Garfield++ User Guide

Version 2025.1

March 2025

Contents

1. Introduction

Garfield++ is an object-oriented toolkit for the detailed simulation of particle detectors based on ionisation measurement in gases or semiconductors.

For calculating electric fields, the following techniques are currently being offered:

- solutions in the thin-wire limit for devices made of wires and planes;
- interfaces with finite element programs, which can compute approximate fields in nearly arbitrary two- and three-dimensional configurations with dielectrics and conductors;
- an interface with the Synopsys Sentaurus device simulation program [47];
- an interface with the neBEM field solver [20, 21].

For calculating the transport properties of electrons in gas mixtures, an interface to the Magboltz program [6, 7] is available.

The ionisation pattern produced by relativistic charged particles can be simulated using the program Heed [44]. For simulating the ionisation produced by low-energy ions, results calculated using the SRIM software package [50] can be imported. The program Degrade simulates ionisation by electrons.

The present document aims to give an overview of Garfield++, but does not provide an exhaustive description of all classes and functions. A number of examples and code snippets are included which may serve as a basis for the user's own programs. Further examples and information can be found on the website `http://garfieldpp.web.cern.ch/garfieldpp/`. If you have questions, doubts, comments *etc.* about the code or this manual, please do not hesitate to contact the authors.

Fig. 1.1 gives an overview of the different types of classes and their interplay. Readers familiar with the structure of (Fortran) Garfield [48] will recognize a rough correspondence between the above classes and the sections of Garfield. `Medium` classes, for instance, can be regarded as the counterpart of the `&GAS` section; `Component` classes are similar in scope to the `&CELL` section.

Garfield++ also includes a number of classes for visualization purposes, *e. g.* for plotting drift lines, making a contour plot of the electrostatic potential or inspecting the layout of the detector. These classes rely extensively on the ROOT framework [9].

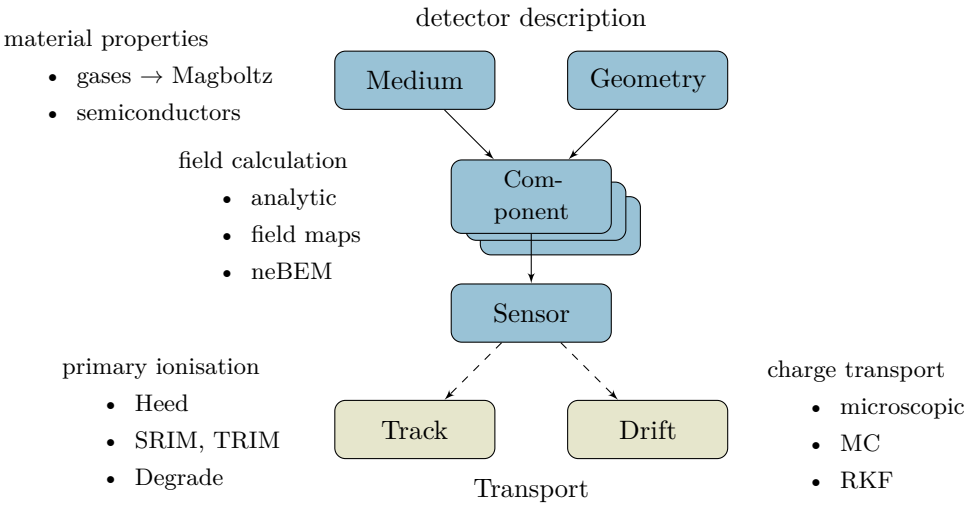

Figure 1.1. Overview of the main classes in Garfield++ and their interplay.

2. Getting started

2.1. Prerequisites

To build Garfield++, and a project or application that depends on it, you need to have the following software correctly installed and configured on your machine.

- ROOT 6,
- GSL[1] (GNU Scientific Library),
- CMake[2] (version 3.12 or or later),
- a C++ compiler compatible with the same C++ standard with which ROOT was compiled,
- a Fortran compiler,
- (optionally) OpenMP[3] to enable some additional parallel computations.

For ROOT installation instructions, see

- `https://root.cern.ch/building-root` or
- `https://root.cern.ch/downloading-root`.

Declare the `ROOTSYS` environment variable to point to the base folder of the ROOT installation. This operation is typically performed by the ROOT initialization script, called `thisroot.sh`. Executing that script will also correctly initialize the right environment variables to install Garfield.

2.2. Downloading the source code

The Garfield++ source code is managed by a git repository hosted on the CERN GitLab[4] server, `https://gitlab.cern.ch/garfield/garfieldpp`.

Choose a folder where the source code is to be downloaded. Note that the chosen folder must be empty or non-existing. We will identify this folder with an environment variable named `GARFIELD_HOME`. Note that this is not strictly required and you can simply replace the chosen path in all the following commands where that variable appears. To define that variable in the bash shell family type

```
export GARFIELD_HOME=/home/git/garfield
```

(replace `/home/git/garfield` by the path of your choice).

For (t)csh-type shells, type

[1] https://www.gnu.org/software/gsl/
[2] https://cmake.org/
[3] https://openmp.org
[4] https://gitlab.cern.ch/help/gitlab-basics/start-using-git.md

```
setenv GARFIELD_HOME /home/git/garfield
```

Download the code from the repository, either using SSH access[5]

```
git clone ssh://git@gitlab.cern.ch:7999/garfield/garfieldpp.git $GARFIELD_HOME
```

or HTTPS access

```
git clone https://gitlab.cern.ch/garfield/garfieldpp.git $GARFIELD_HOME
```

To update the source code with the latest changes, run the following command from the `GARFIELD_HOME` folder:

```
git pull
```

2.3. Building the project

Garfield++ uses the CMake build generator to create the actual build system that is used to compile the binaries of the programs we want to build. CMake is a cross-platform scripting language that allows one to define the rules to build and install a complex project and takes care of generating the low-level files necessary for the build system of choice. On Linux systems that build system typically defaults to the `GNU Make` program. We will therefore assume that `GNU Make` is used to build Garfield++.

The process is divided in two phases. During the build phase, the source files will be used to generate the necessary binaries, along with a large quantities of temporary files, used in the intermediate phases of the build. In the install phase only the necessary files are copied to the final destination.

