmaxz,
solnumptz, vsheets);
```

The field is stored as a grid in r-z space (data member arrays), where `solnumptz` and `solnumptrxy` are the number of z and r nodes, `[0.,solmaxrxy]` and `[-solmaxz, solmaxz]` define the domain of the grid. The parameters associated with a spline fit of $B_z(z)$ and $B_r(z)$ at fixed values of r (nodes) are also stored as `BTSpline1D` members of `BTSolenoid`. From a linear interpolation of the spline fits at a fixed r and z location of the trajectory of the particle, $B_r(r, z)$ and $B_z(r, z)$ are calculated and provided to the Geant4 tracking code. The more nodes in z, the better the accuracy of the spline fit; the more nodes in r, the better the accuracy of the interpolation. To ensure good accuracy, the field map should extend well beyond the physical limits of the magnet, since the field at a given point in space is the sum of contributions from all magnets in the lattice. Note that `solmaxrxy` is provided through `GridLengthR` in milimeters, but `solmaxz` is provided through `GridLengthZ` in number of sheet radii.

The next step is to construct the logical volume of a solenoid, that is a concrete coil system associated with the `solSH` field. The `BTSolenoidLogicVol` constructor needs the field grid `solSH`, and the coil material (copper). It may also have shielding for the coils, although the associated parameters are set to zero and Vacuum in the MuCool example. `ostSolLogPre.str()` is the name of the solenoid volume. The next two zeros reflect the fact that no extra longitudinal or radial space is needed when there is no shielding. The third zero is the value of the shielding thickness parameter. The `false` argument means that the solenoid tube and associated logical volume is a ring encompassing the coils and shielding, not a solid cylinder as `true` would mean. The `makeSheet` boolean variable is related to the input parameter `MakeVolumeSheet`. If the former is `true` (the latter would be 1), then the sheets forming the solenoid are drawn in the visualization window. `solmaxstep` is the maximum step length associated with the solenoid logical volume. Figure 9 shows a solenoidal copper coil system modelled with four infinitecimally thin sheets equally spaced in radius.

```
solLogSec1 = new BTSolenoidLogicVol(solSH, Copper, 0, Vacuum,
ostSolLogSec1.str(), 0., 0., 0., false, makeSheet, solmaxstep);
```

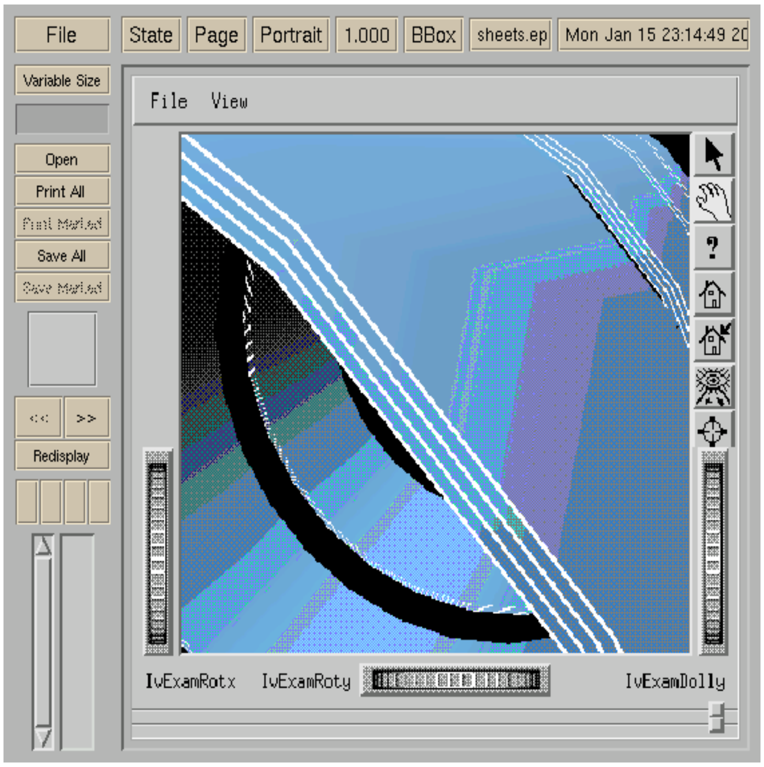


Fig. 9. A solenoidal copper coil system modelled with four infinitecimally thin sheets equally spaced in radius. Some changes were made to the default `MuCool.in` parameters: the number of sheets was increased to four, and the sheet length was decreased for increasing the gaps. The knee on the surface of the sheet is an artifact of the visualization tool.

The last step is to place the four solenoid sections inside the lattice unit cell separated by gaps, using the `BTSolenoidPhysVol` constructor. The first parameter, set here to zero, is a pointer to a `G4RotationMatrix` object. By rotating individual solenoids independently with respect to the lattice unit cell, the user can create a set of short "tilted" solenoids adding a dipole component to the magnetic field. To make "ring" accelerators, however, the best option is to rotate the lattice unit cells with respect to the global coordinate system and leave the actual solenoids unrotated, as discussed in Section 7.4.1. `V3sheet` is the position of the solenoid with respect to the center of the lattice unit cell volume. The `BTSolenoidPhysVol` constructor also needs a pointer to its `BTSolenoidLogicVol *solLogSec1` and its "mother" volume `aCellPhysSec1`, because it is placed inside the lattice unit cell. Other input parameters are the copy number `0` through `3`, a scale factor `-1` to increase, reduce the field strength, or change its sign, and a pointer to the global magnetic field `magFF`. The code below shows how to place a set of four solenoids in a single lattice unit cell, separated by a short gap, `gapSec1=1mm`. Figure 10 shows a similar configuration, with longer gaps.

```
offset = -myperiodLengthSec1 + 0.5*lenSheet + extraZ;
V3sheet[2] = offset;
BTSolenoidPhysVol *aSolPhysSec1sh1 =
new BTSolenoidPhysVol(0, V3sheet, solLogSec1, aCellPhysSec1,
false, 0 , -1.0 , magFF);

offset = offset + gapSec1 + lenSheet;
V3sheet[2] = offset;
BTSolenoidPhysVol *aSolPhysSec1sh2 =
new BTSolenoidPhysVol(0, V3sheet, solLogSec1, aCellPhysSec1,
false, 1 , -1.0, magFF);

offset = offset + gapSec1 + lenSheet;
V3sheet[2] = offset;
BTSolenoidPhysVol *aSolPhysSec1Sh3 =
new BTSolenoidPhysVol(0, V3sheet, solLogSec1, aCellPhysSec1,
false, 2 , -1.0, magFF);

offset = offset + gapSec1 + lenSheet;
V3sheet[2] = offset;
BTSolenoidPhysVol *aSolPhysSec1Sh4 =
new BTSolenoidPhysVol(0, V3sheet, solLogSec1, aCellPhysSec1,
false, 3 , -1.0, magFF);
```

In summary, the user must create only one `BTSolenoid` and its associated `BTSolenoidLogicVol` per type of magnet in the channel. Each solenoid type is placed as many times as needed (using the `BTSolenoidPhysVol` constructor). It is possible to control the strength and sign of the magnetic field at this stage.

7.4.3 r.f. Systems

This section tells how to simulate realistic r.f. systems using resonant cavities. `BT-v1.50` provides classes to construct `BTPillBox`, `BTrfCavityLogicVol`, `BTrfWindowLogicVol` and `BTLinacPhysVol` objects. `BT-v1.50` also provides a class to read a generic magnetic field produced or calculated externally (see Section 7.4.5 for details). For information on these `BT` classes, see the reference guide in Section 9. `BTPillBox` is a class of Pill Box cavities. The associated logical volume is created with the `BTrfCavityLogicVol` constructor. For better cavity performance (increased shunt impedance), threre is a `BTrfWindowLogicVol` class to cover the cavity iris with thin windows. Placement is done with the `BTLinacPhysVol` class by positioning a Linac, that is an array of identical cavities.

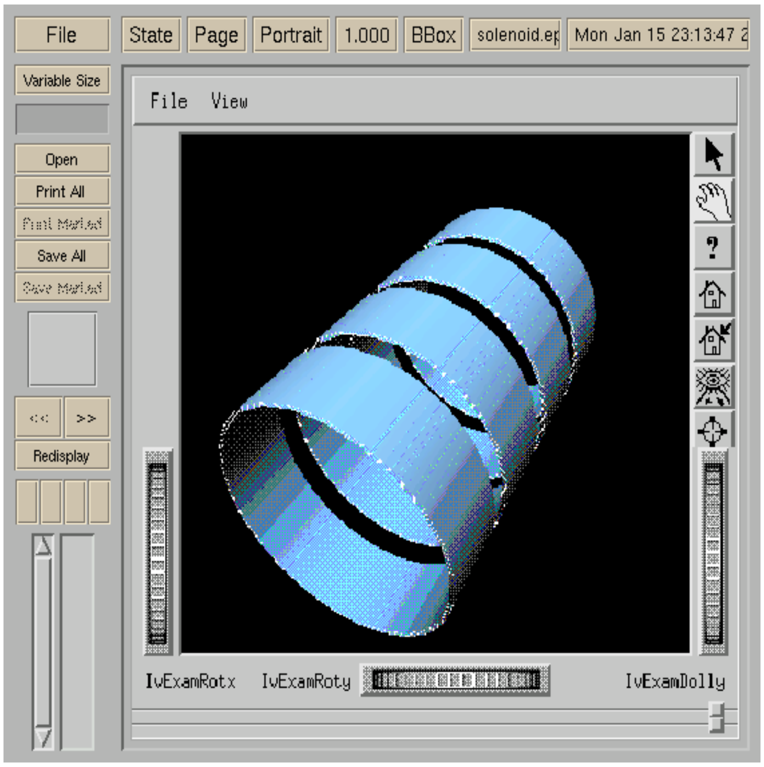


Fig. 10. Array of four solenoids in a single Double Flip unit cell. The gaps between solenoids are larger than the 1 mm default value in the MuCool example.

In `MuCoolConstruct::Construct()`, every r.f. related statement is inside an `if` block controled by the input parameter `rfCellType`. If `rfCellType` is not ``` ``PillBox'' ``` or ``` ``rfmap'' ```, the r.f. system is not built and the reference particle is not processed (no need for phase tuning).

A Pill Box cavity is a cylinder of radius R and length L made of a conductor material. In real life, the beam pipe goes through the cavity hole (iris) in the end cups of the cavity. `BT-v1.50` provides the option to place thin windows to close the iris and obtain electric fields closer to the ideal Pill Box fields. The cavity fields, however, correspond to the ideal case even if windows are not placed. The only objective of the windows in the simulation is, therefore, to account for the interactions between the beam and the window material. The Pill Box fields are given by:

$$E_{\mathrm{z}} = V_{\mathrm{p}}\, J_0\left(\frac{2\pi\nu}{c} r\right) \sin(\phi_s + 2\pi\nu t) \tag{1}$$

$$B_{\varphi} = \frac{V_{\mathrm{p}}}{c} J_1\left(\frac{2\pi\nu}{c} r\right) \cos(\phi_s + 2\pi\nu t) \tag{2}$$

where V_{p} is the cavity peak voltage, ν the wave frequency, ϕ_s the synchronous phase, and $J_{0,1}$ the Bessel functions evaluated at $\left(\frac{2\pi\nu}{c} r\right)$. The radius of the cavity is derived from:

$$\frac{V_{\mathrm{p}}}{c} J_1 \left(\frac{2\pi\nu}{c} r \right) = 0 \quad \text{at} \quad r = R \tag{3}$$

$$\left(\frac{2\pi\nu}{c} R \right) = 2.405 \;\Rightarrow\; R = \frac{2.405\, c}{2\pi\nu} \tag{4}$$

but the length, L, must be calculated and provided by the user as an input parameter. L is a function of particle velocity (v), r.f. phase advance, and cavity frequency:

$$\frac{2\pi\nu}{v} L = \frac{\pi}{2} \;\text{(phase advance)} \;\Rightarrow\; R = \frac{v}{4\nu} \tag{5}$$

The r.f. input parameters are controled from the r.f block in `MuCool.in`:

```
#
# r.f. system information
#
StepSizeInRf 100.0
# maximum step size for the Runge-Kutta integrator (in mm) in r.f
rfWindowMaterial Beryllium
# material for rf window cavities
rfFrequencySec1 0.20125
# frequency of rf cavities (in GHz)
rfPeakFieldSec1 0.01648
# peak voltage of rf cavities (in MV/mm)
rfCellLengthSec1 320.0
# rf cavity length for phase advance pi/2 (in mm)
rfAccelerationPhaseSec1 0.4451
# rf cavity synchronous phase (in radians)
rfCellSkinDepthSec1 0.005
# skin depth for wall effects (in mm)
rfNumCavPerLinacSec1 6.
# number of rf cavities in linac (per cooling cell)
ReferenceEGainPerLinacSec1 12.90
# gain per linac (in MeV)
rfWallThickSec1 5.0
# rf cavity wall thickness (in mm)
rfWindowThickSec1 0.300
# window thickness-inner circle (in mm)
rfWindowradiusSec1 160.0
# window radius (in mm)
```

```
rfWindowOutThickSec1 0.600
# window thickness-outer ring (in mm)
rfWindowRoRO-Sec1 0.7
# fraction of radius defining inner and outer rings
# in step windows
```

Most of the parameters are self explanatory. `ReferenceEGainPerLinacSec1` is an estimated energy gain per lattice cell by the accion of the linac. This parameter is used in reference particle mode to tune the cavity phases to operate at synchronous phase at the time when the particle traverses the geometric center of the cavity. The `ReferenceEGainPerLinacSec1` parameter will be revisited in Section 7.4.6. `rfWindowThickSec1` and `rfWindowradiusSec1` referes to the thickness and radius of the window covering the cavity iris. There is the possibility of implementing a "step window", as a simple way to model a radius dependent thickness. For that, a second "ring" window with inner radius of `rfWindowRoR0-Sec1` × `rfWindowradiusSec1`, outer radius `rfWindowradiusSec1`, and thickness `rfWindowOutThickSec1` must be placed contiguous to the first window. Figure 11 shows a single r.f. cavity (in red), with an outer window ring (dark green) and an inner full window (light green). When a linac of more than two cavities is placed, contiguous cavities share a window.

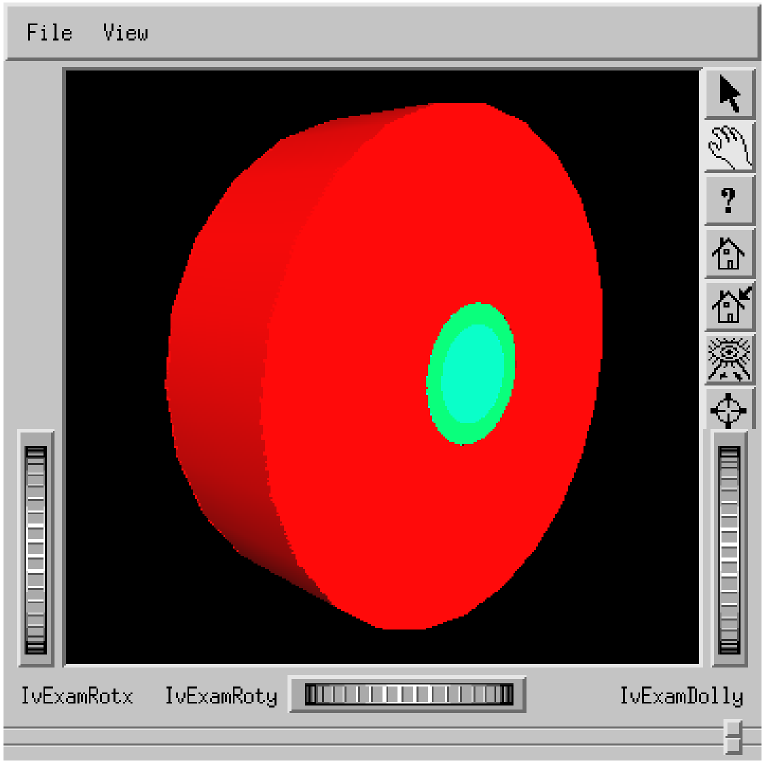


Fig. 11. A single Pill Box cavity (in red), with an outer window ring (dark green) and an inner full window (light green).

The following lines summarize the construction of the r.f. system (a linac) contained in a lattice unit cell:

```
BTPillBox *aRFEMPillBoxSec1;
BTrfCavityLogicVol *aPillBoxLogSec1;
BTrfWindowLogicVol *aRFWindowSec1;

// Read information necessary to construct a Pill Box cavity.

freqInSec1 = MyDataCards.fetchValueDouble("rfFrequencySec1");
lengthPBSec1 = MyDataCards.fetchValueDouble("rfCellLengthSec1");
skinDepthSec1 = MyDataCards.fetchValueDouble("rfCellSkinDepthSec1");
eFieldMaxGradSec1 = MyDataCards.fetchValueDouble("rfPeakFieldSec1");
phaseAccSec1=MyDataCards.fetchValueDouble("rfAccelerationPhaseSec1");

// Construct a Pill Box cavity (it is a cylinder).

aRFEMPillBoxSec1 = new BTPillBox (freqInSec1, lengthPBSec1,
skinDepthSec1, eFieldMaxGradSec1, phaseAccSec1);
```

First, we create the three relevant objects to built the cavity electric electric field: `BTPillBox`, `BTrfCavityLogicVol`, and `BTrfWindowLogicVol`. The `BTPillBox` constructor takes the following arguments: cavity frequency, length, depth inside the walls the electric field penetrates, peak voltage, and synchronous phase. The `BTrfCavityLogicVol` constructor needs a pointer to the associated `BTPillBox` object, the cavity walls material (copper), the cavity inside material (vacuum), the logical volume name, the extra length necessary to accomodate the support structure (end cups or window rims, cooling devices, etc), the wall thickness, and the maximum step size associated with the volume:

```
aPillBoxLogSec1 = new BTrfCavityLogicVol( aRFEMPillBoxSec1, Copper,
Vacuum, ostRFLogSec1.str() , extraPBz, wallThickSec1,rfmaxstep );
```

As a rule, the logical volume name must contain the “RF” string. This is a requirement for the automatic phase tuning to function in reference particle mode. The window solids and logical volumes are created in the `BTrfWindowLogicVol` constructor:

```
aRFWindowSec1 = new BTrfWindowLogicVol( winRadSec1 ,
(rRF + wallThickSec1), winThickSec1, wallThickSec1,
rfWindowMat, Vacuum, ostWinLogSec1.str());
```

The arguments are the window radius, the outer radius of the rim (window support structure), the window thickness, the rim thickness, the window material, the material filling the top volume which contains the window structure (vacuum), and the name of that volume. To use the “step window” option, we need to create the outer ring window logical volume:

```
aRFWindowSec1->AddOuterFoil(ror0Sec1*winRadSec1, thickOutSec1,
rfWindowMat);
```

The first argument takes the ring window inner radius, the second its thickness, and the last its material.

Finally, a linac composed of a number `nnn` of cavities is placed by a call to the `BTLinacPhysVol` constructor:

```
for (G4int ic1=1; ic1 < nnn; ++ic1)

{
zLocsCavsSec1[ic1] = zLocsCavsSec1[0] + ll1*ic1;
}
aLinacSec1 = new BTLinacPhysVol(nnn, zLocsCavsSec1,
aPillBoxLogSec1, aRFWindowSec1, aCellPhysSec1, fullEMField );
```

where `zLocsCavsSec1` is the array which contains the z positions of the `nnn` cavities with respect to the geometric center of the linac. Since the linac is placed in the lattice unit cell, its constructor also takes a pointer `aCellPhysSec1` to the unit cell physical volume. The last argument is a pointer to the global electromagnetic field `fullEMField`.

7.4.4 Magnetic Field Maps

`BT-v1.50` provides the `BTMagFieldMap` class to include user defined field maps for magnets in the simulation. Although `BTMagFieldMap` is restricted to fields with azimuthal symmetry, the user may create a class for arbitrary maps following the same model. Note that `BTMagFieldMap` is a field object not associated with a solid (no coil structure is modelled), in contrast with the case of the cavity maps which are associated with a dummy structure as will be explained in Section 7.4.5.

The user provides an `ASCII` file with a field grid in (r,z) space, following the format:

z(cm)	r(cm)	B_z(KiloGauss)	B_r(KiloGauss)
z_0	r_0	...	...

z_1	r_0	...	...
...	r_0	...	...
z_n	r_0	...	...
...	...	...	...
...	...	...	...
z_0	r_m	...	...
...	...	...	...
z_n	r_m	...	...

The `BTMagFieldMap` constructor must be called in the detector user method `MuCoolConstruct::Construct()`, as shown below:

```
//std::string typeofmap="HardEdge";
std::string typeofmap="Interpolated";
BTMagFieldMap *aMagFieldmapSec1;
aMagFieldmapSec1 = new BTMagFieldmap (‘‘HardEdgeBF.dat’’,
typeofmap, mfzoffSec1, mfzlgthSec1, mfrlgthSec1,numMBnodesZSec1,
numMBnodesRSec1 );

//G4Tubs* aCellTubeSec1
// = new G4Tubs("CellSection1",0., mfrlgthSec1 + 5.0*extraR,
//periodLength1, startZeroAngle, spanningAll360);

//double sdradSec1 = MyDataCards.fetchValueDouble("SensDetradSec1");
//double sdlenSec1 = MyDataCards.fetchValueDouble("SensDetlenSec1");