First we will create a build directory, and move into it to execute the first phase. This can be any folder on the user filesystem. For simplicity we will use a subfolder of the place where the source code was downloaded.

```
mkdir $GARFIELD_HOME/build
cd build
```

Inside the build directory we can run CMake to generate the build system.

```
cmake $GARFIELD_HOME
```

The build can be customized in several ways through CMake by defining a set of internal variables that modify the output accordingly. To set a new value for a CMake variable one can use the `-D<var>=<value>` syntax at the command line. The most relevant parameters that a user may want to customize are described below.

[5] See `https://gitlab.cern.ch/help/gitlab-basics/create-your-ssh-keys.md` for instructions how to create and upload the SSH keys for GitLab.

Installation folder By default Garfield is installed in the folder pointed to by the environment variable `GARFIELD_INSTALL` or, if that variable is missing, in a subfolder of the source directory, that is `$GARFIELD_HOME/install`. To install it elsewhere define the `CMAKE_INSTALL_PREFIX` variable, using a similar command with an appropriate target folder:

```
cmake -DCMAKE_INSTALL_PREFIX=/home/mygarfield $GARFIELD_HOME
```

Debug and optimization mode This is controlled by the CMake variable `CMAKE_BUILD_TYPE` which can have one of the following values:

Release Enables all the compiler optimizations to achieve the best performance

Debug Disables all the compiler optimizations and add the debug symbols to the binary to be able to use an external debugger on the code

RelWithDebInfo Enables all the compiler optimization but stores the debug symbols in the final binaries. This increases the binary size and may reduce the performances

MinSizeRel Enables all the compiler optimization necessary to obtain the smaller executable possible

By default Garfield++ is built in *Release* mode. To add the debugging symbol use the following command before building

```
cmake -DCMAKE_BUILD_TYPE=RelWithDebInfo $GARFIELD_HOME
```

Those variables and many other can be set through a textual or graphical user interface, `ccmake` or `cmake-gui` respectively, that needs to be installed separately.

Once CMake has generated the build system, you can execute the following to compile and install Garfield++.

```
make && make install
```

Once the installation is done, Garfield requires the definition of an environment variable named `HEED_DATABASE` to identify the location of the Heed cross-section database, located in the subfolder `share/Heed/database` of the installation path. Additionally, to build applications that make use of Garfield it may be convenient to append the installation path to an environment variable named `CMAKE_PREFIX_PATH`. To simplify all this, the build procedure generates a shell script, named `setupGarfield.sh`, located in the subfolder `share/Garfield` of the installation path, which correctly defines all those variables. You can append the execution of that script to your shell initialization script (*e. g.* `.bashrc` for the Bash shell) to setup Garfield automatically.

After updating the source code you can run the `make` command from the build folder to update the build. Sometimes it may be necessary to restart from a clean slate, in which case one can remove the build folder completely and restart the procedures of this section.

2.4. Building an application

The recommended way to build a Garfield++-based C++ application is using CMake. Let us consider as an example the program `gem.C` (see Sec. 2.6.2) which together with the corresponding

`CMakeLists.txt` can be found in the directory `Examples/Gem` of the source tree. As a starting point, we assume that you have built Garfield++ using the instructions above and set up the necessary environment variables.

- To keep the source tree clean, and since you will probably want to modify the program according to your needs, it is a good idea to copy the folder to another location.

```
cp -r $GARFIELD_HOME/Examples/Gem .
```

- Create a build directory.

```
mkdir Gem/build; cd Gem/build
```

- Setup the environment.

```
source $GARFIELD_HOME/install/share/Garfield/setupGarfield.sh
```

- Run CMake,

```
cmake ..
```

  followed by

```
make
```

- In addition to the executable (`gem`), the `build` folder should now also contain the field map (`*.lis`) files which have been copied there during the CMake step.
- To run the application, type

```
./gem
```

2.5. Python

Thanks to PyROOT, the Garfield++ classes can also be used from Python. After building the project and setting up the environment following the instructions above, one can load the relevant libraries in the Python interpreter using

```
import ROOT
import Garfield
```

2.6. Examples

Section 2.6.1 discusses the calculation of transport parameters with Magboltz, the use of analytic field calculation techniques, “macroscopic” simulation of electron and ion drift lines, and the calculation of induced signals.

Microscopic transport of electrons and the use of finite element field maps are introduced in Sec. 2.6.2.

Section 2.6.3 presents an example of the simulation of drift lines and induced signals in a silicon sensor.

Further examples can be found on the webpage (`http://garfieldpp.web.cern.ch/garfieldpp/Examples`) and in the directory `Examples` of the source tree.

2.6.1. Drift tube

In this example, we consider a drift tube with an outer diameter of 15 mm and a wire diameter of 50 µm, similar to the ATLAS small-diameter muon drift tubes (sMDTs).

Gas table

First, we prepare a table of transport parameters (drift velocity, diffusion coefficients, Townsend coefficient, and attachment coefficient) as a function of the electric field $\mathbf{E}$ (and, in general, also the magnetic field $\mathbf{B}$ as well as the angle between $\mathbf{E}$ and $\mathbf{B}$). In this example, we use a gas mixture of 93% argon and 7% carbon dioxide at a pressure of 3 atm and room temperature.

```
MediumMagboltz gas("ar", 93., "co2", 7.);
// Set temperature [K] and pressure [Torr].
gas.SetPressure(3 * 760.);
gas.SetTemperature(293.15);
```

We also have to specify the number of electric fields to be included in the table and the electric field range to be covered. Here we use 20 field points between 100 V / cm and 100 kV / cm with logarithmic spacing.

```
gas.SetFieldGrid(100., 100.e3, 20, true);
```

Now we run Magboltz to generate a gas table for this grid. As input parameter we have to specify the number of collisions (in multiples of 10^7) over which the electron is traced by Magboltz.

```
const int ncoll = 10;
gas.GenerateGasTable(ncoll);
```

This calculation will take a while, don't panic. After the calculation is finished, we save the gas table to a file for later use.

```
gas.WriteGasFile("ar_93_co2_7.gas");
```

Once we have saved the transport parameters to file we can skip the steps above, and simply import the table in our program using

```
gas.LoadGasFile("ar_93_co2_7.gas");
```

In order to make sure the calculation of the gas table was successful, it is a good idea to plot, for instance, the drift velocity as a function of the electric field.

```
ViewMedium mediumView(&gas);
mediumView.PlotElectronVelocity('e');
```