//G4Tubs* aSensSec1 = new G4Tubs("Sensor", 0., sdradSec1,
sdlenSec1/2., startZeroAngle, spanningAll360);

//G4LogicalVolume *aSensLogSec1
// = new G4LogicalVolume(aSensSec1, this->theVacuum, "SensSec1" ,
//0,0,0);

// aSensLogSec1->SetUserLimits(new G4UserLimits(maxStepsens));
```

The type of magnet must be “HardEdge” or “Interpolated”. In the first case, the field is constant in between (r,z) nodes, taking the B_r and B_z value at the lower edge of the interval. In the second case, the field is evaluated in between nodes using a linear interpolation. The “HardEdge” option is useful, for example, to create square fields. The “Interpolated” option is aimed to reproduce accurately an arbirary field. The larger the node density, the best the linear interpolation works. `mfzoffSec1` is the z offset if the local field map z-origin is not located at the geometric center of the map. The map length, radius, and number of nodes in both directions must also be provided. The uncommented lines in the previous example would create an interpolated magnetic field. To create a square field, the user must comment the `typeofmap=''Interpolated''` line and uncomment all the others. The `aSens...` objects are created as artificial boundaries, coincident with field transition regions, to change (reduce) the maximum step size at the field flips and avoid Geant4 to "miss" them. For example, a square field has very short non-zero B_r components at the B_z flip regions which will be missed if a virtual detector is not placed there to force the Geant4 tracking code to make a step at the boundary.

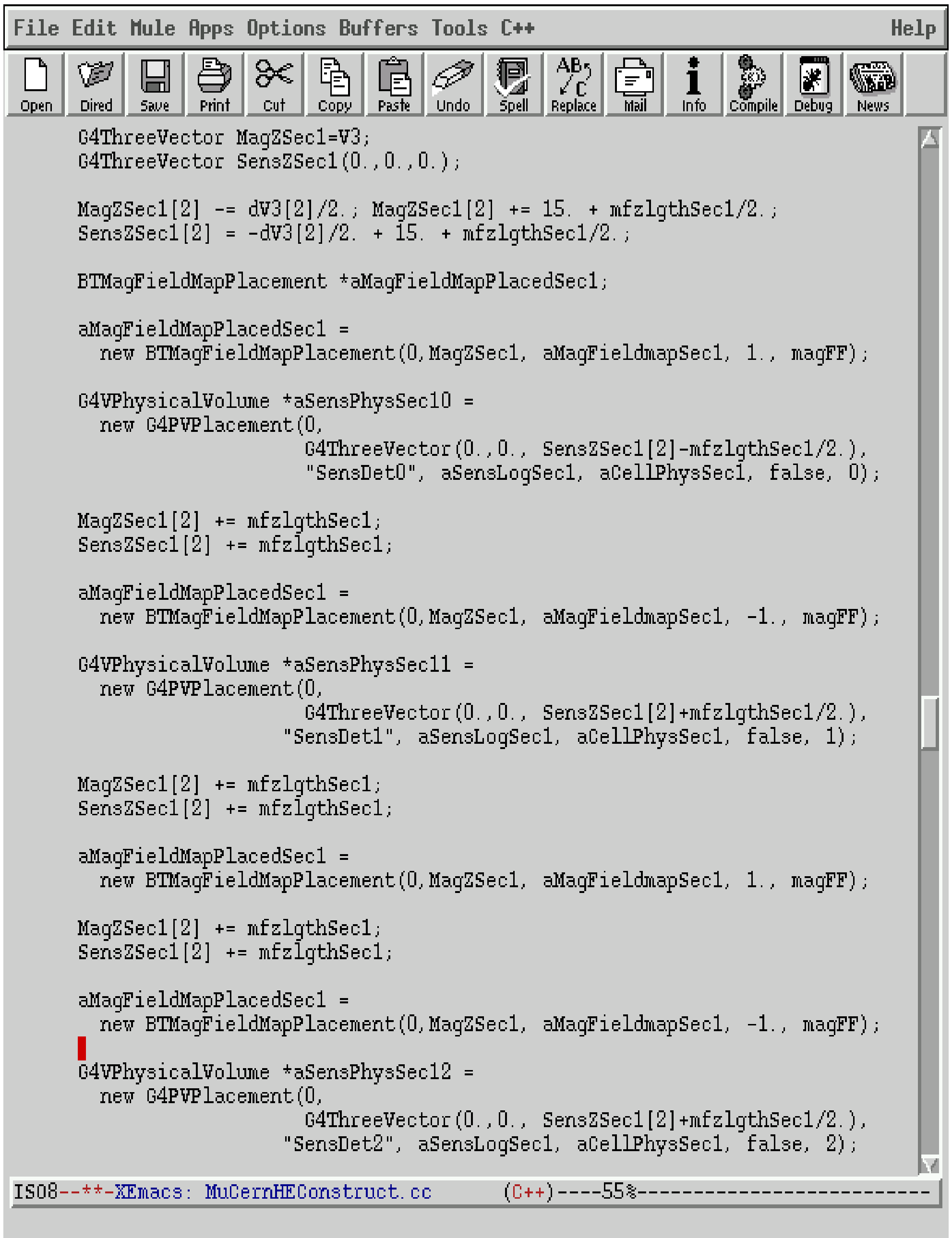
As shown in Fig. 12, the magnetic field map is placed by calling `BTMagFieldMapPlacement` with the first argument being a pointer to a `G4RotationMatrix` object for a rotation of the field with respect its geometric center. `MagZSec1` is a `G4ThreeVector` object which contains the global coordinates of the field geometric center, `aMagFieldmapSec1` is a pointer to the field map which is being placed, `1` is the field scaling factor, and `magFF` is the Geant4 global magnetic field object. The code in Fig. 12 corresponds to the placement of the square magnetic field described in Ref. [8]. B_z is shown in Fig. 13. B_r, in Fig. 14, is calculated from B_z to be consistent with the Maxwell Equations (see Ref. [8] for details). Sensitive detectors are placed at the B_z transition regions where B_r is non-zero. The first few lines of the `ASCII` file containing the field grid in this example are:

```
-50 0 0 0
-49.5 0 20 0
-49 0 20 0
-48.5 0 20 0
-48 0 20 0
-47.5 0 20 0
-47 0 20 0
-46.5 0 20 0
-46 0 20 0
-45.5 0 20 0
```

following the format described before. z goes from -50 cm to 50 cm, r from 0 to 30 cm, B_z is either 20, or 0 KiloGauss, and B_r is in the range [-600,600] KiloGauss depending on r and z. Note that only the $B_z > 0$ range is needed in the file, since the negative range (the other half of the wave) is constructed as a separate map using a `-1` scaling factor argument in `MagFieldMapPlacement`.

7.4.5 Electric Field (r.f.) Maps

`BT-v1.50` provides the `BTrfMap` class to include user defined field maps for




```
G4ThreeVector MagZSec1=V3;
G4ThreeVector SensZSec1(0.,0.,0.);

MagZSec1[2] -= dV3[2]/2.; MagZSec1[2] += 15. + mfzlgthSec1/2.;
SensZSec1[2] = -dV3[2]/2. + 15. + mfzlgthSec1/2.;

BTMagFieldMapPlacement *aMagFieldMapPlacedSec1;

aMagFieldMapPlacedSec1 =
  new BTMagFieldMapPlacement(0,MagZSec1, aMagFieldmapSec1, 1., magFF);

G4VPhysicalVolume *aSensPhysSec10 =
  new G4PVPlacement(0,
                    G4ThreeVector(0.,0., SensZSec1[2]-mfzlgthSec1/2.),
                    "SensDet0", aSensLogSec1, aCellPhysSec1, false, 0);

MagZSec1[2] += mfzlgthSec1;
SensZSec1[2] += mfzlgthSec1;

aMagFieldMapPlacedSec1 =
  new BTMagFieldMapPlacement(0,MagZSec1, aMagFieldmapSec1, -1., magFF);

G4VPhysicalVolume *aSensPhysSec11 =
  new G4PVPlacement(0,
                    G4ThreeVector(0.,0., SensZSec1[2]+mfzlgthSec1/2.),
                  "SensDet1", aSensLogSec1, aCellPhysSec1, false, 1);

MagZSec1[2] += mfzlgthSec1;
SensZSec1[2] += mfzlgthSec1;

aMagFieldMapPlacedSec1 =
  new BTMagFieldMapPlacement(0,MagZSec1, aMagFieldmapSec1, 1., magFF);

MagZSec1[2] += mfzlgthSec1;
SensZSec1[2] += mfzlgthSec1;

aMagFieldMapPlacedSec1 =
  new BTMagFieldMapPlacement(0,MagZSec1, aMagFieldmapSec1, -1., magFF);

G4VPhysicalVolume *aSensPhysSec12 =
  new G4PVPlacement(0,
                    G4ThreeVector(0.,0., SensZSec1[2]+mfzlgthSec1/2.),
                  "SensDet2", aSensLogSec1, aCellPhysSec1, false, 2);
```

Fig. 12. Placement of a magnetic field map (hard edge square field). See Ref. [8] for details.

cavities in the simulation. Although `BTrfMap` is restricted to fields with azimuthal symmetry, the user may create a class for arbitrary maps following the same model. In contrast to `BTMagFieldMap`, `BTrfMap` is associated with a solid, as will be explained below. To include field maps in the simulation, the input argument `rfCellType` must be set to ``rfmap''. The format for the user defined `ASCII` files is:

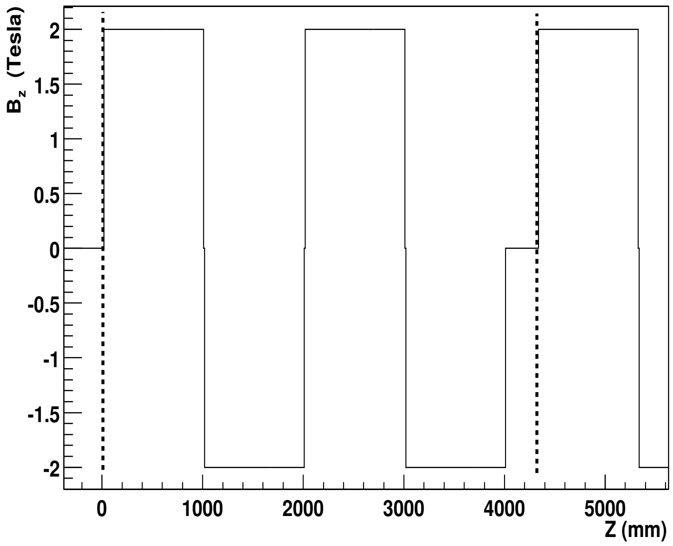


Fig. 13. B_z versus z. Note the zero field regions at the transition points.

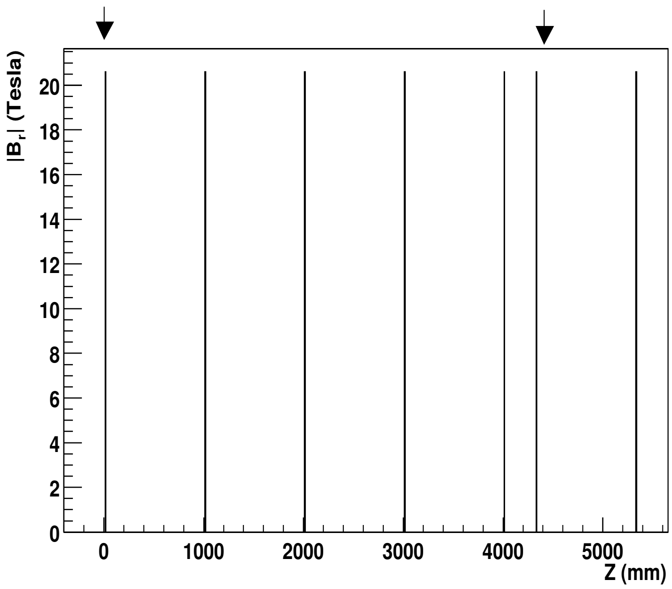


Fig. 14. $|B_r|$ versus z at r=10.3 cm.The spikes are 0.5 cm long at z=0, ≈4 m, and ≈4.32 m, where B_z changes by ±2 T. The spikes are 1 cm in the other locations, where B_z changes by ±4 T. The arrows show the physical limits of a lattice unit cell.

z(cm)	r(cm)	E_z(MV)	E_r(MV)
z_0	r_0	...	...
z_1	r_0	...	...
...	r_0	...	...
z_n	r_0	...	...
...	...	...	...

...	...	...	...
z_0	r_m	...	...
...	...	...	...
z_n	r_m	...	...

The BTrfMap constructor must be called in the detector user method MuCoolConstruct::Construct(), as shown below:

```
BTrfMap *aRFEMmapSec1; BTCavityLogicVol *aRFmapLogSec1;

rfmaxstep = MyDataCards.fetchValueDouble("StepSizeInRf");
freqInSec1 = MyDataCards.fetchValueDouble("rfFrequencySec1");
rfzoffSec1 = MyDataCards.fetchValueDouble("rfzoffsetSec1");
lengthMAPSec1 = MyDataCards.fetchValueDouble("rfCellLengthSec1");
rMaxSec1 = MyDataCards.fetchValueDouble("rfMaximumRadiusSec1");
rEffSec1 = MyDataCards.fetchValueDouble("rfEffectiveRadiusSec1");
wallThickSec1 = MyDataCards.fetchValueDouble("rfWallThickSec1");
phaseAccSec1 =
MyDataCards.fetchValueDouble("rfAccelerationPhaseSec1");
numRFnodesZSec1 =
(int) MyDataCards.fetchValueDouble("rfNumNodesZSec1");
numRFnodesRSec1 =
(int) MyDataCards.fetchValueDouble("rfNumNodesRSec1");
double ZPhaseSec1 = MyDataCards.fetchValueDouble("rfZPhaseSec1");

aRFEMmapSec1 = new BTrfMap ("HardEdgeEF.dat", freqInSec1, rfzoffSec1,
ZPhaseSec1, lengthMAPSec1, rMaxSec1, rEffSec1, phaseAccSec1,
numRFnodesZSec1, numRFnodesRSec1);
```

where the arguments are the name of the ASCII file containing the electric field grid, the cavity frequency, the z-offset of the map coordinate system with respect to the geometric center of the cavity, the z position at which the synchronous phase is defined (zero if the cavity is phased at its geometric center), the length and radius of the cavity solid object, the effective radius of the map used when retrieving the field (can be smaller than the cavity radius), the synchronous phase, and the number of nodes of the grid in (r,z) space. Unlike BTMagFieldMap, BTrfMap is associated with a logical volume BTrfCavityLogicVol. BTrfCavityLogicVol is a cylindric tube bounded by a cylindric ring (wall) with the same geometric disposition as the Pill Box conductor ring, except that it is made of vacuum. Although this structure may have nothing to do with the geometry of the real cavity which produced the field, this object allows the user to visualize the boundaries of the r.f. field. In addition, it provides a dummy software structure to include windows, if necessary.

```
aRFmapLogSec1 = new BTrfCavityLogicVol( aRFEMmapSec1, Vacuum,
Vacuum, ostRFLogSec1.str() , extraMAPz, wallThickSec1,
rfmaxstep);
```

Following the same model as for Pill Box cavities, a linac composed of `nCellPerLinacSec1` cavity maps is placed by a call to the `BTLinacPhysVol` constructor:

```
for (int ic1=1; ic1 < nCellPerLinacSec1; ++ic1)
{
zLocsCellsSec1[ic1] = zLocsCellsSec1[0] + ll1*ic1;
}
aLinacSec1 = new BTLinacPhysVol(nCellPerLinacSec1, zLocsCellsSec1,
aRFmapLogSec1, 0., ZPhaseSec1, aCellPhysSec1, fullEMField );
```

The argument which takes the pointer to a window logical volume is set to `0`, meaning that no windows are modelled in the example. `ZPhaseSec1` is the distance between the phase and the geometric centers of the cavity (zero if they are coincident). The other arguments are the same as in the Pill Box case.

As an example of the above, Fig. 15 shows a lattice cell of a cooling channel where a solenoid is embeded in a large low frequency (44 MHz) cavity. The electric field map represented by the red cylinders was made available in grid format to the `BTrfMap` constructor. Since the beam circulates inside the solenoid, the field map was restricted to a cylindric volume with radius slightly smaller than the inner radii of the magnets. The geometry is illustrated in Fig. 16. For details on this example, see Ref. [8].

`BT-v1.50` also allows to simulate "instantaneous" kicks using thin cavities. The first few lines of an `ASCII` file with an example are shown below. The file contains the field grid for a 1 cm long cavity, which provides a gradient of 200 MV/m.

```
-0.5 0. 200. 0.
0.5 0. 200. 0.
-0.5 1. 200. 0.
0.5 1. 200. 0.
-0.5 2. 200. 0.
0.5 2. 200. 0.
```

In this case, a more realistic equivalent acceleration device would be a 1 m long cavity providing a gradient of 2 MV/m.

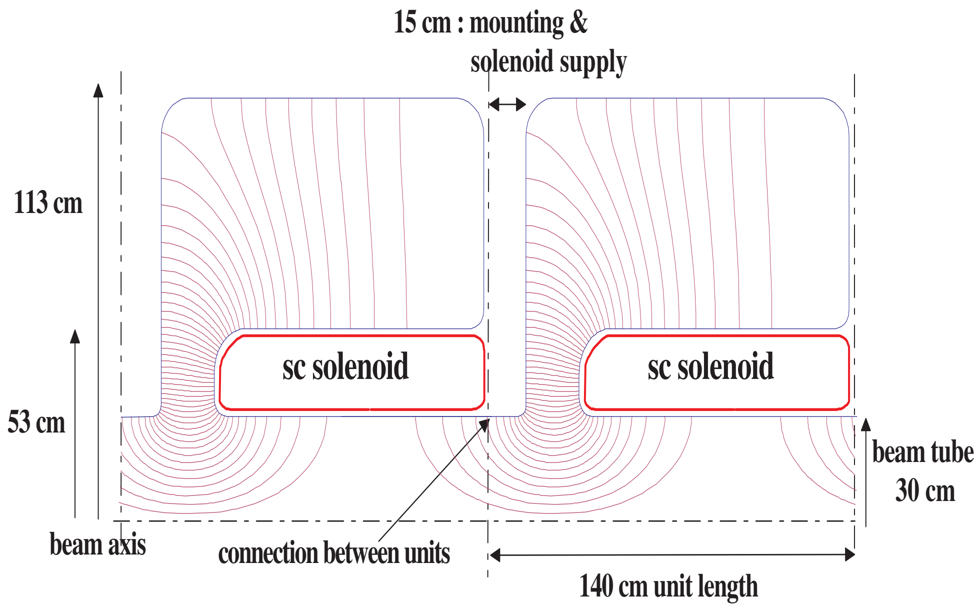


Fig. 15. Unit cell of a cooling channel where a solenoid is embeded in a large low frequency (44 MHz) cavity.

7.4.6 Tuning the r.f. Cavity Phases

One of the critical elements of an accelerator simulation is the "r.f. tuning". Each cavity must be operated at the selected synchronous phase at an instant coincident with the passage of the beam. The r.f. wave must be therefore synchronized with the beam, more specifically, with the region of beam phase space that the user needs to manipulate. For this, there is the concept of a reference particle, which typically takes the average characteristics of the beam. If the kinematic and dynamic variables of the reference particle are set to values which are coincident with the mean values of the corresponding variables for the beam, the r.f. system should affect the mean beam properties in a similar way it affects the reference particle. Note that the r.f. wave does not necessarilly have to be tuned to follow the mean velocity of the beam. Different applications may need a reference particle to represent the leading edge, the trailing edge, or any other sub-range of the total beam phase space.

The MuCool example shoots a "reference particle" to tune the r.f. system before processing the beam. The time instants the particle goes through the phase center of each cavity are calculated, displayed on screen, and used to adjust each cavity phase to provide the proper kick, at the selected synchronous phase. If the accelerator contains absorber elements, stochastic processes like multiple scattering and straggling must be turned off as the reference particle goes through the system. The setup and shooting of the reference particle is controled from `MuCool.cc`, as shown in Fig. 17.

The user sets the kinetic energy (MeV) of the reference particle. As explained

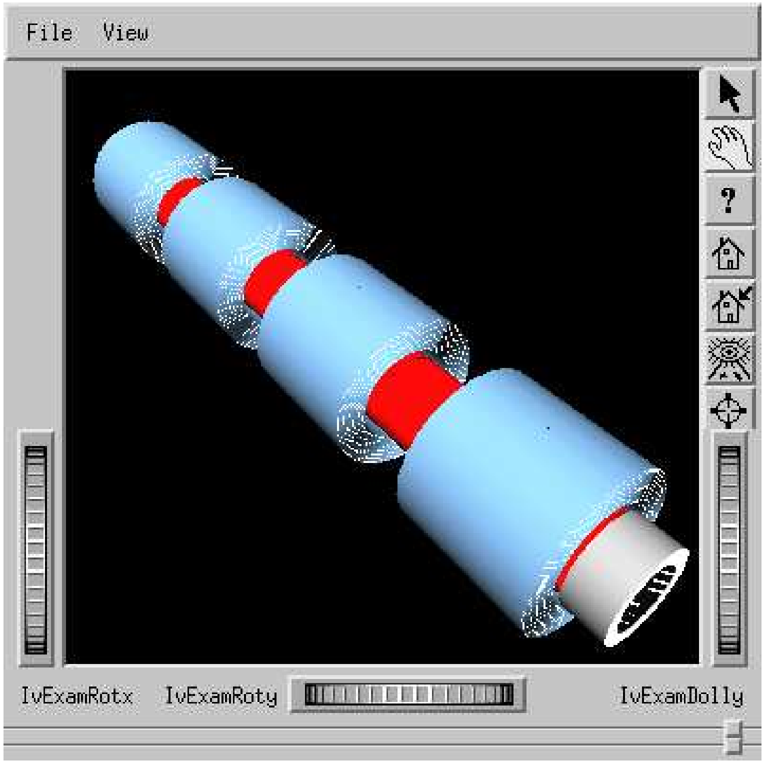


Fig. 16. Geometry of a low frequency cooling channel as simulated in Geant4. The red cylinders are the dummy software structure representing the limits of the electric field map.

in Section 7.3, this parameter is provided to the `GeneratePrimaries` method in `MuCoolPrimaryGeneratorAction`:

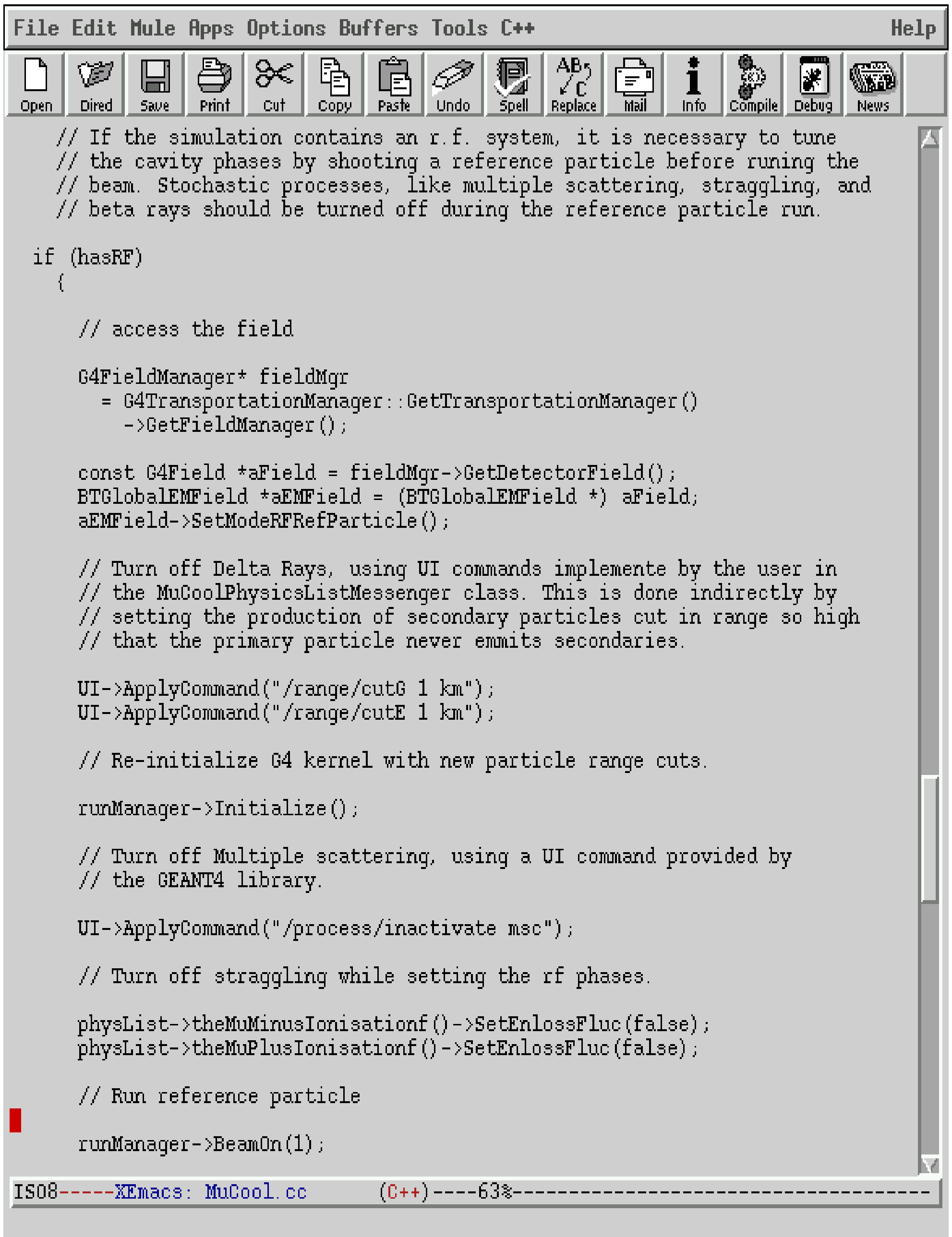
```
ReferenceKineticEnergy 94.35
```

The user must also calculate and input the total energy (in MeV) contributed by each linac operated at the selected synchronous phase. This is done in the r.f. block of the input parameter file:

```
ReferenceEGainPerLinacSec1 12.90
```

Figure 18 shows the reference particle trace through the MuCool example channel. You can see how the energy of the particle increases as it goes through the six cavity linacs (12.9 MeV) and then decreases as it goes through the liquid hydrogen absorbers. The phases of every cavity are calculated from this trajectory to provide a synchronous phase of 25.5° at the time the reference particle goes through its phase center.

An important point to make is that the r.f. model used to accelerate the reference particle is not the same the beam experiences in normal mode. While the latter could be a Pill Box or an arbitrary field map, the former is a Gaussian distributed field around the phase center of the cavity. The σ of the distribution




```
   // If the simulation contains an r.f. system, it is necessary to tune
   // the cavity phases by shooting a reference particle before runing the
   // beam. Stochastic processes, like multiple scattering, straggling, and
   // beta rays should be turned off during the reference particle run.

 if (hasRF)
   {

     // access the field

     G4FieldManager* fieldMgr
       = G4TransportationManager::GetTransportationManager()
         ->GetFieldManager();

     const G4Field *aField = fieldMgr->GetDetectorField();
     BTGlobalEMField *aEMField = (BTGlobalEMField *) aField;
     aEMField->SetModeRFRefParticle();

     // Turn off Delta Rays, using UI commands implemente by the user in
     // the MuCoolPhysicsListMessenger class. This is done indirectly by
     // setting the production of secondary particles cut in range so high
     // that the primary particle never emmits secondaries.

     UI->ApplyCommand("/range/cutG 1 km");
     UI->ApplyCommand("/range/cutE 1 km");

     // Re-initialize G4 kernel with new particle range cuts.

     runManager->Initialize();

     // Turn off Multiple scattering, using a UI command provided by
     // the GEANT4 library.

     UI->ApplyCommand("/process/inactivate msc");

     // Turn off straggling while setting the rf phases.

     physList->theMuMinusIonisationf()->SetEnlossFluc(false);
     physList->theMuPlusIonisationf()->SetEnlossFluc(false);

     // Run reference particle

     runManager->BeamOn(1);
```

Fig. 17. The reference particle is shot from **main** (**MuCool.cc**) using the **runManager->BeamOn(1)** statement. Multiple scattering and straggling processes are previously turned off.

is 20% of the length of the real simulated cavity or map, and the area under the curve is the total energy provided by that cavity to the particle.

The user should tune the r.f. system using the following procedure:

(i) Calculate and set the value for the energy gain per linac `ReferenceEGainPerLinacSec1` for the reference particle. For example,

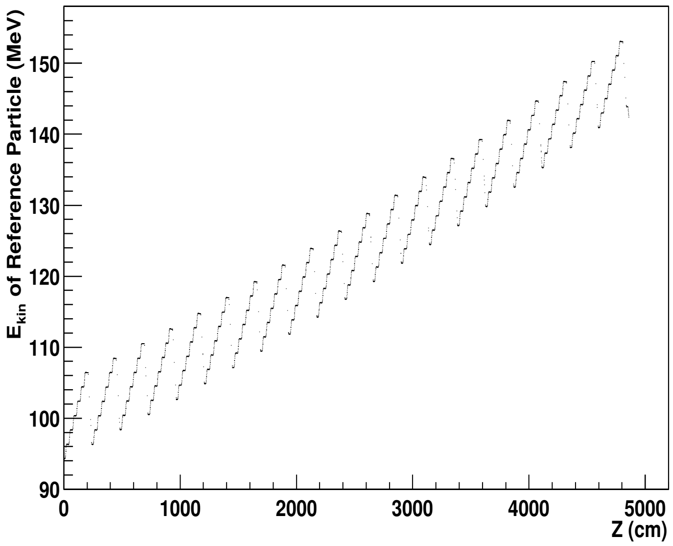


Fig. 18. Reference particle trace through the MuCool channel. The twenty lattice cells with the acceleration and absorber regions are apparent. The section shown is tuned to provide an average net acceleration of 50 MeV to a beam with mean kinematic parameter values coincident with those of the reference particle.

if the cavity peak gradient is V_r =16 MeV/m, its length L =0.32 m, and its synchronous phase ϕ_s =25.5°, the energy gain per cavity will be $V_p \times L \times \sin(\phi_s) \approx$2.2 MeV. A six cavity linac would therefore provide an energy gain of 12.9 MeV.

(ii) Run the simulation for only one particle. This includes the reference particle and the first in the beam input file: `InputBeam.dat`.

(iii) Look at E_{kin} versus z in the `RTTrackRefPart1` NTuple. Go to item (ii), if necessary, and iterate until the desired energy profile is obtained. This is done by adjusting the energy gain per linac to achieve the design net acceleration. For example, in Fig. 18, the design energy gain for the 20 cooling cells is 50 MeV. `ReferenceEGainPerLinacSec1` should be larger than 50 MeV in 20 cells to compensate for the large energy loss in the absorbers.

(iv) Add at the top of `InputBeam.dat` a particle with kinematic parameter values equal to those of the reference particle. Adjust V_p and ϕ_s of the realistic r.f. cavities to reflect any change to `ReferenceEGainPerLinac`.

(v) Run the simulation for only one particle. Make sure `NumberOfTraces` is set to `1`.

(vi) Look at E_{kin} versus z for the `RTTraces0` NTuple, which is the first particle in the `InputBeam.dat` file. This is a reference particle going through the normal run, realistic, r.f. system. Check if the phase tuning worked for this particle by verifying that E_{kin} versus z is quantitatively close to the same plot for `RTTracRefPart1` (the reference particle through the static Gaussian r.f. approximation).

The MuCool example corresponds to a linear accelerator. In the case of closed systems, like ring accelerators, the user should follow the model used in the tools and make the necessary software modifications. One possibility would be to associate a vector of phases and times to each cavity (one vector element related to one turn).

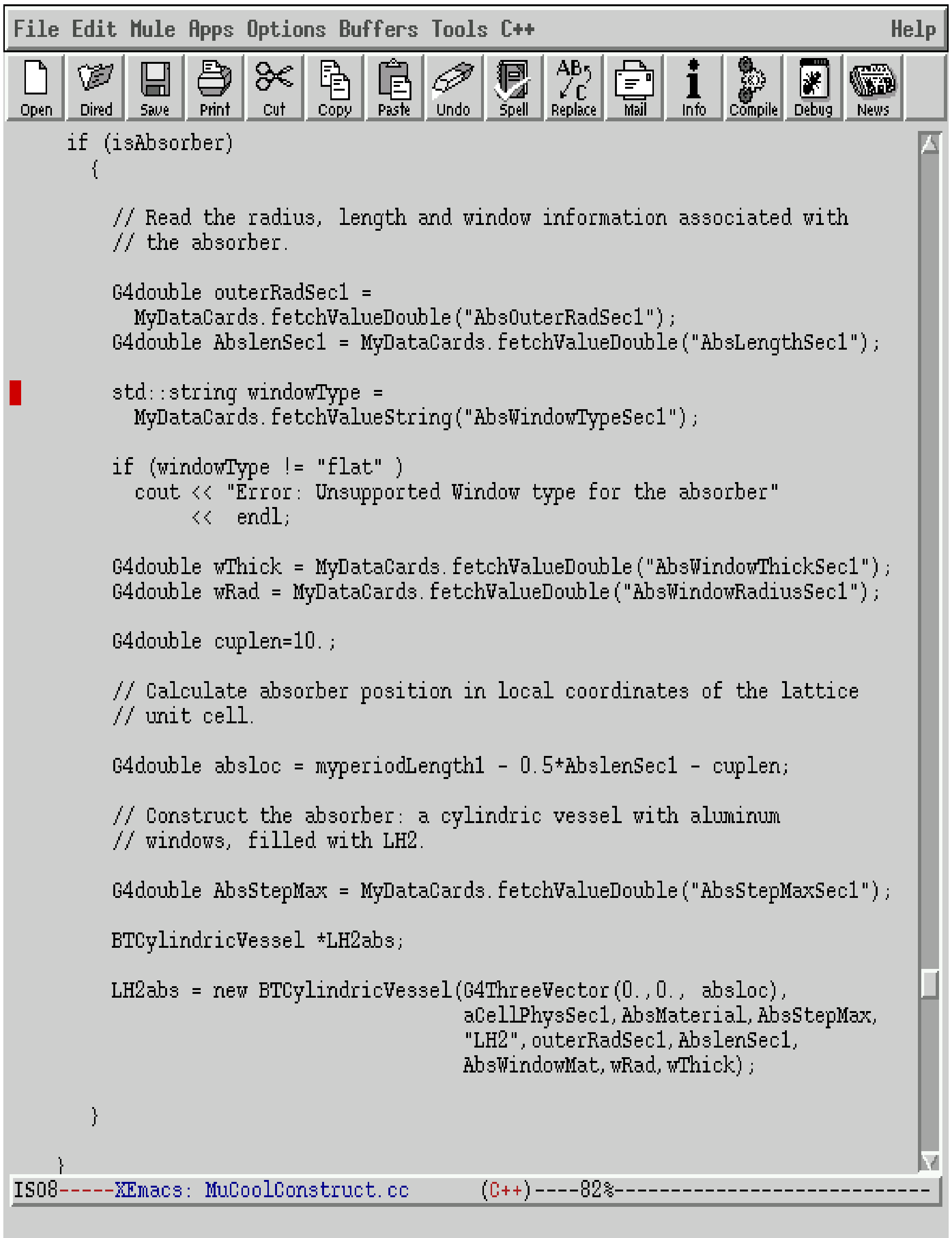
7.4.7 Absorbers

`BT-v1.50` provides a set of classes to simulate blocks of material in the path of the beam. The constructors for different types of absorbers are implemented in `BTAbsorberObjects.cc`. The absorbers are not a common element in accelerators because they typically degrade the beam. There are cases, however, where they can be useful. For example:

- It may be necessary for some experiments to utilize beams of different sizes and qualities. Absorbers may be then used to degrade the beam emittance accordingly.
- Transverse ionization cooling in muon beams may be achieved by reducing the beam total momentum by energy loss through an absorber material. After re-acceleration along the beam direction, the net result will be a reduction in the transverse emittance.
- Emittance exchange may be achieved with wedge or lense absorbers placed in a beam with transverse-longitudinal correlations. For example, if the p_z of a particle in a beam is a function of the distance r to the system center, a wedge or a lense can selectively reduce the speed of faster particles with respect to slower ones.

The MuCool example simulates the cylindric vessels used in most of the cooling channels studied for neutrino factory applications. The `Construct` method in `MuCoolConstruct.cc` calls the `BTCylindricVessel` constructor which builds a cylindric aluminum vessel with aluminum end cup thin windows. The vessel is filled with liquid hydrogen. In Geant4 language, this is a set of tubes and associated logical volumes. It also places the vessel inside the `aCellPhysSec1` lattice cell. As illustrated in Fig. 19, `BTCylindricVessel` takes the absorber location in local coordinates of the lattice unit cell, a pointer to that cell, the absorber material, the maximum step length in the absorber, the name of the object, the outer radius, the length, the absorber window material, the window radius, and its thickness. Both the cylinder and end cup walls are 3 cm thick, and the end cups inner radii adjusts itself automatically depending on the window radius. Realistic vessel windows are typically parabolic in shape to withstand presure. The `BT-v1.50` options, however, include only flat windows.

The grey cylinder in Fig. 1 is a schematic representation of a liquid hydrogen vessel with aluminum walls and windows, visualized using the `MuCool` package.




```
    if (isAbsorber)
      {

        // Read the radius, length and window information associated with
        // the absorber.

        G4double outerRadSec1 =
          MyDataCards.fetchValueDouble("AbsOuterRadSec1");
        G4double AbslenSec1 = MyDataCards.fetchValueDouble("AbsLengthSec1");

        std::string windowType =
          MyDataCards.fetchValueString("AbsWindowTypeSec1");

        if (windowType != "flat" )
          cout << "Error: Unsupported Window type for the absorber"
               <<  endl;

        G4double wThick = MyDataCards.fetchValueDouble("AbsWindowThickSec1");
        G4double wRad = MyDataCards.fetchValueDouble("AbsWindowRadiusSec1");

        G4double cuplen=10.;

        // Calculate absorber position in local coordinates of the lattice
        // unit cell.

        G4double absloc = myperiodLength1 - 0.5*AbslenSec1 - cuplen;

        // Construct the absorber: a cylindric vessel with aluminum
        // windows, filled with LH2.

        G4double AbsStepMax = MyDataCards.fetchValueDouble("AbsStepMaxSec1");

        BTCylindricVessel *LH2abs;

        LH2abs = new BTCylindricVessel(G4ThreeVector(0.,0., absloc),
                                       aCellPhysSec1,AbsMaterial,AbsStepMax,
                                       "LH2",outerRadSec1,AbslenSec1,
                                       AbsWindowMat,wRad,wThick);

      }

  }
```

Fig. 19. Construction of a cylindric vessel with liquid hydrogen absorber.

The vessel parameters are read from the `MuCool.in` absorber block:

```
#
# Absorber information
#
AbsorberType Liquid Hydrogen
# absorber material
AbsWindowMaterial Aluminium
# absorber window materia
```

```
AbsLengthSec1 300.
# absorber length (in mm)
AbsOuterRadSec1 200.
# radial length (in mm)
AbsWindowTypeSec1 flat
# type of window
AbsWindowRadiusSec1 160.
# window radius (in mm)
AbsWindowThickSec1 0.360
# window thickness (in mm)
AbsStepMaxSec1 100.
#
```

Figure 20 shows the location of the absorber vessel in a MuCool lattice unit cell. The wireframe mode allows to visualize the linac and the absorber through a transparent solenoid.

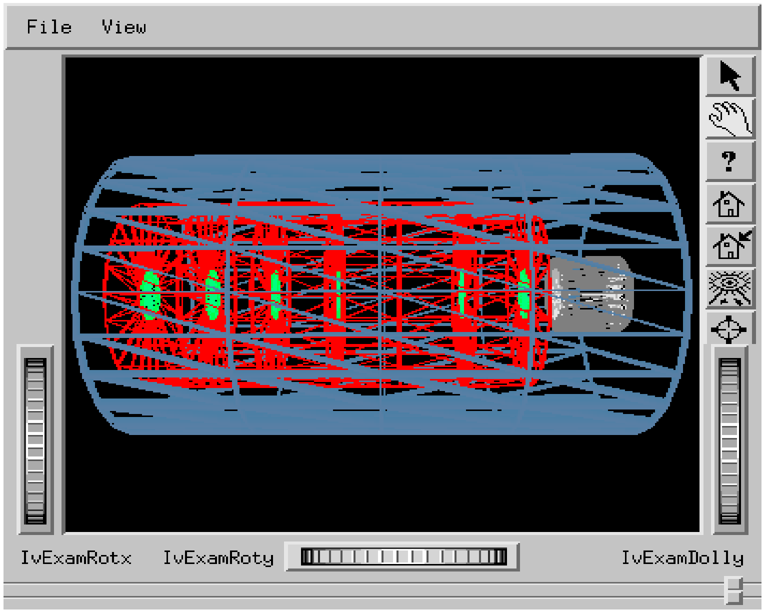


Fig. 20. side view of a lattice unit cell. The wireframe mode allows to visualize the linac and the absorber through a transparent solenoid.

`BT-v1.50` also provides two constructors to simulate absorber lenses: `BTParabolicLense` and `BTCylindricLense`. The first one is a parabolic object with uniform density, and the second is a cylinder object with the density decreasing parabolically as a function of radius. From the point of view of the physics effect on the beam, both objects are almost equivalent. The `BTParabolicLense` constructor takes the object location with respect to the local coordinates of the lattice unit cell, a pointer to the lattice unit cell volume, a pointer to the lense material, the maximum step length allowed, a name, the length (maximum) at r=0, the radius, and the number of cylindric slices which make up the lense:

```
BTParabolicLense::BTParabolicLense( G4ThreeVector location,
G4VPhysicalVolume* pMother, G4Material *material, G4double stepmax,
G4String paraname, G4double parablength, G4double parabradmax,
G4int parabnumslice )
```

The lense is built as a set of short cylinders. The radius is maximum for the central cylinder and reduces symmetrically following a parabolic equation for the others in both sides.

The `BTCylindricLense` constructor takes essencially the same arguments as the parabolic lense, except that the number of slices is replaced by the number of rings. The object is built from concentric cylinder rings of the same length, different radius, and different densities to mimic a real lense.