Electric Field

For calculating the electric field inside the tube, we use the class `ComponentAnalyticField` which can handle (two-dimensional) arrangements of wires, planes and tubes.

```
ComponentAnalyticField cmp;
```

We first set the medium inside the active region.

```
cmp.SetMedium(&gas);
```

Next we add the elements defining the electric field, *i.e.* the wire (which we label "s") and the tube.

```
// Wire radius [cm]
const double rWire = 25.e-4;
// Outer radius of the tube [cm]
const double rTube = 0.71;
// Voltages
const double vWire = 2730.;
const double vTube =    0.;
// Add the wire in the centre.
cmp.AddWire(0, 0, 2 * rWire, vWire, "s");
// Add the tube.
cmp.AddTube(rTube, vTube, 0);
```

Finally we assemble a `Sensor` object which acts as an interface to the transport classes discussed below.

```
// Calculate the electric field using the Component object cmp.
Sensor sensor(&cmp);
// Request signal calculation for the electrode named "s",
// using the weighting field provided by the Component object cmp.
sensor.AddElectrode(&cmp, "s");
```

We further need to set the time interval within which the signal is recorded and the granularity (bin width). In this example, we use use 1000 bins with a width of 0.5 ns.

```
const double tstep = 0.5;
const double tmin = -0.5 * tstep;
const unsigned int nbins = 1000;
sensor.SetTimeWindow(tmin, tstep, nbins);
```

Drift lines from a track

We use Heed (Sec. 5.1) to simulate the ionisation produced by a charged particle crossing the tube (a 170 GeV muon in our example).

```
TrackHeed track(&sensor);
track.SetParticle("muon");
track.SetEnergy(170.e9);
```

The drift lines of the electrons created along the track are calculated using Runge-Kutta-Fehlberg (RKF) integration, implemented in the class `DriftLineRKF`. This method uses the previously computed tables of transport parameters to calculate drift lines and multiplication.

```
DriftLineRKF drift(&sensor);
```

Let us consider a track that passes at a distance of 3 mm from the wire centre. After simulating the passage of the charged particle, we loop over the "clusters" (*i.e.* the ionizing collisions of the primary particle) along the track and calculate a drift line for each electron produced in the cluster.

```
const double rTrack = 0.3;
const double x0 = rTrack;
const double y0 = -sqrt(rTube * rTube - rTrack * rTrack);
track.NewTrack(x0, y0, 0, 0, 0, 1, 0);
// Loop over the clusters along the track.
for (const auto& cluster : track.GetClusters()) {
  // Loop over the electrons in the cluster.
  for (const auto& electron : cluster.electrons) {
    drift.DriftElectron(electron.x, electron.y, electron.z, electron.t);
  }
}
```

As a check whether the simulation is doing something sensible, it can be useful to visualize the drift lines. Before simulating the charged particle track and the electron drift lines, we have to instruct `TrackHeed` and `DriftLineRKF` to pass the coordinates of the clusters and the points along the drift line to a `ViewDrift` object which then takes care of plotting them.

```
// Create a canvas.
cD = new TCanvas("cD", "", 600, 600);
ViewDrift driftView;
driftView.SetCanvas(cD);
drift.EnablePlotting(&driftView);
track.EnablePlotting(&driftView);
```

We use the class `ViewCell` to draw the outline of the tube and the position of the wire on the same plot as the drift lines.

```
ViewCell cellView(&cmp);
cellView.SetCanvas(cD);
```

After we've simulated all drift lines from a charged particle track, we create a plot using

```
cellView.Plot2d();
constexpr bool twod = true;
constexpr bool drawaxis = false;
driftView.Plot(twod, drawaxis);
```

and we plot the current induced on the wire by the drift lines simulated in the previous step.

```
TCanvas* cS = new TCanvas("cS", "", 600, 600);
sensor.PlotSignal("s", cS);
```

Ion tail

So far we have only considered the electron contribution to the induced signal. If we want to include the contribution from the ions produced in the avalanche we need to import a table of ion mobilities

```
gas.LoadIonMobility("IonMobility_Ar+_Ar.txt");
```

By default, `DriftLineRKF` will then include the ion tail in the simulation.

2.6.2. GEM

In this example, we will simulate the drift of electrons and ions in a standard GEM [40], which consists of a 50 µm thick kapton foil coated on both sides with a 5 µm layer of copper, with a hexagonal pattern of holes (outer hole diameter: 70 µm, inner hole diameter: 50 µm, pitch: 140 µm) etched into the foil.

Field map

As a first step, we need to calculate the electric field in the GEM. In this example, we use Ansys [2] but the steps for importing field maps from other finite-element solvers like Elmer [15] or Comsol [11] are very similar.

In the following we assume that the output files resulting from the Ansys run are located in the current working directory. The initialisation of `ComponentAnsys123` consists of

- loading the mesh (`ELIST.lis`, `NLIST.lis`), the list of nodal solutions (`PRNSOL.lis`), and the material properties (`MPLIST.lis`);
- specifying the length unit of the values given in the `.LIS` files;
- setting the appropriate periodicities/symmetries.

```
ComponentAnsys123 fm;
// Load the field map.
fm.Initialise("ELIST.lis", "NLIST.lis", "MPLIST.lis", "PRNSOL.lis", "mm");
// Set the periodicities
fm.EnableMirrorPeriodicityX();
fm.EnableMirrorPeriodicityY();
// Print some information about the cell dimensions.
fm.PrintRange();
```

We need to apply mirror periodicity in x and y in `ComponentAnsys123` since we had exploited the symmetry of the geometry and modelled only half a hole in Ansys.

Using the class `ViewField`, we do a visual inspection of the field map to make sure it looks sensible. We first make a plot of the potential along the hole axis (z axis).

```
ViewField fieldView(&fm);
// Plot the potential along the hole axis.
fieldView.PlotProfile(0., 0., 0.02, 0., 0., -0.02);
```

We also make a contour plot of the potential in the $x - z$ plane.

```
const double pitch = 0.014;
// Set the normal vector (0, -1, 0) of the viewing plane.
fieldView.SetPlane(0., -1., 0., 0., 0., 0.);
fieldView.SetArea(-pitch / 2., -0.02, pitch / 2., 0.02);
fieldView.SetVoltageRange(-160., 160.);
fieldView.PlotContour();
```

Next we create a `Sensor` based on the field map component.

```
Sensor sensor(&fm);
```

Normally, particles are transported until they exit the mesh. To speed up the calculation we restrict the drift region to $-100\mu\text{m} < z < +250\mu\text{m}$.

```
sensor.SetArea(-5 * pitch, -5 * pitch, -0.01, 5 * pitch,  5 * pitch,  0.025);
```

Gas

We use a gas mixture of 80% argon and 20% CO_2.

```
MediumMagboltz gas("ar", 80., "co2", 20.);
// Set temperature [K] and pressure [Torr].
gas.SetTemperature(293.15);
gas.SetPressure(760.);
```

Electron collision rates In this example, we will simulate electron avalanches using a "microscopic" Monte Carlo method, based on the electron-atom/molecule cross-sections in the database of the Magboltz program. As discussed in more detail in Sec. 6.3, the algorithm takes as input the collision rates (as function of the electron's kinetic energy) for each scattering mechanism that can take place in the gas. The preparation of the tables of collision rates and the interpolation in these tables is done by the class `MediumMagboltz`, which – as the name suggests – provides an interface to Magboltz.

In `MediumMagboltz` the collision rates are stored on an evenly spaced energy grid. The max. energy can be set by the user. For avalanche calculations, 50 – 200 eV is usually a reasonable choice.

```
gas.SetMaxElectronEnergy(200.);
gas.Initialise();
```

Penning transfer Argon includes a number of excitation levels with an excitation energy exceeding the ionisation potential of CO_2 (13.78 eV). These excited states can contribute to the gain, since (part of) the excitation energy can be transferred to a CO_2 molecule through collisions or by photo-ionisation.

In the simulation, this so-called Penning effect can be described in terms of a probability r that an excitation is converted to an ionising collision (Sec. 3.2.3).

```
// Penning transfer probability.
const double rPenning = 0.57;
// Mean distance from the point of excitation.
const double lambdaPenning = 0.;
gas.EnablePenningTransfer(rPenning, lambdaPenning, "ar");
```

Ion transport properties Unlike electrons, ions cannnot be tracked microscopically in Garfield++. Moreover, there is no program like Magboltz that can compute the macroscopic transport properties (such as the drift velocity), so we have to provide these data ourselves. In this example, we use the mobility of Ar^+ ions in Ar as an approximation because there is no literature data for drift in the mixture. Example files with mobility data for various pure gases are located in the `Data` directory of the project source tree and are copied to the `install` directory during installation. If you don't provide a full path or if the mobility file is not in your current working directory, `MediumMagboltz` will look for it in the subdirectory `share/Garfield/Data` of the `install` folder.

```
gas.LoadIonMobility("IonMobility_Ar+_Ar.txt");
```

Associating the gas to a field map region In order to track a particle through the detector we have to tell `ComponentAnsys123` which field map material corresponds to which `Medium`. The gas can be distinguished from the other materials (here: kapton and copper) by its dielectric constant, in our case $\varepsilon = 1$.

```
const size_t nMaterials = fm.GetNumberOfMaterials();
for (size_t i = 0; i < nMaterials; ++i) {
  const double eps = fm.GetPermittivity(i);
  if (fabs(eps - 1.) < 1.e-3) fm.SetMedium(i, &gas);
}
// Print a list of the field map materials (for information).
fm.PrintMaterials();
```

Instead of iterating over the materials and retrieving the relative dielectric constant, we can also use the helper function `SetGas`.

```
fm.SetGas(&gas);
```

Electron transport

Microscopic tracking is handled by the class `AvalancheMicroscopic` (Sec. 6.3).

```
AvalancheMicroscopic aval(&sensor);
```

We are now ready to simulate an electron avalanche in the GEM. We place the initial electron 200 µm above the centre of the GEM hole and set its initial energy to a typical energy in the electric field of the drift gap.

```
// Initial position [cm] and starting time [ns]
double x0 = 0., y0 = 0., z0 = 0.02;
double t0 = 0.;
// Initial energy [eV]
double e0 = 0.1;
// Initial direction
// In case of a null vector, the initial direction is randomized.
double dx0 = 0., dy0 = 0., dz0 = 0.;
// Calculate an electron avalanche.
aval.AvalancheElectron(x0, y0, 0, t0, e0, dx0, dy0, dz0);
```

After the calculation, we can extract information such as the number of electrons/ions produced in the avalanche and the start- and endpoints of all electron trajectories.

```
int ne, ni;
// Get the number of electrons and ions in the avalanche.
aval.GetAvalancheSize(ne, ni);
// Loop over the electrons in the avalanche.
for (const auto& electron : aval.GetElectrons()) {
  // Initial position.
  const auto& p0 = electron.path.front();
  std::cout << "Electron␣started␣at␣("
            << p0.x << ",␣" << p0.y << ",␣" << p0.z << ")\n";
  // Final position.
  const auto& p1 = electron.path.back();
  std::cout << "Electron␣stopped␣at␣("
            << p1.x << ",␣" << p1.y << ",␣" << p1.z << ")\n";
  std::cout << "Status␣code:␣" << electron.status << "\n";
}
```

Ion transport For tracking the ions, we use the class `AvalancheMC`, which takes as input the macroscopic drift velocity as function of the electric field and simulates drift lines using a Monte Carlo technique (Sec. 6.2).

```
AvalancheMC drift(&sensor);
drift.SetDistanceSteps(2.e-4);
```

After simulating an electron avalanche, we loop over all the electron trajectories, and calculate an ion drift line starting from the same initial point as the electron.

```
// Loop over the avalanche electrons.
```

```
for (const auto& electron : aval.GetElectrons()) {
  const auto& p0 = electron.path[0];
  drift.DriftIon(p0.x, p0.y, p0.z, p0.t);
  // ...
}
```

We can subsequently retrieve the endpoint of the ion drift line using

```
const auto& p2 = drift.GetIons()[0].path.back()
std::cout << "Ion␣stopped␣at␣("
          << p2.x << ",␣" << p2.y << ",␣" << p2.z << ").\n";
```

Visualizing the drift lines

To plot the electron and ion drift lines together with the geometry, we use the classes `ViewDrift` and `ViewFEMesh`. When setting up the `AvalancheMicroscopic` and `AvalancheMC` objects, we need to switch on the storage of the drift line points and attach a pointer to a `ViewDrift` object.

```
ViewDrift driftView;
aval.EnablePlotting(&driftView);
drift.EnablePlotting(&driftView);
```