```
BTCylindricLense::BTCylindricLense( G4ThreeVector location,
G4VPhysicalVolume* pMother, G4Material *material, G4double stepmax,
G4String cyliname, G4double cylilength, G4double cyliradmax,
G4int cylinumrings )
```

Figure 21 shows a set of six parabolic lenses placed in a field flip region at the end of the Double Flip first section where the MuCool example ends [9]. The lenses are placed to mitigate the effect of the decrease in $\langle p_z \rangle$ at large radii in the flip region, using an emittance exchange mechanism.

Wedge absorbers are also useful in some cases. Although `BT-v1.50` does not provide a constructor to build a wedge object, it can be easily constructed using the Geant4 trapezoid shape `G4Trap`. For more details, see the Geant4 user's guide [3]. Fig. 22 and Fig. 23 show a muon track, represented by a red line, propagated through a solenoidal plus a rotating dipole field. The yellow and green objects are the wedge absorbers, and the blue discs are representations of a thin cavity r.f. field map. More information about this "Helical Cooling Channel" is available in Ref. [10].

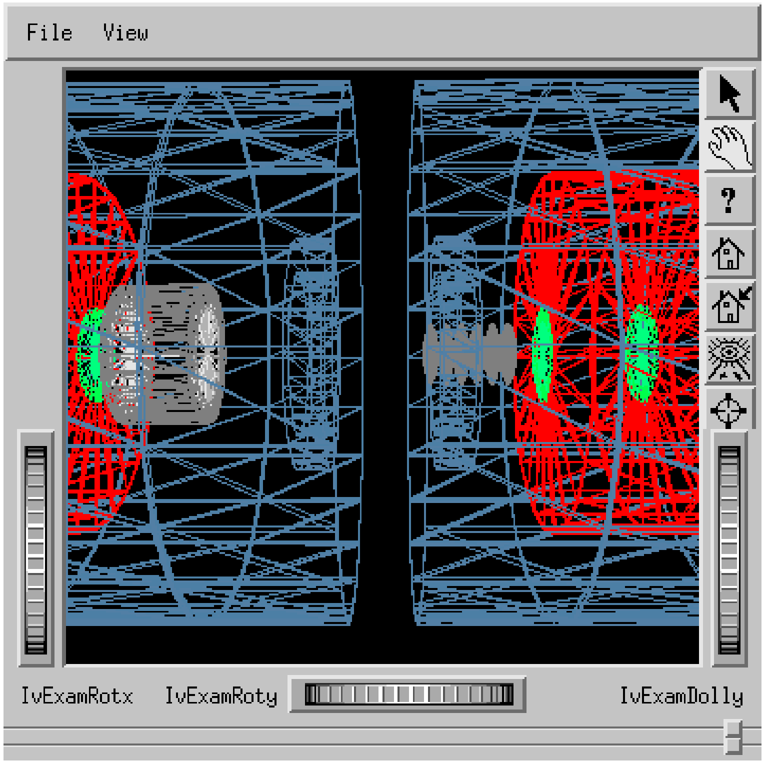


Fig. 21. Side view of a field flip region of the Double Flip channel in wireframe mode. The flip section inner coils are shown in the center of the figure following a regular lattice cell. The six parabolic lenses, in grey, preceed a new cell.

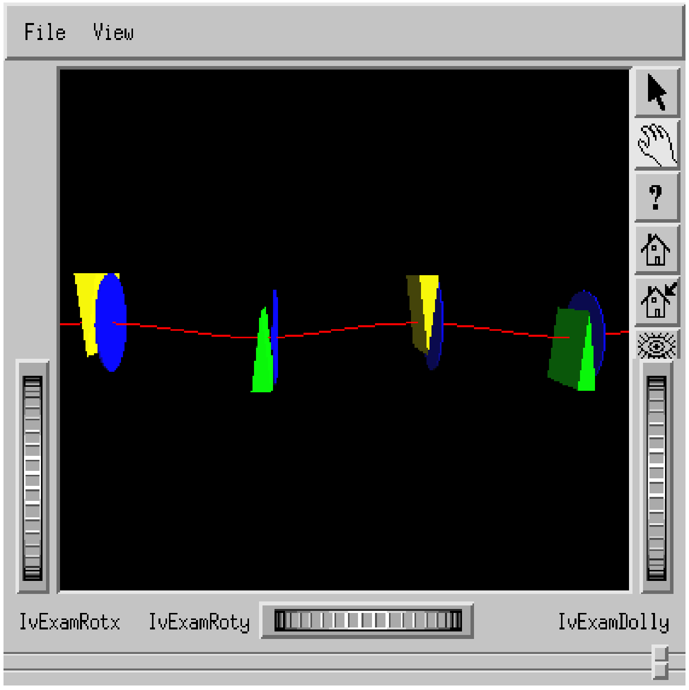


Fig. 22. A muon track, represented by a red line, propagating through a solenoidal plus a rotating dipole field. The yellow and green objects are the wedge absorbers, and the blue discs are representations of a thin cavity r.f. field map.

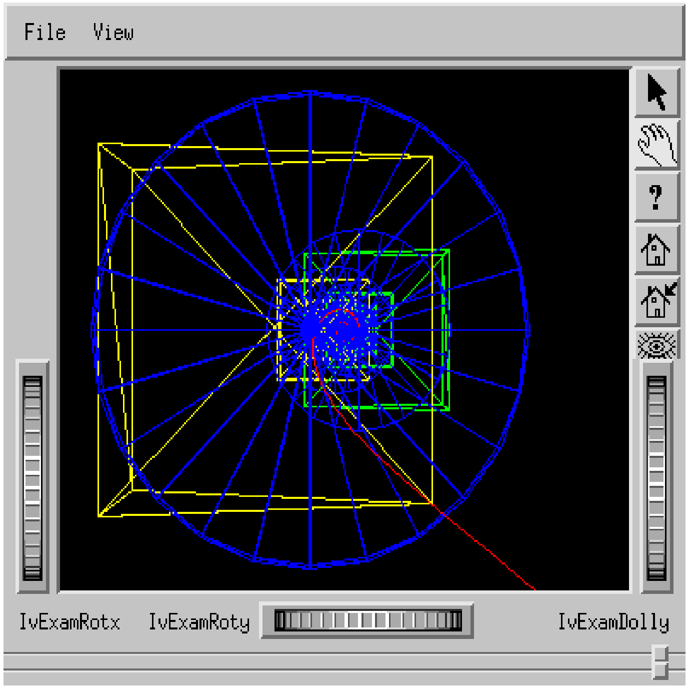


Fig. 23. Wireframe front view of the same system described in Fig. 22.

The sensitive detectors are empty software volumes which only provide boundaries, that is a volume transition, to force the stepper to make a pause and allow the user to execute some actions through `UserSteppingAction`. Sensitive detectors may also be used to force a change in the maximum step size to help the stepper not to miss an abrupt and short change in the electromagnetic field. In MuCool, we use them to produce the output NTuples.

The "data" output file, `MuCool.root`, includes a set of histograms and NTuples. The NTuples contain kinematic information of the beam at different locations along the channel (z-planes). The NTuple associated with the initial beam is filled in `MuCoolPrimaryGeneratorAction::GeneratePrimaries`, every time the method is invoked before each particle is processed. The other NTuples, however, must be filled along the channel at z-locations selected by the user. This is done in the `MuCoolSteppingAction::UserSteppingAction` method implemented in `MuCoolSteppingAction.cc`. This method is invoked at the end of each step in the integration of the equation of motion, along the particle trajectory. Usage of the stepping action user class will be discussed in Section 7.6. The issue is discussed here because the way to define the z-planes is by placing "sensitive detectors" which force a step in that location for the `UserSteppingAction` method to fill the NTuple.

In the MuCool example, the sensitive detectors are short cylinders, 1 mm long and 500 mm in radius, which will be intersected by all particles in the beam. The user provides the geometry arguments through `MuCool.in`:

```
#
# Sensitive Detector information
#
SensDetRadius 500.
# Radius of sensitive detectors big enough to intersect the beam,
# small enough to avoid intersecting an accelerator element.
SensDetLength 1.
# z-length of sensitive detectors
```

and calls the `MuCoolConstruct::SetDetectors` at the end of the `MuCoolConstruct::Construct` method:

```
G4double radSensDet, zlengthSensDet;
radSensDet = MyDataCards.fetchValueDouble("SensDetRadius");
zlengthSensDet = MyDataCards.fetchValueDouble("SensDetLength");

// set sensitive detectors to fill
// NTuples at different channel
```

```
// locations (one per cell)
SetDetectors(radSensDet,zlengthSensDet);

cout << "Sensitive detectors are now in place" << endl;
```

Figs. 24 and 25 show an example on how to implement the `SetDetectors` method. In a first step, we construct a tube and its associated logical volume. In a second step, we create a loop to place each of the sensitive detectors, one per lattice cell. Note that we use two data member vectors of the `MuCoolConstruct` user class, which were previously filled in `MuCoolConstruct::Construct`, inmediatelly before constructing the `BTSolenoidLogicVol` object:

```
// Store in vector data member myCellPhys a pointer to the lattice
unit cell physical volume. Store in vector data member
// myAllLengthsPeriods the location of the Sensitive Detectors
// in local coordinates, slightly off from the left edge
(beginning) of the lattice unit cell.

myCellPhys.push_back(aCellPhysSec1);
myAllLengthPeriods.push_back(-myperiodLength1+extraZ);
```

The `myCellPhys` vector contains the pointers to the physical volumes associated with the lattice unit cells, and `myAllLengthPeriods` the positions of the sensitive detectors in local coordinates of the lattice cell. The number of cells is extracted from the size of `myCellPhys`, and the local position `ll` from the `myAllLengthPeriods` vector.

The placement of the `aSensDetPhys` sensitive detectors, illustrated in Fig. 25, is followed by the construction of a vector `Dets` (global) of `BTSensDetGrid` objects, which is accessed in `UserSteppingAction`. The constructor of the grid object takes the arguments: pointer to the sensitive detector mother volume, pointer to the physical volume the particle comes from when it reaches the sensitive detector, pointer to the sensitive detector physical volume, grid object identification number, and name.

Note that for closed orbit accelerators there should be one detector per cell multiplied by the number of turns of the beam around the ring. One option would be to modify the `BTSensDetGrid` to contain turn information.

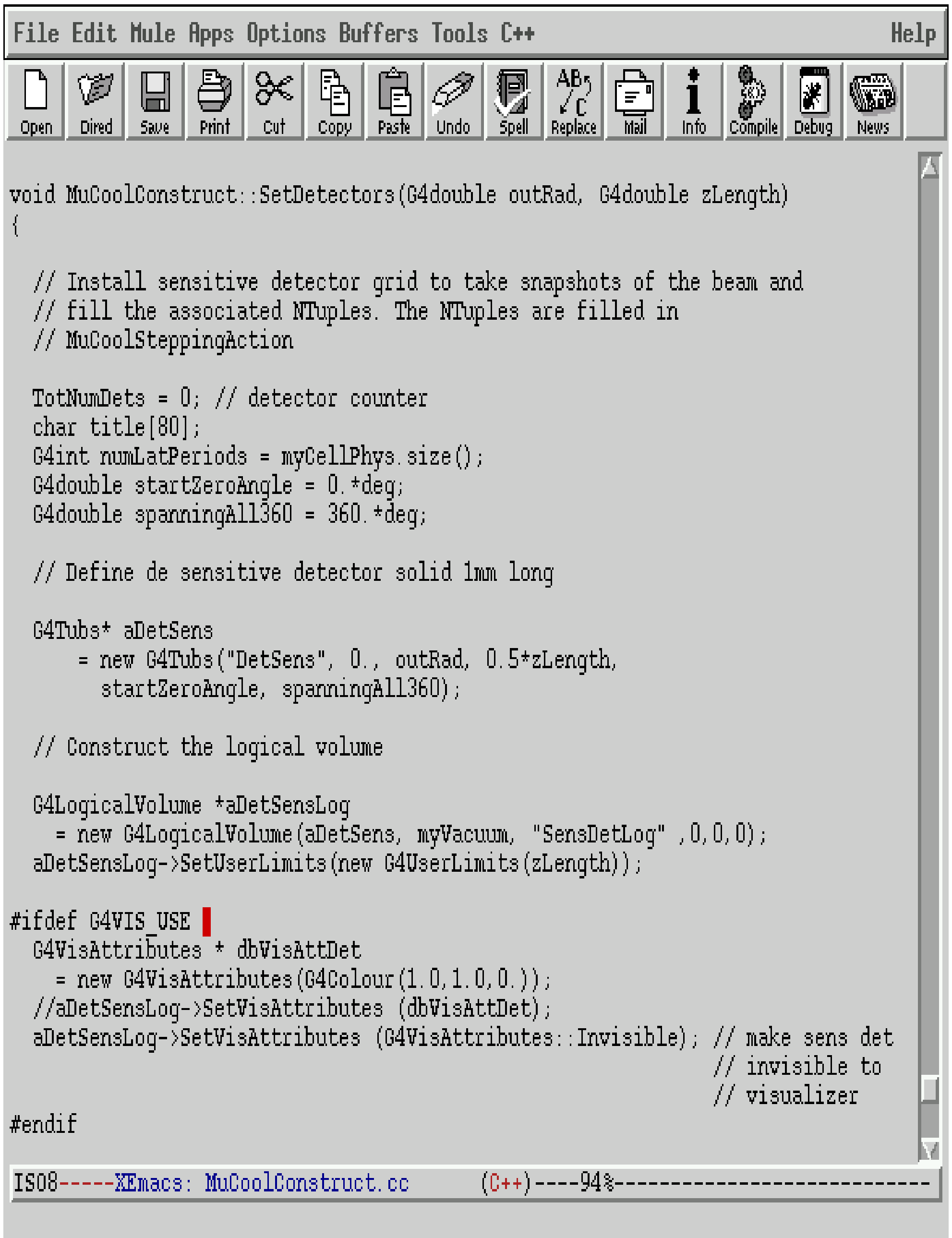
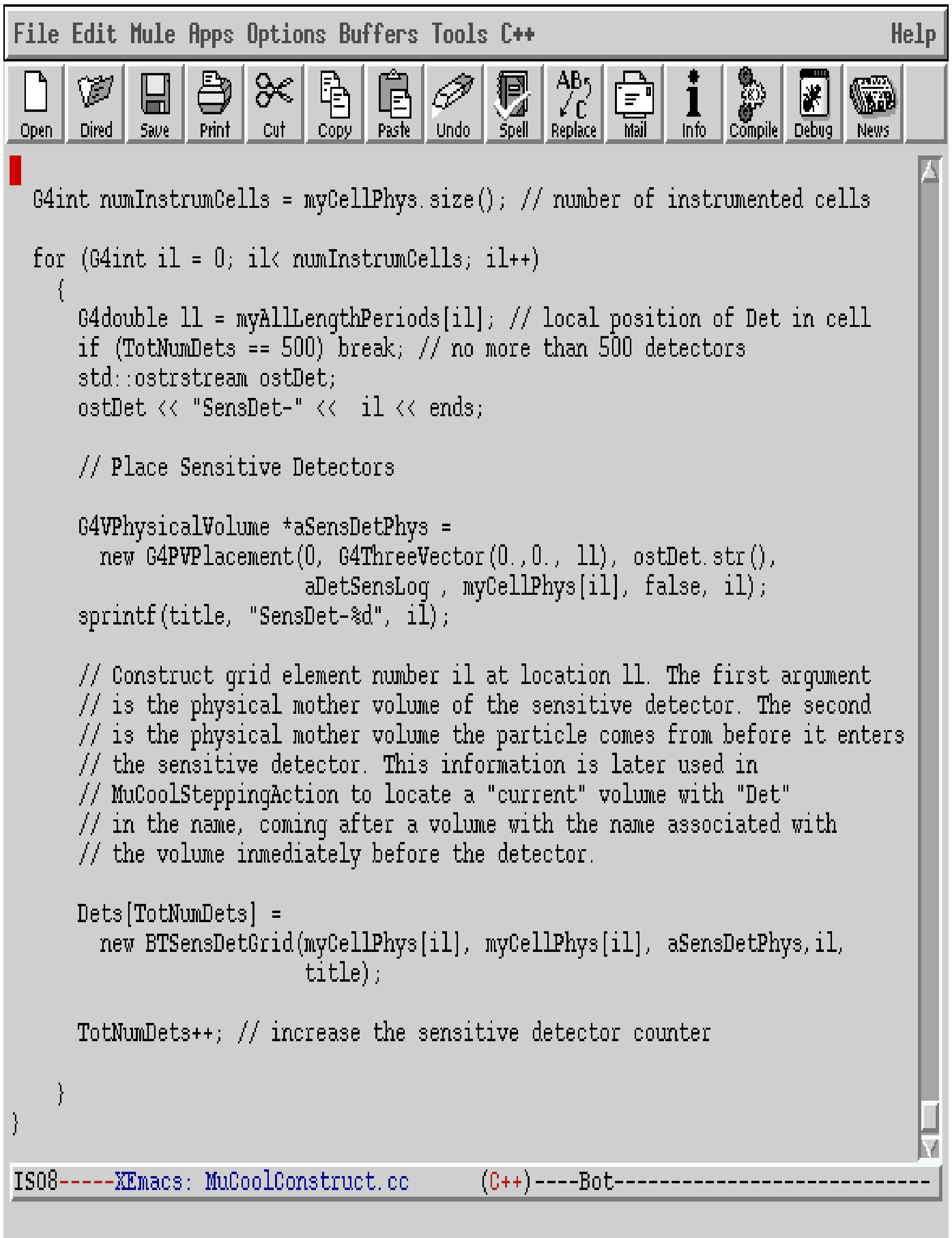


Fig. 24. `SetDetectors` method in `MuCoolConstruct` (first part).



```
File Edit Mule Apps Options Buffers Tools C++                    Help

  G4int numInstrumCells = myCellPhys.size(); // number of instrumented cells

  for (G4int il = 0; il< numInstrumCells; il++)
    {
      G4double ll = myAllLengthPeriods[il]; // local position of Det in cell
      if (TotNumDets == 500) break; // no more than 500 detectors
      std::ostrstream ostDet;
      ostDet << "SensDet-" <<  il << ends;

      // Place Sensitive Detectors

      G4VPhysicalVolume *aSensDetPhys =
        new G4PVPlacement(0, G4ThreeVector(0.,0., ll), ostDet.str(),
                          aDetSensLog , myCellPhys[il], false, il);
      sprintf(title, "SensDet-%d", il);

      // Construct grid element number il at location ll. The first argument
      // is the physical mother volume of the sensitive detector. The second
      // is the physical mother volume the particle comes from before it enters
      // the sensitive detector. This information is later used in
      // MuCoolSteppingAction to locate a "current" volume with "Det"
      // in the name, coming after a volume with the name associated with
      // the volume immediately before the detector.

      Dets[TotNumDets] =
        new BTSensDetGrid(myCellPhys[il], myCellPhys[il], aSensDetPhys,il,
                          title);

      TotNumDets++; // increase the sensitive detector counter

    }
}
ISO8-----XEmacs: MuCoolConstruct.cc      (C++)----Bot----------------------------
```

Fig. 25. `SetDetectors` method in `MuCoolConstruct` (second part).

The `MuCoolSteppingAction` user class allows to perform actions at the end of each step of the integration of the equation of motion. This actions may include killing a particle under certain conditions, retrieving information for diagnostics, etc. Remember that different stepper packages are available from the Geant4 library. The choice is made at the beginning of the `MuCoolConstruct::Construct` method. See the Geant4 documentation for more information about steppers [3].

Any information provided through `MuCool.in` should be read from the `MuCoolSteppingAction` constructor (called only once). The `MuCoolSteppingAction::UserSteppingAction` is called at the end of every step to perform the user actions. For example, if I want to kill a particle when it is too slow or goes too far from the center of the beam, I will read the thresholds in the `MuCoolSteppingAction` constructor:

```
// Read space boundaries for particle

mymaxRadCut = MyDataCards.fetchValueDouble("StopAtRadius");
mymaxZCut = MyDataCards.fetchValueDouble("StopAtZ");
myKinEneCut = MyDataCards.fetchValueDouble("KineticEnergyCut");
```

and execute the action in `MuCoolSteppingAction::UserSteppingAction`, following the syntax shown in Fig. 26. Figure 27 shows how to retrieve information such as the global field and pointers to the physical volumes inmediatelly before (pre) and after (post) the current step position. Figure 28 lists the necessary code to perform the r.f. phase tuning. If we are processing the reference particle, and Geant4 has made a step in between a volume with an "RF" string in its name (pre) and a volume with a "DetMiddle" in its name, then the particle is at the phase center of a cavity and the phase delay must be calculated to adjust the total phase to the selected synchronous phase. Note that a sensitive detector with name "DetMiddle" is hard-coded in the phase center of each cavity. These sensitive detectors are placed automatically by the Beam Tools when constructing a linac. The user must, however, make sure that there is an "RF" string in the name of the cavity logical volume; otherwise, he/she must change the `if` statement to reflect the name of the cavity volume. The method `SetPhaseDelayAtZ` of `BTGlobalEMField` retrieves the global time associated with the reference particle as it goes through the phase center of each cavity. It also stores these phase delays in a `BTLinacCellPhaseInfo` object which is retrieved at the time of calculating the global field for a normal run particle at a given position.

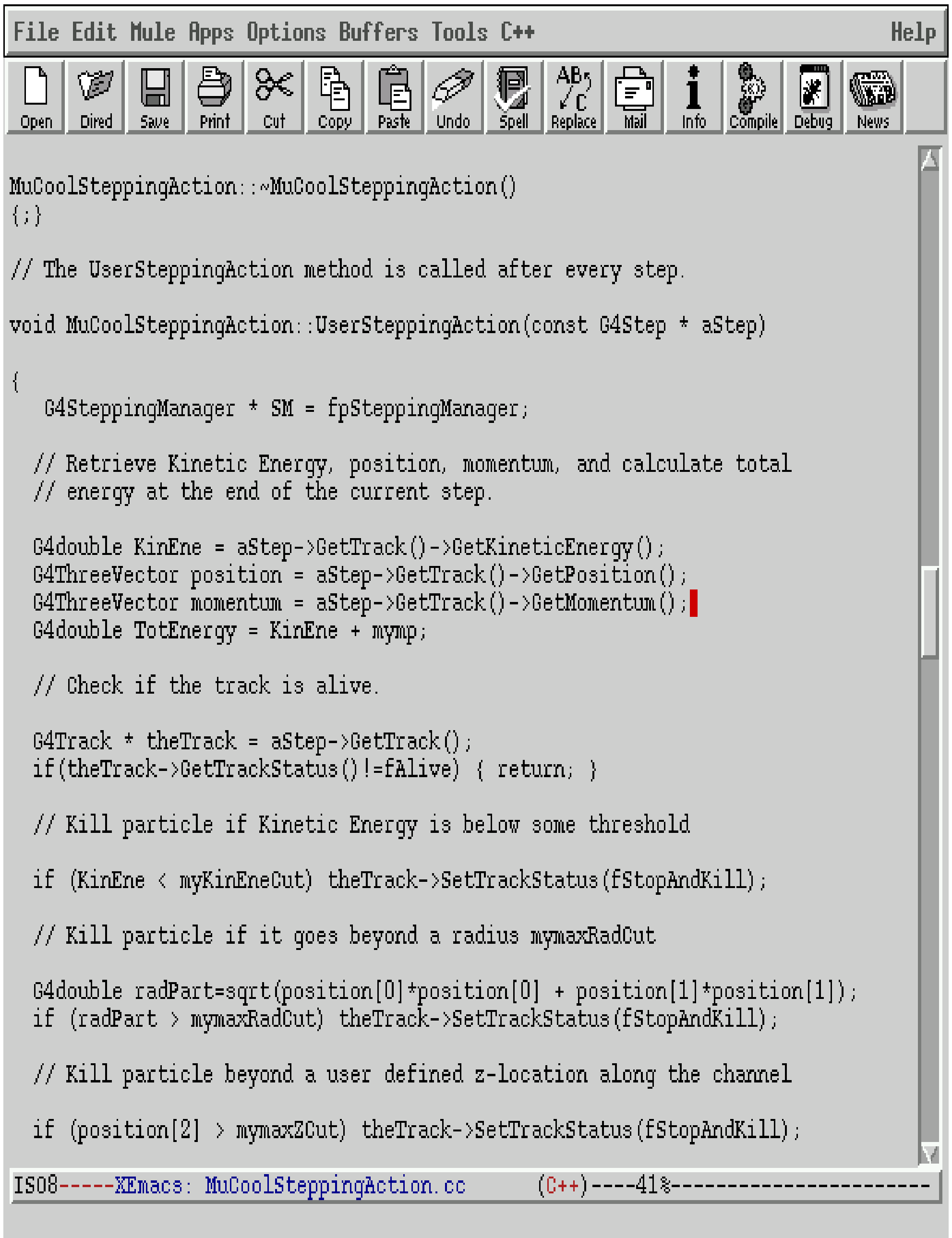


Fig. 26. The user can retrieve particle information and use it to perform actions at the end of each integration step in `MuCoolSteppingAction::UserSteppingAction`. For example, kill a particle under certain conditions.

Figure 29 shows the process of filling the z-plane NTuples during a normal run. For this, we use the global `BTSensDetGrid` object `Dets` created in `MuCoolConstruct::Construct`. A given detector element `Dets[jV]` is flagged when the particle is at a point in space in between a sensitive detector volume (`thePostPV`) and a different preceeding volume (`thePrePV`), which is provided by the user to the `BTSensDetGrid` in `MuCoolConstruct::Construct`. Then, the NTuple is filled with the particle kinematic information available from the

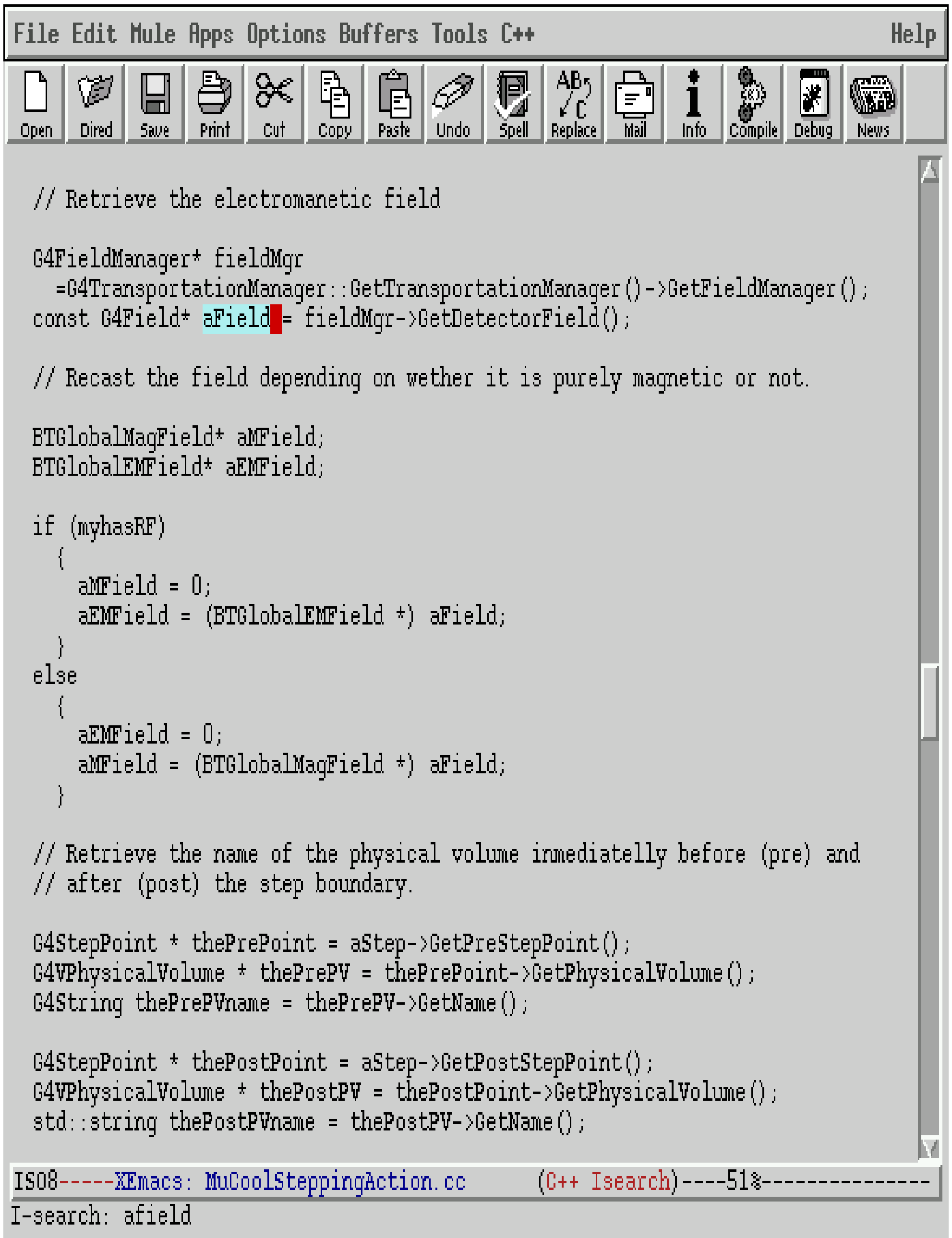
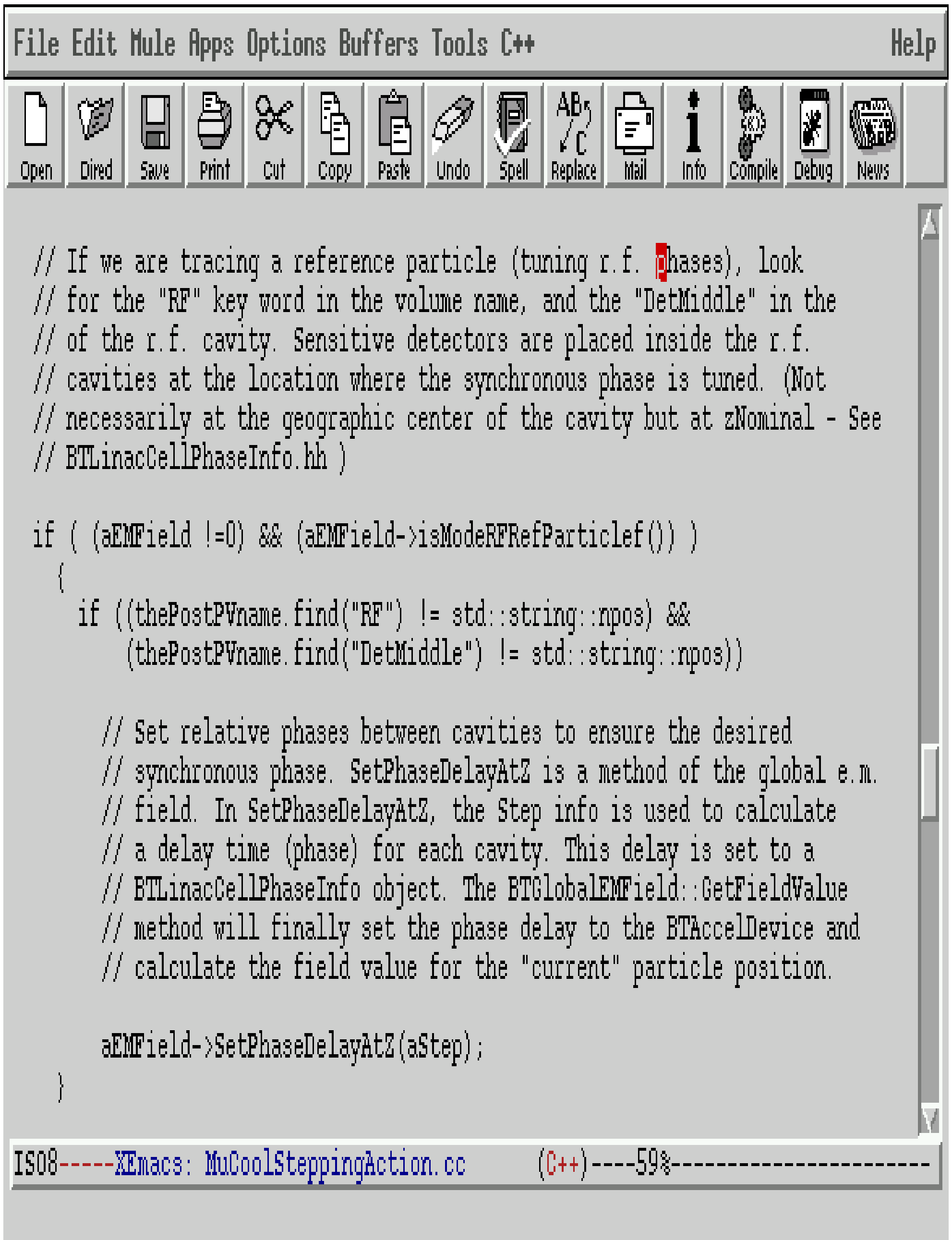


Fig. 27. The user can also retrieve information like a pointer to the global field and the name of the volumes before and after the current step location.

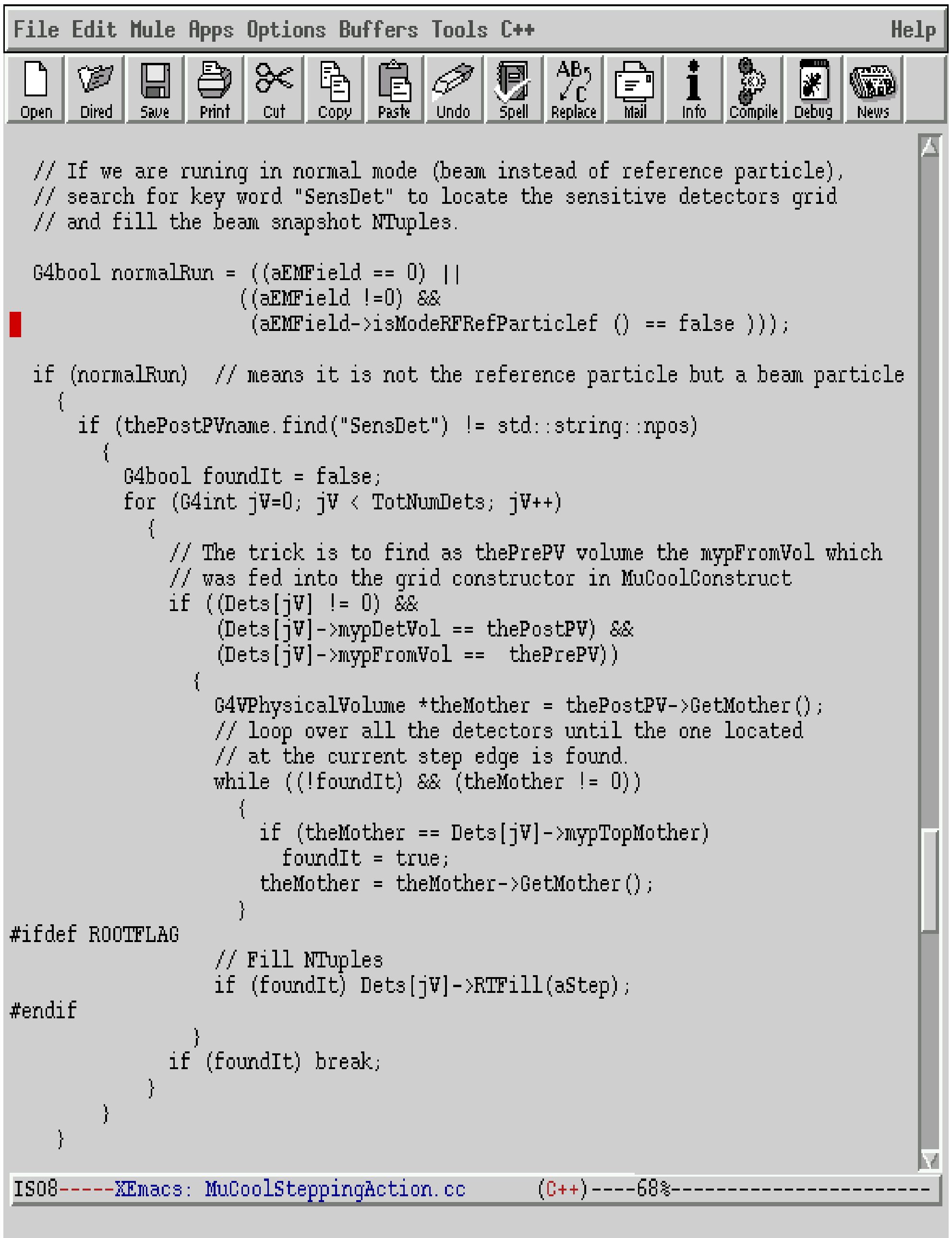
current step object `aStep`.

The subsequent blocks in `UserSteppingAction` fill the trace NTuples, both for a normal run and the reference particle. While in the case of the z-plane NTuples there is one entry per particle at a given z location, the trace NTuples are related to one single particle and are filled at the end of every step. Note that all NTuples are data members of `MuCoolSteppingAction`.




```
// If we are tracing a reference particle (tuning r.f. phases), look
// for the "RF" key word in the volume name, and the "DetMiddle" in the
// of the r.f. cavity. Sensitive detectors are placed inside the r.f.
// cavities at the location where the synchronous phase is tuned. (Not
// necessarily at the geographic center of the cavity but at zNominal - See
// BTLinacCellPhaseInfo.hh )

if ( (aEMField !=0) && (aEMField->isModeRFRefParticlef()) )
  {
    if ((thePostPVname.find("RF") != std::string::npos) &&
        (thePostPVname.find("DetMiddle") != std::string::npos))

      // Set relative phases between cavities to ensure the desired
      // synchronous phase. SetPhaseDelayAtZ is a method of the global e.m.
      // field. In SetPhaseDelayAtZ, the Step info is used to calculate
      // a delay time (phase) for each cavity. This delay is set to a
      // BTLinacCellPhaseInfo object. The BTGlobalEMField::GetFieldValue
      // method will finally set the phase delay to the BTAccelDevice and
      // calculate the field value for the "current" particle position.

      aEMField->SetPhaseDelayAtZ(aStep);
  }
```

Fig. 28. This block of code is necessary to perform the r.f. phase tuning.



```
// If we are runing in normal mode (beam instead of reference particle),
// search for key word "SensDet" to locate the sensitive detectors grid
// and fill the beam snapshot NTuples.

G4bool normalRun = ((aEMField == 0) ||
                    ((aEMField !=0) &&
                     (aEMField->isModeRFRefParticlef () == false )));

if (normalRun)  // means it is not the reference particle but a beam particle
  {
    if (thePostPVname.find("SensDet") != std::string::npos)
      {
        G4bool foundIt = false;
        for (G4int jV=0; jV < TotNumDets; jV++)
          {
            // The trick is to find as thePrePV volume the mypFromVol which
            // was fed into the grid constructor in MuCoolConstruct
            if ((Dets[jV] != 0) &&
                (Dets[jV]->mypDetVol == thePostPV) &&
                (Dets[jV]->mypFromVol ==  thePrePV))
              {
                G4VPhysicalVolume *theMother = thePostPV->GetMother();
                // loop over all the detectors until the one located
                // at the current step edge is found.
                while ((!foundIt) && (theMother != 0))
                  {
                    if (theMother == Dets[jV]->mypTopMother)
                      foundIt = true;
                    theMother = theMother->GetMother();
                  }
#ifdef ROOTFLAG
                // Fill NTuples
                if (foundIt) Dets[jV]->RTFill(aStep);
#endif
              }
            if (foundIt) break;
          }
      }
  }
```

Fig. 29. Filling of the z-plane NTuples during a normal run.

Geant4 allows the user to select among a variety of physics processes which may occur during the interaction of the incident particles with the material of the simulated apparatus. There are electromagnetic, hadronic and other interactions available like: "electromagnetic", "hadronic", "transportation", "decay", "optical", "photolepton_hadron", "parameterisation". In the MuCool example, the information on physics processes is contained in the `MuCoolPhysicsList *physList` object. The contructor of the `MuCoolPhysicsList` class is implemented in `MuCoolPhysicsList.cc`, together with other methods to create the different types of particles and processes. More information on physics processes can be found in the Geant4 user and physics reference guides [3]).

There is also a method, `SetCuts()` of `MuCoolPhysicsList`, which must be implemented in `MuCoolPhysicsList.cc` (see Fig 30). It allows to select what type of particles to apply production thresholds on. Each particle type may have different cut values which are assigned to data member variables of `MuCoolPhysicsList` by the `SetGammaCut`, `SetElectronCut`, `SetProtonCut` methods. The cuts are applied in a Geant4 internal call to `SetCuts()`. The implementation of these methods is illustrated in Fig. 31.

7.7.1 *The* `MuCoolPhysicsListMessenger` *Class*

Geant4 provides a tool to pass run parameters interactivelly through predefined commands. We have seen two examples in `main` (`MuCool.cc`). In one case, `/process/inactivate msc` is a built-in command provided by the Geant4 library to inactivate the multiple scattering process:

```
// Turn off Multiple scattering, using a UI command provided by
// the GEANT4 library.

UI->ApplyCommand("/process/inactivate msc");
```

In the second case, `/range/cut..` are user defined commands to change the particle production thresholds:

```
// Set particle production thresholds explicitly to default values.
// Use the user interface commands defined BY THE USER in the
// MuCoolPhysicsListMessenger class.