After having calculated the electron avalanche and ion drift lines, we create a plot using the snippet of code below.

```
ViewFEMesh* meshView = new ViewFEMesh(&fm);
meshView->SetArea(-2 * pitch, -2 * pitch, -0.02,
                   2 * pitch,  2 * pitch,  0.02);
// x-z projection
meshView->SetPlane(0, -1, 0, 0, 0, 0);
meshView->SetFillMesh(true);
// Set the color of the kapton.
meshView->SetColor(2, kYellow + 3);
meshView->EnableAxes();
meshView->SetViewDrift(&driftView);
meshView->Plot();
```

2.6.3. Silicon sensor

In this example, we will simulate the signal in a 100 µm thick planar silicon sensor due to the passage of a charged particle. We will adopt a coordinate system where the back side of the sensor is at $y = 0$ and the front side (*i.e.* the strip or pixel side) is at $y = 100\,\mu\text{m}$.

Transport properties

We start by creating a `MediumSilicon` object, which provides the transport parameters (*e.g.* drift velocity and diffusion coefficients) of electrons and holes as function of the electric field (and, in general, also the magnetic field, but we will assume that it is zero in this example).

```
MediumSilicon si;
// Set the temperature [K].
si.SetTemperature(293.15);
```

Unless the user overrides the default behaviour (by providing a table of velocities at different electric fields), `MediumSilicon` calculates the drift velocities according to analytic parameterizations. A description of the mobility models is given in Sec. 3.3.1. In this example, we will use the default parameterizations, which correspond to the default models in Sentaurus Device [47]. The diffusion coefficients are calculated according to the Einstein relation.

Electric field

The active volume in our example is a box with a length of $d = 100\,\mu\text{m}$ along y, centred at $y = 50\,\mu\text{m}$, and made of silicon. For accurate calculations of the electric field in a segmented silicon sensor, one normally uses TCAD device simulation programs such as Synopsys Sentaurus Device [47]. In the present example, we will follow a simplified approach and approximate the electric field by that of an overdepleted pad sensor. In that case, the x and z components of the electric field vanish, and the y component varies linearly between

$$E_y = \frac{V_{\text{bias}} - V_{\text{dep}}}{d}$$

at the back side of the sensor ($y = 0$) and

$$E_y = \frac{V_{\text{bias}} + V_{\text{dep}}}{d}$$

at the front side of the sensor ($y = d$), where V_{dep} is the depletion voltage of the sensor and V_{bias} is the applied bias voltage. In this example, we will use $V_{\text{dep}} = -20\,\text{V}$ and $V_{\text{bias}} = -50\,\text{V}$.

In order to use this expression for the electric field in our simulation, we write a lambda expression

```
// Depletion voltage [V]
constexpr double vdep = -20.;
auto eLinear = [d,vbias,vdep](const double /*x*/, const double y,
                              const double /*z*/,
                              double& ex, double& ey, double& ez) {
  ex = ez = 0.;
  ey = (vbias - vdep) / d + 2 * y * vdep / (d * d);
};
```

and set up a `ComponentUser` object which delegates the calculation of the electric field to this function.

```
ComponentUser linearField;
linearField.SetArea(-2 * d, 0., -2 * d, 2 * d, d, 2 * d);
linearField.SetMedium(&si);
linearField.SetElectricField(eLinear);
```

A pointer to this `Component` is then passed to a `Sensor` which acts as an interface to the transport classes.

```
Sensor sensor;
sensor.AddComponent(&linearField);
```

Weighting field

For signal simulations, we need to know not only the actual electric field in the sensor, but also the weighting field of the electrode for which we want to calculate the induced current (Chapter 7).

In this example, we will use an analytic expression for the weighting field of a strip, as implemented in the class `ComponentAnalyticField`. We thus create a `ComponentAnalyticField` object, define the equipotential planes ($y = 0$ and $y = d$) and set the voltages at these planes to ground and $V = V_{\text{bias}}$. We will not use this class to calculate the "real" electric field in the sensor though, so the voltage settings don't actually matter for our purposes.

```
ComponentAnalyticField wField;
wField.SetMedium(&si);
wField.AddPlaneY(0, vbias, "back");
wField.AddPlaneY(d, 0, "front");
```

We now define a strip (55 µm width, centred at $x = 0$) on the front side of the sensor.

```
constexpr double pitch = 55.e-4;
constexpr double halfpitch = 0.5 * pitch;
wField.AddStripOnPlaneY('z', d, -halfpitch, halfpitch, "strip");
```

The last argument of the above function is a label, which we will use later to identify the signal induced on the strip. Similarly we could have set up the weighting field of a pixel electrode.

```
wField.AddPixelOnPlaneY(d, -halfpitch, halfpitch, -halfpitch, halfpitch, "pixel");
```

Finally, we need to instruct the `Sensor` to use the strip weighting field we just prepared for computing the induced signal

```
// Request signal calculation for the electrode named "strip",
// using the weighting field provided by the Component object wField.
sensor.AddElectrode(&wField, "strip");
```

and we need to set the granularity with which we want to record the signal (in our example: 1000 bins between 0 and 10 ns).

```
// Set the time bins.
const unsigned int nTimeBins = 1000;
const double tmin =  0.;
const double tmax = 10.;
const double tstep = (tmax - tmin) / nTimeBins;
sensor.SetTimeWindow(tmin, tstep, nTimeBins);
```

Primary ionization and charge carrier transport

We use Heed (Sec. 5.1) to simulate the electron/hole pairs produced by a 180 GeV / c charged pion traversing the sensor.

```
TrackHeed track(&sensor);
// Set the particle type and momentum [eV/c].
track.SetParticle("pion");
track.SetMomentum(180.e9);
```

For transporting the electrons and holes, we use the class `AvalancheMC`. When setting up the `AvalancheMC` object, we need to set the step size used for the drift line calculation to a reasonable value. In this example, we use steps of 1 µm. This means that at each step, the electron/hole will be propagated by 1 µm in the direction of the drift velocity at the local field, followed by a random step based on the diffusion coefficient.

```
AvalancheMC drift(&sensor);
// Set the step size [cm].
drift.SetDistanceSteps(1.e-4);
```