UI->ApplyCommand("/range/cutG 2 mm");
UI->ApplyCommand("/range/cutE 2 mm");
```

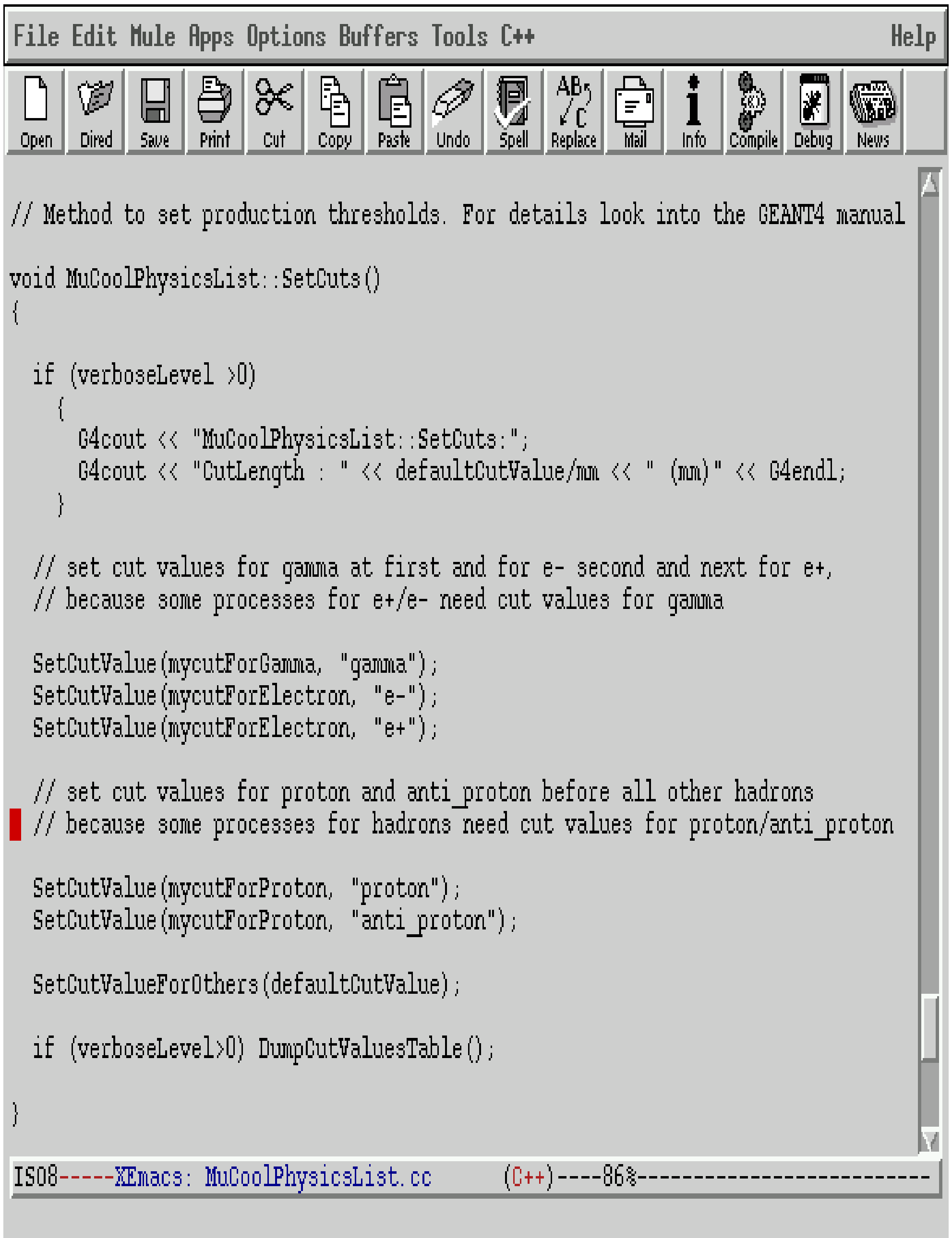


Fig. 30. The `SetCuts()` method of `MuCoolPhysicsList`.

Physics related commands may be defined in the `MuCoolPhysicsMessenger` constructor, as shown in Fig. 32. The new values are set by the `SetNewValue` method of `MuCoolPhysicsMessenger` which is invoked internally by Geant4. Figure 33 shows the implementation of `SetNewValue`.

There is a variety of built-in commands and messenger user classes available from the Geant4 library. More information on user interface capabilities is

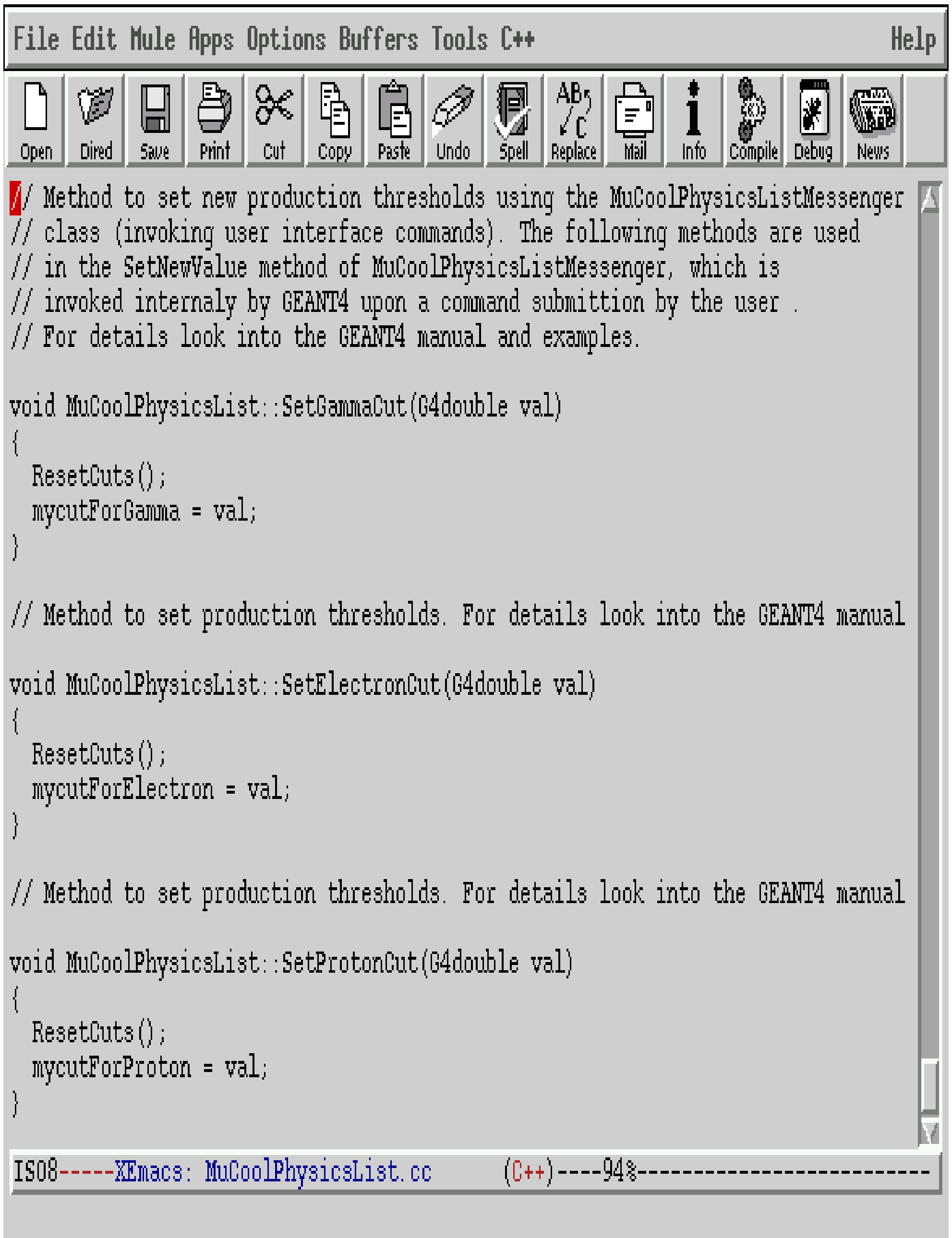


Fig. 31. Each particle type may have different cut values which are assigned to data member variables of `MuCoolPhysicsList` by the `SetGammaCut`, `SetElectronCut`, `SetProtonCut` methods.

available in the Geant4 user's guide [3].


```
// Constructor of the messenger objects

MuCoolPhysicsListMessenger::MuCoolPhysicsListMessenger(MuCoolPhysicsList* EvAct
)
:physList(EvAct)

{
  // Construct user interface commands (UIcmd) to set run parameters
  // interactivelly. Secondary particle production thresholds by range, by
  // energy. See header file and GEANT4 user's manual.

  cutGCmd = new G4UIcmdWithADoubleAndUnit("/range/cutG",this);
  cutGCmd->SetGuidance("Set cut values by RANGE for Gamma.");
  cutGCmd->SetParameterName("range",false);
  cutGCmd->SetRange("range>0.");
  cutGCmd->SetUnitCategory("Length");
  cutGCmd->AvailableForStates(PreInit,Idle);

  cutECmd = new G4UIcmdWithADoubleAndUnit("/range/cutE",this);
  cutECmd->SetGuidance("Set cut values by RANGE for e- e+.");
  cutECmd->SetParameterName("range",false);
  cutECmd->SetRange("range>0.");
  cutECmd->SetUnitCategory("Length");
  cutECmd->AvailableForStates(PreInit,Idle);

  cutPCmd = new G4UIcmdWithADoubleAndUnit("/range/cutP",this);
  cutPCmd->SetGuidance("Set cut values by RANGE for proton and others.");
  cutPCmd->SetParameterName("range",false);
  cutPCmd->SetRange("range>0.");
  cutPCmd->SetUnitCategory("Length");
  cutPCmd->AvailableForStates(PreInit,Idle);

  eCmd = new G4UIcmdWithADoubleAndUnit("/range/cutEnergy",this);
  eCmd->SetGuidance("Set cut values by ENERGY for charged particles.");
  eCmd->SetParameterName("energy",false);
  eCmd->SetRange("energy>0.");
  eCmd->SetUnitCategory("Energy");
  eCmd->AvailableForStates(Idle);

}
```

Fig. 32. The `MuCoolPhysicsListMessenger` constructor.

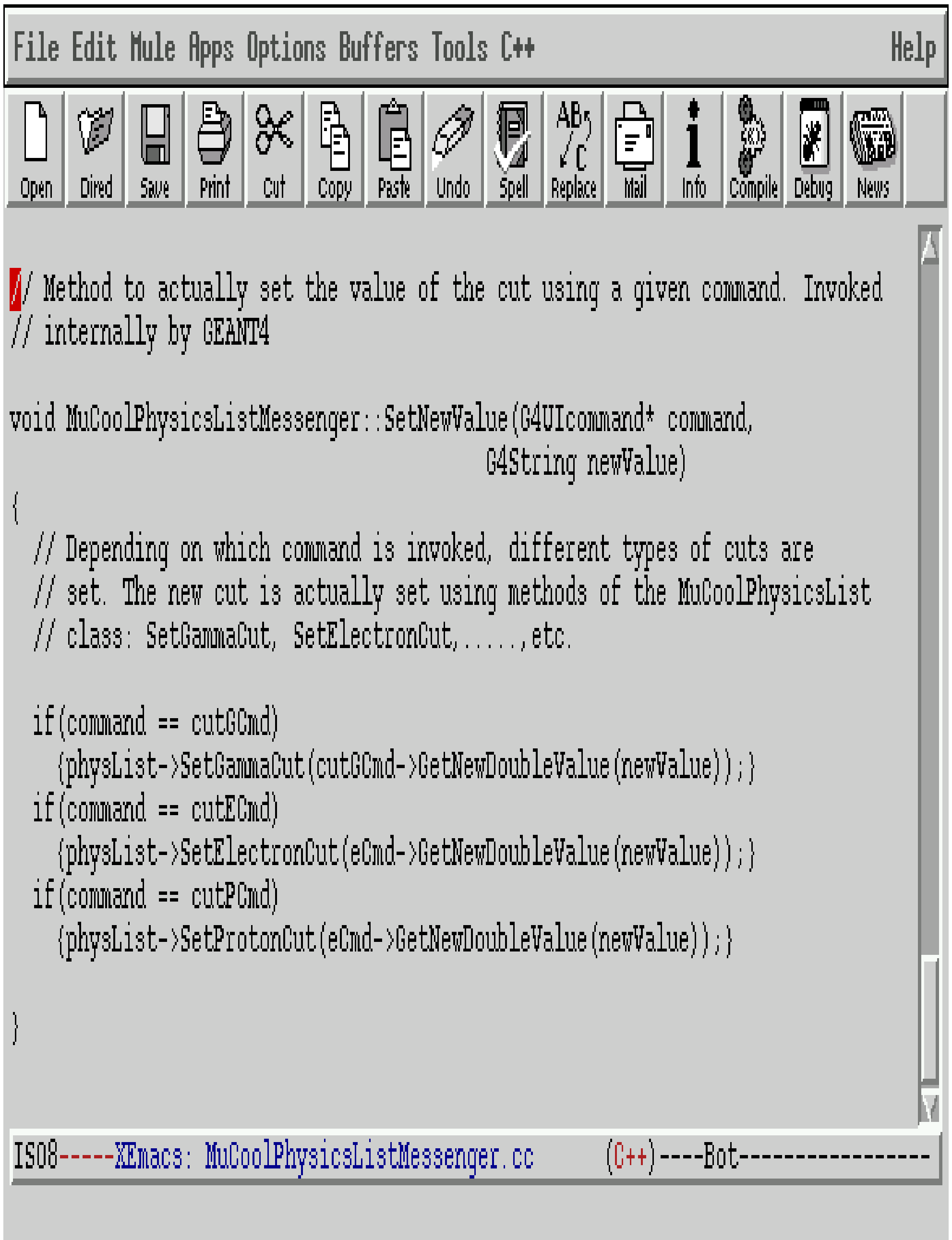


Fig. 33. `SetNewValue` method of `MuCoolPhysicsMessenger`.

7.8 Tracking Actions

The tracking action user class allows to perform actions on tracks. In the MuCool example, we are interested in tracing only the muons. Secondary particles from muon decay or product of the interaction of muons with matter are therefore discarded. This is done in the `PreUserTrackingAction` method of the `MuCoolTrackingAction` class:

```
if(aTrack->GetParentID()==0) // particle ID=0 means primary particle
{
fpTrackingManager->SetStoreTrajectory(true);
}
else
{
fpTrackingManager->SetStoreTrajectory(false);
}
```

7.9 Event Actions

The event action user class allows to perform actions at the begining or the end of an event, defined as the process of one particle through the simulated apparatus. In the MuCool example, we implemented the `MuCoolEventAction::EndOfEventAction` method to print the event number on the screen at the end of the process. We print the number for all events for the first 10, one every ten for the next 90, one every 100 for the following 900, and one every 1000 for the rest:

```
void MuCoolEventAction::EndOfEventAction(const G4Event* currentEvent)
{ // Retrieve the identification number of the current event

G4int iEvt = currentEvent->GetEventID();

// Print on screen the identification number of the just finished
// event. Will print all event numbers for the first 10, 1 every
// 10 events for the first 100, 1 every 100 events above 100,
// and 1 every 1000 above 1000.

if (iEvt < 10) cout << " Event " << iEvt << endl;
else if ((iEvt < 100) && (iEvt%10 == 0)) cout <<
" Event " << iEvt << endl;
else if ((iEvt < 1000) && (iEvt%100 == 0)) cout << " Event "
```

```
<< iEvt << endl;
else if ((iEvt < 10000) && (iEvt%1000 == 0)) cout << " Event "
<< iEvt << endl;
}
```

8 MuCool Header Files

This section is a reference guide for the MuCool example. It contains the header files associated with all the MuCool user classes. Pay attention to the data members of each class since this property may be crucial to access these variables in different parts of the code. This section does not contain the header files of the Beam Tool classes. A reference guide with this information is available on line [2].

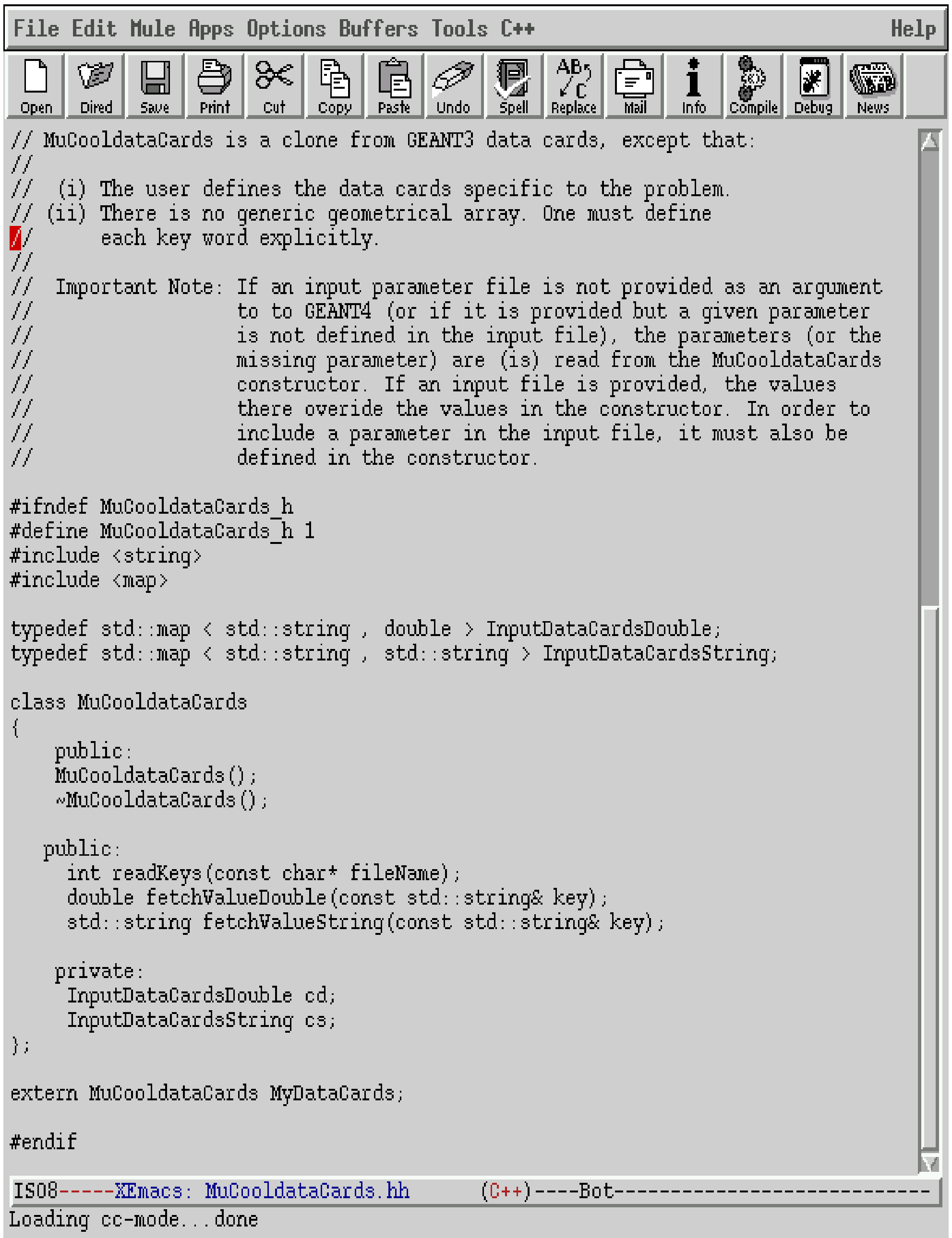


Fig. 34. The `MuCooldataCards.hh` header file.

```
//    Implement the Initial Muon Beam associated with the MuCool example
//
// Date: May 5th 2002
//
#ifndef MuCoolPrimaryGeneratorAction_h
#define MuCoolPrimaryGeneratorAction_h 1

#include <string.h>
#include "g4std/fstream"

#include "G4VUserPrimaryGeneratorAction.hh"
#include "G4ParticleDefinition.hh"

#ifdef ROOTFLAG
#include "TROOT.h"
#include "TNtuple.h"
#include "TFile.h"
#include "TH1.h"
#include "TH2.h"
#endif

extern std::string mode;

class G4ParticleGun;
class G4Event;

#define NUMVARROOT 11

class MuCoolPrimaryGeneratorAction : public G4VUserPrimaryGeneratorAction
{
  public:
     MuCoolPrimaryGeneratorAction();
    ~MuCoolPrimaryGeneratorAction();

  public:
    void GeneratePrimaries(G4Event* anEvent);

  private:
    G4ParticleGun* mymuonGun;  // the gun object
    G4ParticleDefinition* myMuon; // the muon object

    G4double myzStart; // Initial z-position of a muon
    G4double mytimeStart; // Initial time of a muon

    std::string myBeamMode; // Beam mode can be Gaussian or read from file
    G4std::ifstream myinputbeam; // input file

    G4double myEkmuon; // Mean kinetic energy of the initial beam in case of
                     // Gaussian mode
    G4double mysigXY; // sig_x of the initial beam in case of gaussian mode
    G4double mysigTime; // sig_time of initial beam in case of gaussian mode
    G4double mydeltaEoE; // sig_E/E of initial beam in case of gaussian mode
    G4double myBetaFunc; // Initial beta function of the beam

    G4bool myhasRF; // Flag to comunicate that the channel has r.f. an system

// Declare NTuple containing initial beam information

#ifdef ROOTFLAG
    TNtuple *myrt_NTuBegin;

    float myrootvec[NUMVARROOT];
#endif

};
ISO8--**-XEmacs: MuCoolPrimaryGeneratorAction.hh      (C++)----49%----------------
```

Fig. 35. The `MuCoolPrimaryGeneratorAction.hh` header file.

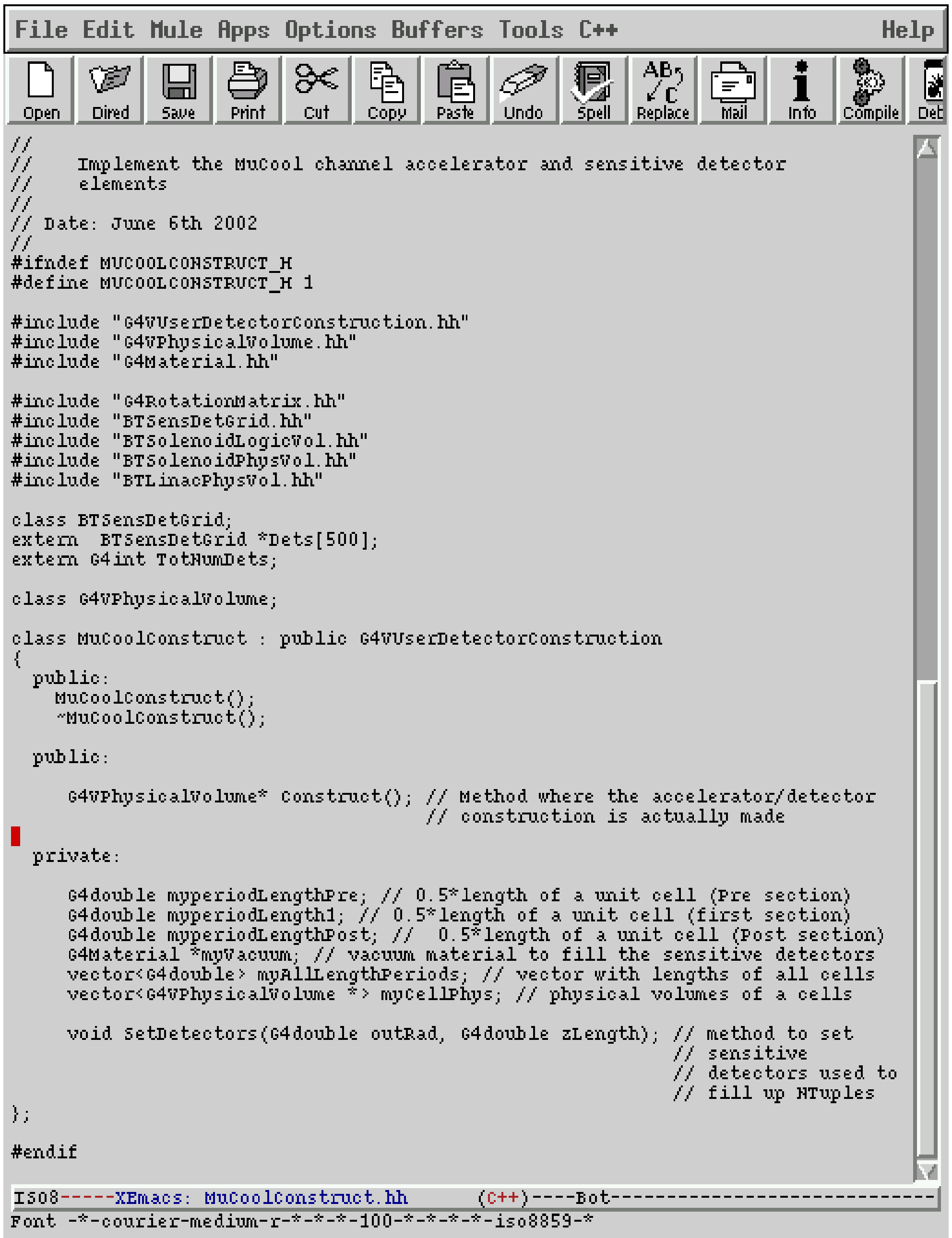
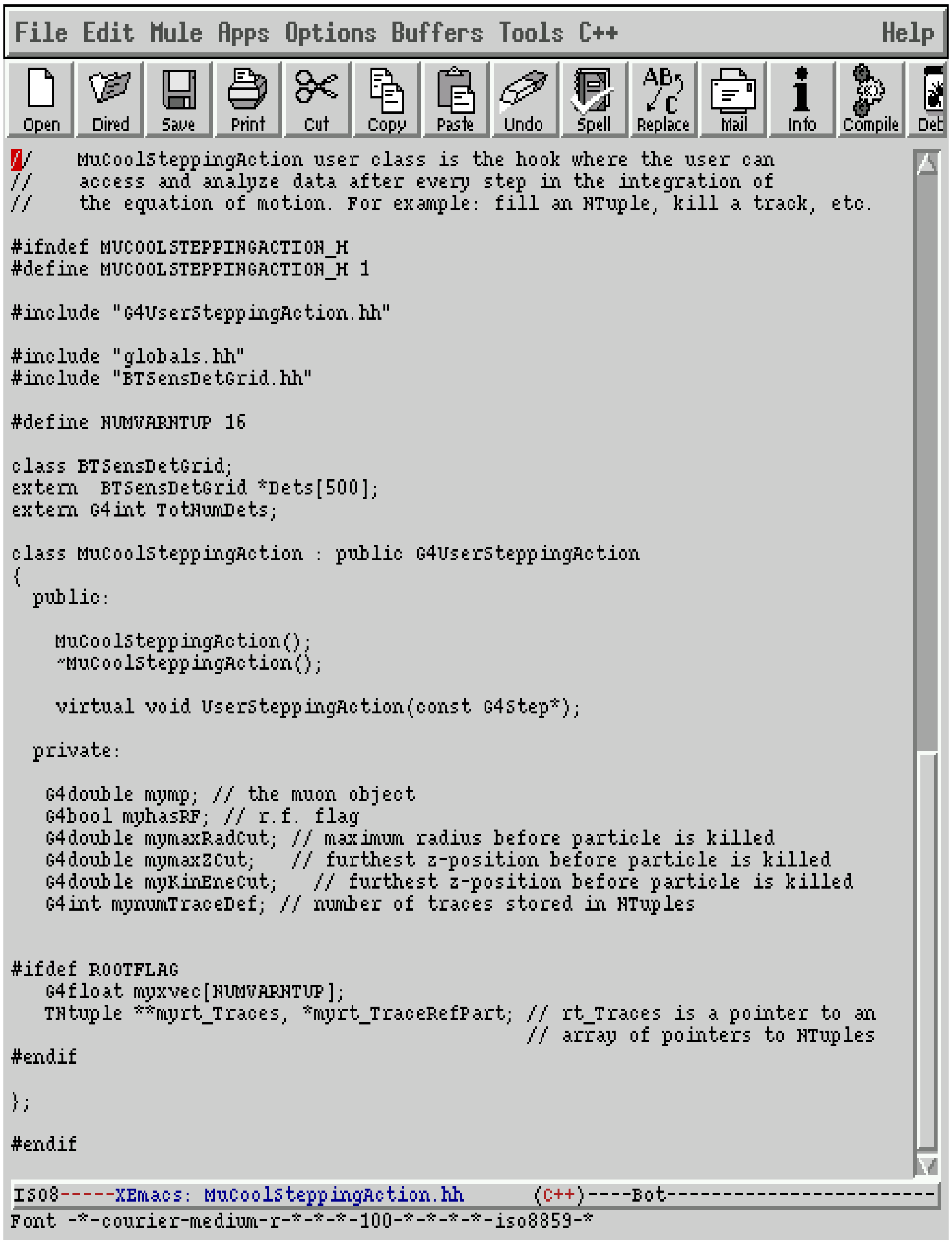
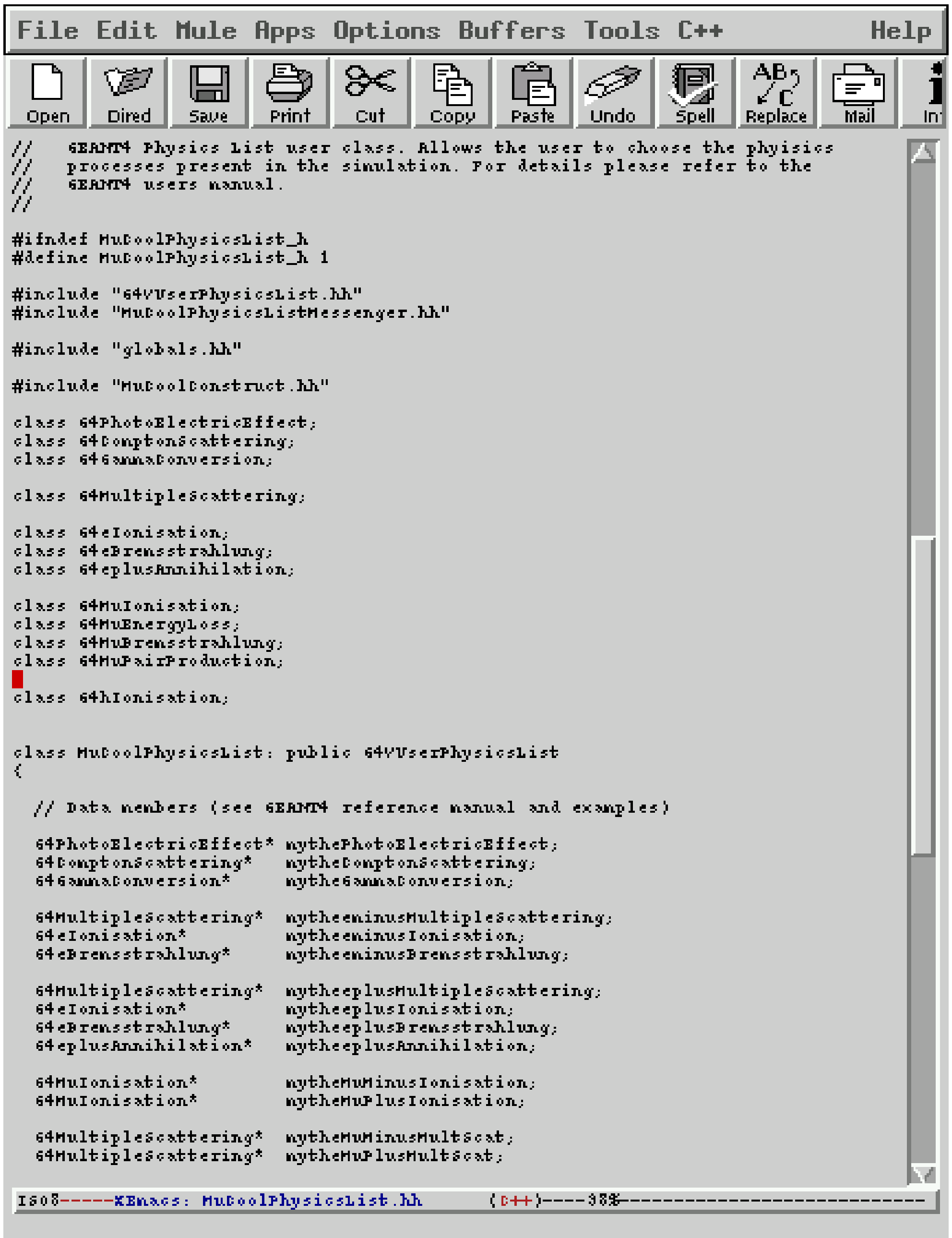


Fig. 36. The `MuCoolConstructor.hh` header file.



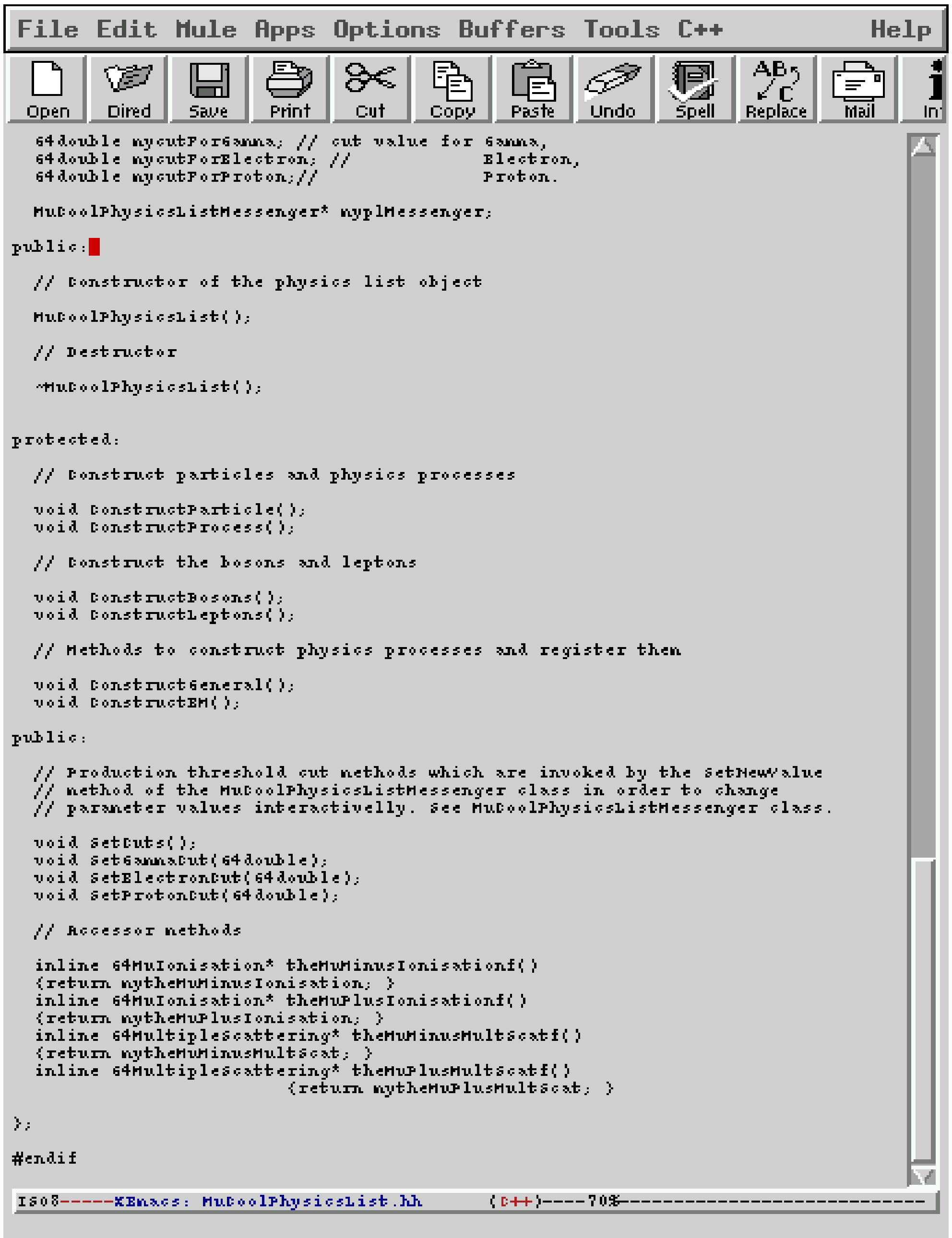

```
//    MuCoolSteppingAction user class is the hook where the user can
//    access and analyze data after every step in the integration of
//    the equation of motion. For example: fill an NTuple, kill a track, etc.

#ifndef MUCOOLSTEPPINGACTION_H
#define MUCOOLSTEPPINGACTION_H 1

#include "G4UserSteppingAction.hh"

#include "globals.hh"
#include "BTSensDetGrid.hh"

#define NUMVARNTUP 16

class BTSensDetGrid;
extern  BTSensDetGrid *Dets[500];
extern G4int TotNumDets;

class MuCoolSteppingAction : public G4UserSteppingAction
{
  public:

    MuCoolSteppingAction();
    ~MuCoolSteppingAction();

    virtual void UserSteppingAction(const G4Step*);

  private:

   G4double mymp; // the muon object
   G4bool myhasRF; // r.f. flag
   G4double mymaxRadCut; // maximum radius before particle is killed
   G4double mymaxZCut;   // furthest z-position before particle is killed
   G4double myKinEneCut;   // furthest z-position before particle is killed
   G4int mynumTraceDef; // number of traces stored in NTuples


#ifdef ROOTFLAG
   G4float myxvec[NUMVARNTUP];
   TNtuple **myrt_Traces, *myrt_TraceRefPart; // rt_Traces is a pointer to an
                                              // array of pointers to NTuples
#endif

};

#endif
```


Fig. 37. The `MuCoolSteppingAction.hh` header file.




```
//   GEANT4 Physics List user class. Allows the user to choose the physics
//   processes present in the simulation. For details please refer to the
//   GEANT4 users manual.
//

#ifndef MuCoolPhysicsList_h
#define MuCoolPhysicsList_h 1

#include "G4VUserPhysicsList.hh"
#include "MuCoolPhysicsListMessenger.hh"

#include "globals.hh"

#include "MuCoolConstruct.hh"

class G4PhotoElectricEffect;
class G4ComptonScattering;
class G4GammaConversion;

class G4MultipleScattering;

class G4eIonisation;
class G4eBremsstrahlung;
class G4eplusAnnihilation;

class G4MuIonisation;
class G4MuEnergyLoss;
class G4MuBremsstrahlung;
class G4MuPairProduction;

class G4hIonisation;


class MuCoolPhysicsList: public G4VUserPhysicsList
{

  // Data members (see GEANT4 reference manual and examples)

  G4PhotoElectricEffect* mythePhotoElectricEffect;
  G4ComptonScattering*   mytheComptonScattering;
  G4GammaConversion*     mytheGammaConversion;

  G4MultipleScattering*  mytheeminusMultipleScattering;
  G4eIonisation*         mytheeminusIonisation;
  G4eBremsstrahlung*     mytheeminusBremsstrahlung;

  G4MultipleScattering*  mytheeplusMultipleScattering;
  G4eIonisation*         mytheeplusIonisation;
  G4eBremsstrahlung*     mytheeplusBremsstrahlung;
  G4eplusAnnihilation*   mytheeplusAnnihilation;

  G4MuIonisation*        mytheMuMinusIonisation;
  G4MuIonisation*        mytheMuPlusIonisation;

  G4MultipleScattering*  mytheMuMinusMultScat;
  G4MultipleScattering*  mytheMuPlusMultScat;
```


Fig. 38. The `MuCoolPhysicsList.hh` header file (first part).




```
  G4double mycutForGamma; // cut value for Gamma,
  G4double mycutForElectron; //           Electron,
  G4double mycutForProton;//              Proton.

  MuCoolPhysicsListMessenger* myplMessenger;

public:

  // Constructor of the physics list object

  MuCoolPhysicsList();

  // Destructor

  ~MuCoolPhysicsList();


protected:

  // Construct particles and physics processes

  void ConstructParticle();
  void ConstructProcess();

  // Construct the bosons and leptons

  void ConstructBosons();
  void ConstructLeptons();

  // Methods to construct physics processes and register them

  void ConstructGeneral();
  void ConstructEM();

public:

  // Production threshold cut methods which are invoked by the SetNewValue
  // method of the MuCoolPhysicsListMessenger class in order to change
  // parameter values interactivelly. See MuCoolPhysicsListMessenger class.

  void SetCuts();
  void SetGammaCut(G4double);
  void SetElectronCut(G4double);
  void SetProtonCut(G4double);

  // Accessor methods

  inline G4MuIonisation* theMuMinusIonisationf()
  {return mytheMuMinusIonisation; }
  inline G4MuIonisation* theMuPlusIonisationf()
  {return mytheMuPlusIonisation; }
  inline G4MultipleScattering* theMuMinusMultScatf()
  {return mytheMuMinusMultScat; }
  inline G4MultipleScattering* theMuPlusMultScatf()
                        {return mytheMuPlusMultScat; }

};

#endif

ISO8-----XEmacs: MuCoolPhysicsList.hh      (C++)----70%---------------------------
```

Fig. 39. The `MuCoolPhysicsList.hh` header file (second part).