We are now ready to run the simulation. In the snippet below, we simulate a perpendicularly incident charged particle track passing through the centre of the strip ($x = 0$), loop over the electron/hole pairs produced by the particle, and simulate a drift line for each electron and hole.

```
double x0 = 0., y0 = 0., z0 = 0., t0 = 0.;
double dx = 0., dy = 1., dz = 0.;
track.NewTrack(x0, y0, z0, t0, dx, dy, dz);
// Retrieve the clusters along the track.
for (const auto& cluster : track.GetClusters()) {
  // Loop over the electrons in the cluster.
  for (const auto& electron : cluster.electrons) {
    // Simulate the electron and hole drift lines.
    drift.DriftElectron(electron.x, electron.y, electron.z, electron.t);
    drift.DriftHole(electron.x, electron.y, electron.z, electron.t);
  }
}
```

To check whether the results are sensible, it can be instructive to visualize the drift lines using the class `ViewDrift`.

```
ViewDrift driftView;
driftView.SetArea(-0.5 * d, 0, -0.5 * d, 0.5 * d, d, 0.5 * d);
track.EnablePlotting(&driftView);
drift.EnablePlotting(&driftView);
```

With the plotting option switched on, `AvalancheMC` will pass the coordinates of all drift line points to a `ViewDrift` object. After having simulated all drift lines from a track, we can create a plot using

```
constexpr bool twod = true;
driftView.Plot(twod);
```

Plotting the drift lines can slow down the execution time quite a bit, so it is advisable to switch it off when simulating a large number of tracks.

Retrieving the signal

After having simulated the charge carrier drift lines, we can plot the induced current.

```
TCanvas* cSignal = new TCanvas("cSignal", "", 600, 600);
sensor.PlotSignal("strip", cSignal);
```

To post-process the induced current pulse, one can convolute it with a transfer function that describes the response of the front-end electronics.

Often it can also be useful to save the signal to a file. An example for doing so is given in the code snippet below.

```
std::ofstream outfile;
outfile.open("signal.txt", std::ios::out);
for (unsigned int i = 0; i < nTimeBins; ++i) {
  const double t = (i + 0.5) * tstep;
  const double f = sensor.GetSignal(label, i);
  const double fe = sensor.GetElectronSignal(label, i);
  const double fh = sensor.GetIonSignal(label, i);
  outfile << t << "␣␣" << f << "␣␣" << fe << "␣␣" << fh << "\n";
}
```

3. Media

Media are derived from the abstract base class `Medium`.

The name (identifier) of a medium can be read using the function

```
const std::string& GetName() const;
```

For compound media (*e. g.* gas mixtures), the identifiers and fractions of the constituents are available via

```
unsigned int GetNumberOfComponents();
void GetComponent(const unsigned int i, std::string& label, double& f);
```

3.1. Transport parameters

`Medium` classes provide the following functions for calculating macroscopic electron transport parameters as a function of the electric and magnetic field:

```
bool ElectronVelocity(const double ex, const double ey, const double ez,
                      const double bx, const double by, const double bz,
                      double& vx, double& vy, double& vz);
bool ElectronDiffusion(const double ex, const double ey, const double ez,
                       const double bx, const double by, const double bz,
                       double& dl, double& dt);
bool ElectronTownsend(const double ex, const double ey, const double ez,
                      const double bx, const double by, const double bz,
                      double& alpha);
bool ElectronAttachment(const double ex, const double ey, const double ez,
                        const double bx, const double by, const double bz,
                        double& eta);
```

ex, ey, ez electric field (in V / cm)

bx, by, bz magnetic field (in T)

vx, vy, vz drift velocity (in cm / ns)

dl, dt longitudinal and transverse diffusion coefficients (in $\sqrt{\text{cm}}$)

alpha Townsend coefficient (in cm^{-1})

eta attachment coefficient (in cm^{-1})

The above functions return `true` if the respective parameter is available at the requested field.

Table 3.1. Pressure scaling relations for gases.

transport parameter	scaling
drift velocity	v vs. E/p
diffusion coefficients	$\sigma\sqrt{p}$ vs. E/p
Townsend coefficient	α/p vs. E/p
attachment coefficient	η/p vs. E/p

Analogous functions are available for holes (albeit of course not meaningful for gases), and also for ions (except for the Townsend and attachment coefficients).

The components of the drift velocity are stored in a right-handed coordinate system that is aligned with the electric and magnetic field vectors. More precisely, the axes are along

- the electric field $\mathbf{E}$,
- the component of the magnetic field $\mathbf{B}$ transverse to $\mathbf{E}$, $\mathbf{B}_t = (\mathbf{E} \times \mathbf{B}) \times \mathbf{E}$,
- $\mathbf{E} \times \mathbf{B}$.

The longitudinal diffusion is measured along $\mathbf{E}$. The transverse diffusion is the average of the diffusion coefficients along the two remaining axes.

3.1.1. Transport parameter tables

The transport parameters can either be stored in a one-dimensional table (as a function of the electric field only) or in a three-dimensional table (as a function of $\mathbf{E}$, $\mathbf{B}$, and the angle θ between $\mathbf{E}$ and $\mathbf{B}$). If only a one-dimensional table is present and the drift velocity at $B \neq 0$ is requested, the Langevin equation [8]

$$\mathbf{v} = \frac{\mu}{1 + \mu^2 B^2} \left(\mathbf{E} + \mu \mathbf{E} \times \mathbf{B} + \mu^2 \mathbf{B} \left(\mathbf{E} \cdot \mathbf{B}\right)\right), \qquad \mu = v/E.$$

is used.

All transport parameters share the same grid of electric fields, magnetic fields, and angles. By default, the field and angular ranges are

- 20 electric field points between 100 V / cm and 100 kV / cm, with logarithmic spacing
- $\mathbf{B} = 0$, $\theta = \pi/2$

For specifying the field grid, two functions are available:

```
void SetFieldGrid(double emin, double emax, const size_t ne, bool logE,
                  double bmin, double bmax, const size_t nb,
                  double amin, double amax, const size_t na);
void SetFieldGrid(const std::vector<double>& efields,
                  const std::vector<double>& bfields,
                  const std::vector<double>& angles);
```

emin, emax min. and max. value of the electric field range to be covered by the tables

ne number of electric field grid points

logE flag specifying whether the E-field grid points should be evenly spaced (`false`), or logarithmically spaced (`true`)

bmin, bmax, ne magnetic field range and number of values

amin, amax, na angular range and number of angles

efields, bfields, angles lists of E, B, and θ (in ascending order)

Electric fields have to be supplied in V / cm, magnetic fields in Tesla, and angles in radian.