```
//   The MuCoolPhysicsListMessenger class contains messenger objects which
//   allow to setup run parameters interactivelly through predefined commands
//

#ifndef MuCoolPhysicsListMessenger_h
#define MuCoolPhysicsListMessenger_h 1

#include "globals.hh"
#include "G4UImessenger.hh"

class MuCoolPhysicsList;
class G4UIcmdWithADoubleAndUnit;

class MuCoolPhysicsListMessenger: public G4UImessenger
{
  // Data members

  MuCoolPhysicsList* physList; // pointer to the physics list object

  // Pointers to user interface command (UIcmd) objects. They set a production
  // cut in range for gammas, electors, protons, others. This means that given
  // a primary particle going through matter (for example a muon), a
  // secondary particle (electron, gamma, etc) will not be generated from the
  // primary once the energy of primary particle is low enough so that it
  // would come to rest in a given range (set cut value by range). Cuts could
  // also be set directly by energy. For more details please refer to the
  // GEANT4 user's manual

  G4UIcmdWithADoubleAndUnit* cutGCmd; // UI command to set cut values by
                                      // range for gammas
  G4UIcmdWithADoubleAndUnit* cutECmd; // UI command to set cut values by
                                      // range for electrons
  G4UIcmdWithADoubleAndUnit* cutPCmd; // UI command to set cut values by
                                      // range for protons and others
  G4UIcmdWithADoubleAndUnit* eCmd; // UI command to set cut values by
                                   // energy for charged particles

public:

  // Constructor of the messanger object

  MuCoolPhysicsListMessenger(MuCoolPhysicsList*);

  // Destructor

  ~MuCoolPhysicsListMessenger();

  // Method to actually set the value of the cut using a given command

  void SetNewValue(G4UIcommand*, G4String);

};

#endif
```

Fig. 40. The `MuCoolPhysicsListMessenger.hh` header file.

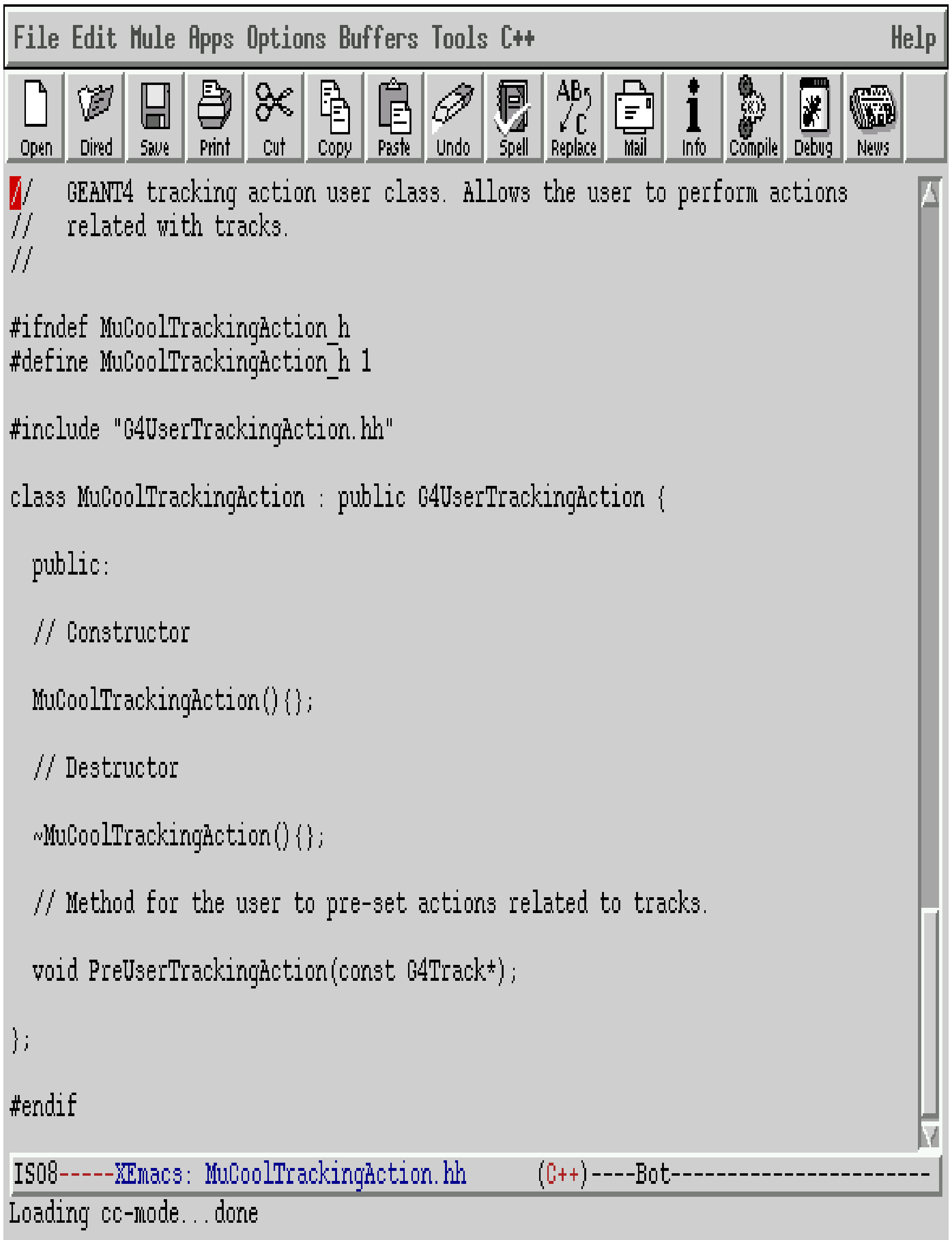
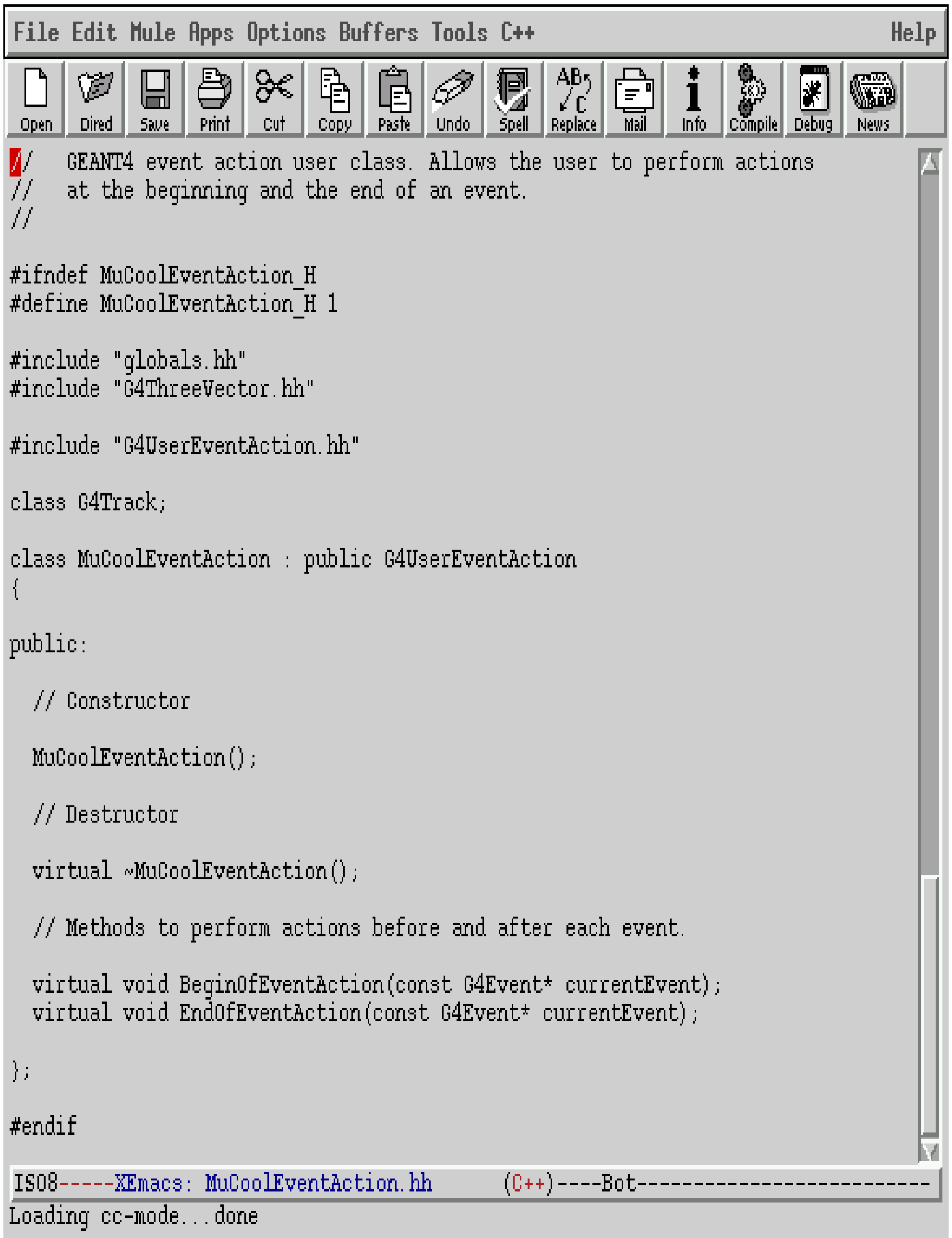
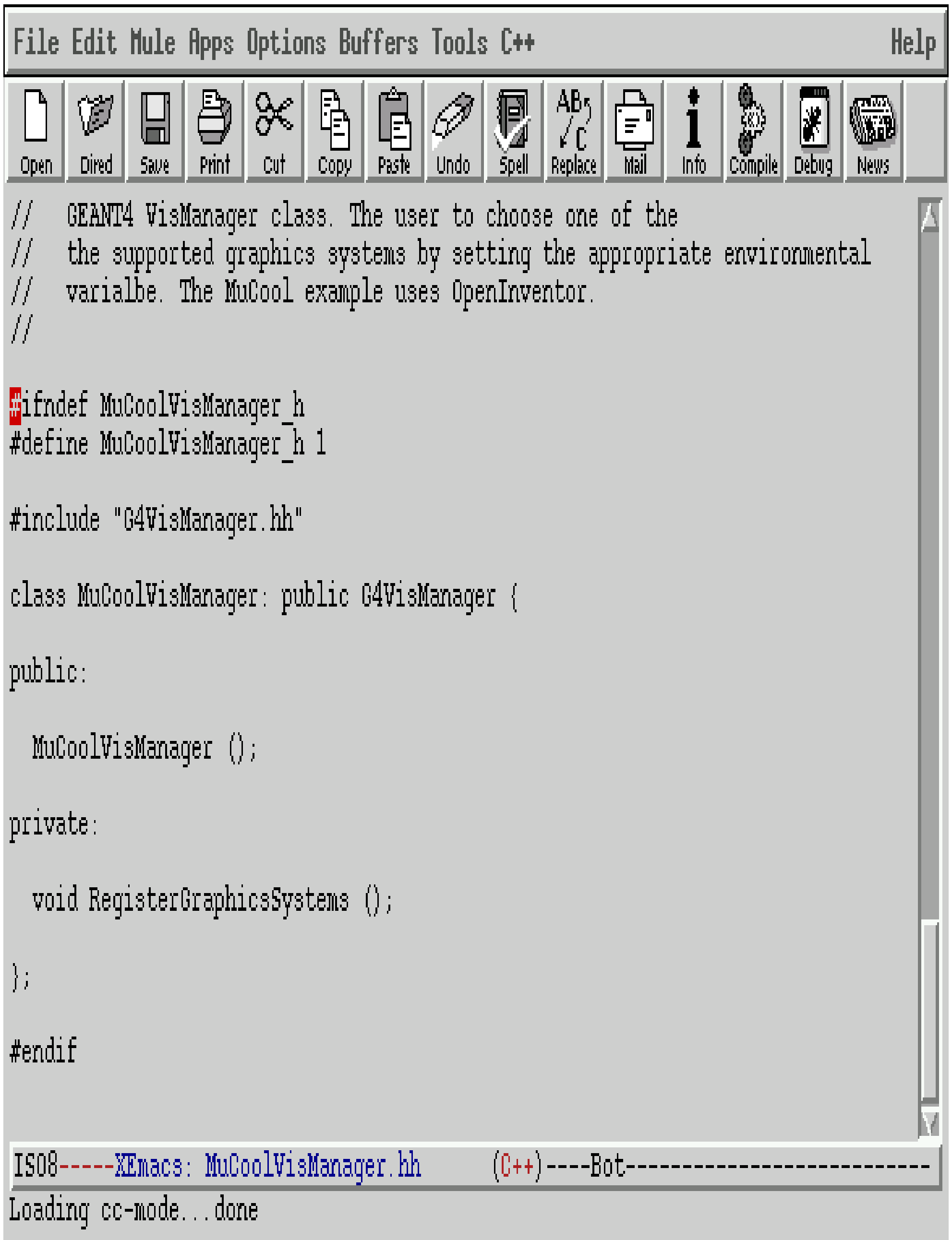


Fig. 41. The `MuCoolTrackingAction.hh` header file.




```cpp
//   GEANT4 event action user class. Allows the user to perform actions
//   at the beginning and the end of an event.
//

#ifndef MuCoolEventAction_H
#define MuCoolEventAction_H 1

#include "globals.hh"
#include "G4ThreeVector.hh"

#include "G4UserEventAction.hh"

class G4Track;

class MuCoolEventAction : public G4UserEventAction
{

public:

  // Constructor

  MuCoolEventAction();

  // Destructor

  virtual ~MuCoolEventAction();

  // Methods to perform actions before and after each event.

  virtual void BeginOfEventAction(const G4Event* currentEvent);
  virtual void EndOfEventAction(const G4Event* currentEvent);

};

#endif
```

Fig. 42. The `MuCoolEventAction.hh` header file.




```
//   GEANT4 VisManager class. The user to choose one of the
//   the supported graphics systems by setting the appropriate environmental
//   varialbe. The MuCool example uses OpenInventor.
//

#ifndef MuCoolVisManager_h
#define MuCoolVisManager_h 1

#include "G4VisManager.hh"

class MuCoolVisManager: public G4VisManager {

public:

  MuCoolVisManager ();

private:

  void RegisterGraphicsSystems ();

};

#endif
```

Fig. 43. The `MuCoolVisManager.hh` header file.

9 Class Structure and Program Flow

This Section is aimed to developers or users interested in learning about how the Beam Tools work. It is an "abstract" of the reference guide. We present some information on the program flow and class structure, but little code detail. The Beam Tools header and code implementation files are thoroughly commented and available on line as a reference guide [2].

9.1 The Magnet Classes

- The `BTSheet Class` inherits from `G4MagneticField`. The class objects are field maps generated by an infinitecimally thin solenoidal current sheet. The class data members are all the parameters necessary to generate analytically a magnetic field in r-z space (there is φ symmetry). No geometric volumes or materials are associated with the `BTSheet` objects. `GetFieldValue` is a concrete method of `BTSheet` inherited from `G4Field`, through `G4MagneticField`. It returns the field value at a given point in space and time.
- The `BTSolenoid Class` inherits from `G4MagneticField`. The class objects are field maps in the form of a grid in r-z space. They are generated by a set of infinitecimally thin solenoidal current sheets. The `BTSheet` objects forming the solenoid are data members of `BTSolenoid`, as well as the `BTSpline1D` objects containing the spline fits of B_z and B_r versus z for each r in the field grid. No geometric volumes or materials are associated with `BTSolenoid`. The field at a point in space and time is accessed through a `GetFieldValue` method, which performs a linear interpolation in r of the spline fit objects.
- The `BTSolenoidLogicVol Class` defines the material and physical size of the coil system which is represented by the set of current sheets. A `BTSolenoid` must first be constructed out of a list of current `BTSheets`. The `BTSolenoid` object is a data member of `BTSolenoidLogicVol`. The `BTSolenoidLogicVol` class constructor creates `G4Tubs` solid volumes and associated logical volumes for the coil system, the shielding, and the inside regions. The last one may be omitted if the pointer to the shielding material is set to 0. Only the logical volumes are defined here. No placement is done.
- The `BTSolenoidPhysVol Class` is the placed version of the `BTSolenoidLogicVol`. It contains the associated `BTSolenoid` object, and pointers to the physical volumes of its logical constituents.
- The `BTMagFieldMap Class` inherits from `G4MagneticField`. The constructor reads the map information from an `ASCII` file containing the value of the field at a set of nodes of a grid. No geometric objects are associated with the field. The field at a point in space and time is accessed through a `GetFieldValue` method, which does a linear interpolation of the values at the nodes of the field grid in r-z space.

- The BTMagFieldMapPlacement Class is a placed BTMagFieldMap object. Only the field is placed because there is no geometric object associated with it (no coil or support system).

9.2 The r.f. Classes

- The BTAccelDevice.hh Class is abstract. All accelerator device classes are derived from BTAccelDevice, which inherits from G4ElectroMagneticField.
- The BTPillBox Class inherits from BTAccelDevice and represents single $\pi/2$ pillbox cavity objects. This time dependent electric field is computed using a simple Bessel function. An important feature is the reference particle mode, used to tune the phase of the r.f. system. In reference mode, the field is static (does not depend on time), and given by a Gaussian function centered at the geometrical center of the Pill Box in z. The integral of E_z over the domain of the function is the energy transfered by the cavity to the particle. The BTPillBox object is a field, with no associated solid. The field at a point in space and time is accessed through a GetFieldValue method.
- The BTrfMap Class inherits from BTAccelDevice. The class objects are e.m. field maps which represent an r.f. cavity. In this way, complex r.f. fields can be measured or generated with software and then included in the geant4 simulation. The field map, in the form of a grid, is read in the BTrfMap constructor from an ASCII file. The BTrfMap object is a field, with no associated solid. A GetFieldValue method returns the field value at a point in space and time by means of a linear interpolation of the field grid.
- The BTrfCavityLogicVol Class constructor creates solids and logical volumes associated with the r.f. field classes. In the case of a map, a vacuum cylinder box (with end cups) represents its limits. In addition to geometry and material parameters of the cavity, the class contains e.m. field and accelerator device information.
- The BTrfWindowLogicVol Class is used together with the BTCavityLogicVol class to create the geometry and logical volume of the r.f. cavity windows, including the support structure.
- The BTLinacPhysVol Class is a placed linac object. A linac is a set of contiguous r.f. cavities, which include the r.f. field, the support and conductor material, windows, and sensitive detectors for the phase delay calculation. The BTLinacPhysVol constructor is overloaded. One version places a linac of Pill Box cavities (phase set at the geometric center of the cavity). A second version places field maps (phase may be set at a distance from the geometric center of the cavity).
- The BTLinacCellPhaseInfo Class contains cavity related information necessary to add the r.f. field to the global electromagnetic field. The object is created in the BTLinacPhysVol constructor. See section 9.3 for more details.

There are two global field classes: one for the case that the field is purely magnetic, and another one in the case there is also an electric component.

- The BTGlobalMagField Class inherits from G4MagneticField. A single BTGlobalMagField object is constructed from the sum of individual magnetic fields generated by all magnetic objects, like solenoids and magnetic field maps. The global magnetic field is accessed by a GetFieldValue method, where the sum of the fields is implemented. To perform these operations, the magnet objects and their positions in global coordinates need to be data members of the global field class. The BTGlobalMagField constructor only does a trivial initialization to zero of some of the data members. The global magnetic field is actually filled up or "included" by the IncludeSolenoid and IncludeMaps methods, which are invoked in the BTSolenoidPhysVol and BTMagFieldMapPlacement constructors, respectively. In summary, the magnet objects, their location, and field scale factor are constructed or defined in the Construct() method of the detector construction user class. This information is then passed to IncludeSolenoid and IncludeMaps through BTSolenoidPhysVol and BTMagFieldMapPlacement. IncludeSolenoid and IncludeMaps assign this information to the data members of BTGlobalMagField for its use by GetFieldValue. Since this is the GetFieldValue associated with the field fed to the equation of motion, it will be called internally by Geant4 to retrieve the global magnetic field.
- The BTGlobalEMField Class inherits from G4ElectroMagneticField. A single BTGlobalEMField global e.m. object is constructed from the existing BTGlobalMagField, by adding the fields from the acceleration elements, such as r.f. cavities. The mechanism for feeding the global e.m. field information into Geant4 is the same as in the case of the BTGlobalMagField. The r.f. field is added to the global field by the IncludeAnRFCell method of BTGlobalEMField, which is called in BTLinacPhysVol. The global e.m. field has an additional feature related with the r.f. system phase tuning. The SetPhaseDelayAtZ method is invoked from the UserSteppingAction method only in reference particle mode. It uses the step object at the phase center of each cavity to calculate and set a phase delay for the cavity to operate at the required synchronous phase in a normal run. Although redundant, the BTLinacCellPhaseInfo class is convinient as it defines the right object to be manipulated by the global field. The synchronous phase may be set by the user at either the geometric center of the cavity, or at a distance zPhase from it (only available for maps). This argument is passed to the BTLinacPhysVol constructor, then to the BTLinacCellPhaseInfo object, and finally to the global e.m. by IncludeAnRFCell.

- The `BTAbsObj Class` is the abstract class from which all the absorber objects are derived.
- The `BTCylindricVessel Class` is a concrete class derived from `BTAbsObj`. Objects are physical volumes of a cylindric object, defined as a central cylindric rim (3 cm thick), two end cup rims (same thickness), with thin windows of radius equal to the inner radius of the vessel. The material is the same for the vessel walls and windows, and the window thickness is constant. The vessel is filled with an absorber material.
- The `BTCylindricLense Class` is a concrete class derived from BTAbsObj. Objects are physical volumes of cylindric lenses, that is an absorber with the shape of a cylinder but density which depends parabolically from the radius. The lense is actually constructed from a number of concentric cylindric rings of different density.
- The `BTParabolicLense Class` is a concrete class derived from BTAbsObj. Objects are physical volumes of parabolic lenses, that is an absorber with uniform density and the shape of a lense. It is actually constructed from a number of cylinders of maximum radius in the middle, decreasing parabolically towards the edges.

We thank Mark Fishler, for contributing the data cards and spline fit classes, Jeff Kallenbach for helping with visualization issues, and Walter Brown for providing C++ consultancy. We are also gratefull to the Geant4 Collaboration for answering our questions. In particular, we thank J. Apostolakis, M. Asai, G. Cosmo, M. Maire, L. Urban, and V. Grichine.

References

[1] "The Double Flip Cooling Channel", V. Balbekov, V. D. Elvira, et al. MuCool note #203, 4/20/01. http://www-mucool.fnal.gov/notes/noteSelMin.html

[2] See Fermilab Geant4 web page at: http://www-cpd.fnal.gov/geant4.

[3] See Geant4 home page at: wwwinfo.cern.ch/asd/geant4/geant4.html.

[4] See `Root` home page at: http://root.cern.ch/root.

[5] Open Inventor. Registered trademark of Silicon Graphics Inc.

[6] See `ZOOM` home page at: http://www.fnal.gov/docs/working-groups/fpcltf/Pkg/WebPages/zoom.html.

[7] DPgeant geant is a double precision implementation of Geant3 by P. Lebrun.

[8] "Pseudo-Realistic GEANT4 Simulation of a 44/88 MHz Cooling Channel for the Neutrino Factory", V. D. Elvira, H. Li, P. Spentzouris.
MuCool note #230, 12/10/01.
http://www-mucool.fnal.gov/notes/noteSelMin.html

[9] "Double Field Flip Cooling Channel with Parabolic Absorbers", V. Balbekov.
MuCool note #204, 4/20/01.
http://www-mucool.fnal.gov/notes/noteSelMin.html

[10] "Simulation of a Helical Channel Using GEANT4", V. D. Elvira et al.
MuCool note #193, 02/20/01.
http://www-mucool.fnal.gov/notes/noteSelMin.html