The electron drift velocity components at a specific point in the table can be set and retrieved using

```
bool SetElectronVelocityE(const size_t ie, const size_t ib,
                          const size_t ia, const double v);
bool SetElectronVelocityB(const size_t ie, const size_t ib,
                          const size_t ia, const double v);
bool SetElectronVelocityExB(const size_t ie, const size_t ib,
                            const size_t ia, const double v);
bool GetElectronVelocityE(const size_t ie, const size_t ib,
                          const size_t ia, double& v);
bool GetElectronVelocityB(const size_t ie, const size_t ib,
                          const size_t ia, double& v);
bool GetElectronVelocityExB(const size_t ie, const size_t ib,
                            const size_t ia, double& v);
```

ie index in the list of electric fields,

ib index in the list of magnetic fields,

ia index in the list of angles,

v velocity

Analogous functions are available for the other transport parameters. For the Townsend coefficient α and the attachment coefficient η, the logarithms $\ln\alpha, \ln\eta$ and not the actual values are stored in the tables.

The gas tables are interpolated using Newton polynomials. The order of the interpolation polynomials can be set by means of

```
void SetInterpolationMethodVelocity(const unsigned int intrp);
void SetInterpolationMethodDiffusion(const unsigned int intrp);
void SetInterpolationMethodTownsend(const unsigned int intrp);
void SetInterpolationMethodAttachment(const unsigned int intrp);
void SetInterpolationMethodIonMobility(const unsigned int intrp);
void SetInterpolationMethodIonDissociation(const unsigned int intrp);
```

intrp order of the interpolation polynomial

The interpolation order must be between 1 and the smallest of the two numbers: 10 and number of table entries - 1. Orders larger than 2 are not recommended.

The method for extrapolating to E values smaller and larger than those present in the table can be set using

```
void SetExtrapolationMethodVelocity(const std::string extrLow,
                                    const std::string extrHigh);
```

extrLow, extrHigh extrapolation method to be used ("constant", "exponential", or "linear")

Similar functions are available for the other transport parameters. The extrapolation method set using this function has no effect on extrapolation in three-dimensional tables. In such tables, polynomial extrapolation is performed with the same order as for the interpolation.

The default settings are

- quadratic interpolation,
- constant extrapolation towards low values,
- linear extrapolation towards high values.

3.1.2. Visualization

For plotting transport parameters as function of the electric field, the member functions

```
void PlotVelocity(const std::string& carriers, TPad* pad);
void PlotDiffusion(const std::string& carriers, TPad* pad);
void PlotTownsend(const std::string& carriers, TPad* pad);
void PlotAttachment(const std::string& carriers, TPad* pad);
void PlotAlphaEta(const std::string& carriers, TPad* pad);
```

of the class `Medium` can be used, where the option string `carriers` indicates the charge carriers (`"e"`: electrons, `"i"`: ions, `"h"`: holes) for which to plot the requested transport parameters. The following example plots the drift velocity of electrons and holes in silicon as well as the Townsend and attachment coefficients.

```
MediumSilicon si;
TCanvas* c1 = new TCanvas("c1", "", 600, 600);
si.PlotVelocity("eh", c1);
TCanvas* c2 = new TCanvas("c2", "", 600, 600);
si.PlotAlphaEta("eh", c2);
```

Internally, the above functions use the class `ViewMedium`. Using `ViewMedium` directly, another way to produce the first plot in the above example would be:

```
MediumSilicon si;
ViewMedium view(&si);
view.PlotVelocity("eh", 'e');
```

The second argument (of type `char`) of the function `ViewMedium::PlotVelocity` indicates the quantity to plot on the x-axis. Valid options are `'e'` (electric field), `'b'` (magnetic field) and `'a'` (angle between electric and magnetic field).

By default, `ViewMedium` will try to determine the range of the x axis based on the grid of electric fields, magnetic fields, and angles, and the range of the y axis based on the minima and maxima of the function to be plotted. This feature can be switched on or off using the functions

```
void EnableAutoRangeX(const bool on = true);
void EnableAutoRangeY(const bool on = true);
```

The ranges can be set explicitly using

```
void SetRangeE(const double emin, const double emax, const bool logscale);
void SetRangeB(const double bmin, const double bmax, const bool logscale);
void SetRangeA(const double amin, const double amax, const bool logscale);
void SetRangeY(const double ymin, const double ymax, const bool logscale);
```

When making a plot as function of the electric field, the magnetic field and angle can be set using

```
void SetMagneticField(const double bfield);
void SetAngle(const double angle);
```

Similarly, the electric field to be used when making a plot as function of magnetic field or angle can be set using

```
void SetElectricField(const double efield);
```

The (ROOT) colours with which to draw the graphs/curves and the labels to be used to identify them can be customized using

```
void SetLabels(const std::vector<std::string>& labels);
void SetColours(const std::vector<short>& cols);
```

If the function

```
void EnableExport(const std::string& txtfile);
```

txtfile name of the output text file

is called before one of the `Plot..` functions, the plot data points will be saved to a text file.

3.2. Gases

There are currently two classes implemented that can be used for the description of gaseous media: `MediumGas` and its daughter class `MediumMagboltz`. While `MediumGas` deals only with the interpolation of gas tables and the import of gas files, `MediumMagboltz` – owing to an interface to the Magboltz program [7] – can be used for the calculation of transport parameters. In addition, the latter class provides access to the electron-molecule scattering cross-sections used in Magboltz and is thus suitable for microscopic tracking (chapter 6).

The composition of the gas mixture is specified using

```
bool SetComposition(const std::string& gas1, const double f1 = 1.,
                    const std::string& gas2 = "", const double f2 = 0.,
                    const std::string& gas3 = "", const double f3 = 0.,
```

```
                const std::string& gas4 = "", const double f4 = 0.,
                const std::string& gas5 = "", const double f5 = 0.,
                const std::string& gas6 = "", const double f6 = 0.);
```

gas1, ..., gas6 identifier of the molecule/atom

f1, ..., f6 number fraction of the respective molecule/atom

A valid gas mixture comprises at least one and at most six different species.

The function

```
void PrintGases();
```

prints out a list of the available gases and their identifiers (see also Table B.1).

The fractions have to be strictly positive and may add up to any non-zero value; internally they will be normalized to one.

The gas density is specified in terms of pressure (in Torr) and temperature (in K)

```
void SetPressure(const double p);
void SetTemperature(const double t);
```

and calculated using the ideal gas law.

In the following example the gas mixture is set to Ar/CH_4 (80/20) at atmospheric pressure and 20° C.

```
MediumMagboltz gas;
// Set the composition
gas.SetComposition("ar", 80., "ch4", 20.);
gas.SetTemperature(293.15);
gas.SetPressure(760.);
```

The gas composition can also be specified in the constructor of `MediumMagboltz`.

```
MediumMagboltz gas("ar", 80., "ch4", 20.);
gas.SetTemperature(293.15);
gas.SetPressure(760.);
```

The function

```
void PrintGas();
```

prints information about the present transport parameter tables and cross-section terms (if available).

3.2.1. W values and Fano factors

`MediumGas` has default settings for the W value (average energy to create an electron-ion pair) and the Fano factor of each gas available in Magboltz. Where available, measurements of the W

value reported in Refs. [4, 33, 29] and measurements of the Fano factor reported in Refs. [3, 45, 29] were used.

For gases for which there no experimental data could be found in the literature, the W values were calculated based on the set of cross-sections implemented in Magboltz and the Fano factor was calculated using the empirical relation [18]

$$F = 0.188\frac{W}{I} - 0.15$$

where I is the ionisation potential of the gas.

3.2.2. Ion transport

The `Data` directory of the project includes a number of text files (*e. g.* `IonMobility_Ar+_Ar.txt` for Ar^+ ions in argon) with ion mobility data. When building the project, these files are copied to the folder `share/Garfield/Data/` of the install directory. More precisely, the files contain a table of reduced electric fields E/N and reduced mobilities. The reduced electric fields are given in units of Td[1] (Townsend) and the mobility values in $cm^2\,V^{-1}\,s^{-1}$. Mobility files for positive ions can be imported using

```
bool MediumGas::LoadIonMobility(const std::string& filename,
                                const bool quiet = false);
```

filename path and filename of the mobility file

Extensive compilations of ion mobilities and diffusion coefficients can be found in Refs. [12, 13, 14, 49].

Analogously, mobility data for negative ions can be imported using

```
bool LoadNegativeIonMobility(const std::string& filename,
                             const bool quiet = false);
```

3.2.3. Magboltz

Magboltz, written by Steve Biagi, is a program for the calculation of electron transport properties in gas mixtures using semi-classical Monte Carlo simulation [7]. It includes a database of electron-atom/molecule cross-sections for a large number of detection gases.

The function

```
void GenerateGasTable(const int numCollisions, const bool verbose);
```

runs Magboltz for all values of **E**, **B**, and θ included in the current field grid.

In addition to the transport parameters, this function also retrieves the rates calculated by Magboltz for each excitation and ionisation level, and stores them in the gas table. These can be used later to adjust the Townsend coefficient based on the Penning transfer probabilities set by the user.

[1] $1\,\text{Td} = 10^{-17}\,\text{V}\,\text{cm}^2$

By default, the max. energy of the cross-section table is chosen automatically by Magboltz. This feature can be enabled or disabled using

```
void EnableAutoEnergyLimit(const bool on = true);
```

If it is switched off, the program uses the upper energy limit set using

```
bool SetMaxElectronEnergy(const double e);
```

For inelastic gases, setting `nColl` $= 2 \ldots 5$ should give an accuracy of about 1%. An accuracy better than 0.5% can be achieved by `nColl` > 10. For pure elastic gases such as Ar, `nColl` should be at least 10.

Recent versions of Magboltz allow the thermal motion of the gas atoms/molecules to be taken into account in the simulation. This feature can be enabled or disabled using

```
void EnableThermalMotion(const bool on);
```

By default the option is switched off, *i. e.* the gas is assumed to be at 0 K.

Electron transport parameter tables can be saved to file and read from file by means of

```
bool WriteGasFile(const std::string& filename);
bool LoadGasFile(const std::string& filename);
```

The format of the gas file used in Garfield++ is compatible with the one used in Garfield 9.

Gas files for the same gas composition and the same temperature and pressure can be merged using

```
bool MergeGasFile(const std::string& filename, const bool replaceOld);
```

filename name of the gas file to be loaded and merged with the present gas table,

replaceOld flag indicating whether new (`replaceOld = true`) or existing values should be used in case of overlaps between the two tables.

Suppose we have two gas files for Ar/CO_2, one for a B field of 1 T and one for $B = 2$ T. We can combine the two tables with the following snippet of code.

```
MediumMagboltz gas;
gas.LoadGasFile("ar_co2_1T.gas");
gas.MergeGasFile("ar_co2_2T.gas");
// Save the merged table.
gas.WriteGasFile("ar_co2_merged.gas");
```

Scattering rates

As a prerequisite for "microscopic tracking" a table of the electron scattering rates (based on the electron-atom/molecule cross-sections included in the Magboltz database) for the current gas mixture and density needs to be prepared. This can be done using the function

Table 3.2. Classification of electron collision processes.

collision type	index
elastic collision	0
ionisation	1
attachment	2
inelastic collision	3
excitation	4
superelastic collision	5
virtual ("null") collision	6

```
bool Initialise(const bool verbose);
```

If the flag `verbose` is set to `true`, some information (such as gas properties, and collision rates at selected energies) is printed during the initialisation.

If

```
void EnableCrossSectionOutput();
```

is called prior to `Initialise`, a table of the cross-sections (as retrieved from Magboltz) is written to a file `cs.txt` in the current working directory.

By default, the table of scattering rates extends from 0 to 40 eV. The max. energy to be included in the table can be set using

```
SetMaxElectronEnergy(const double e);
```

e max. electron energy (in eV).

Up to an upper limit of 400 eV, equidistant energy steps are used. If the max. energy exceeds this value, logarithmically spaced energy steps are used for the high-energy part (> 400 eV) of the cross-section table.

The parameters of the cross-section terms in the present gas mixture can be retrieved via

```
int GetNumberOfLevels();
bool GetLevel(const unsigned int i, int& ngas, int& type, std::string& descr, double& e);
```

i index of the cross-section term

ngas index of the gas in the mixture

type classification of the cross-section term (see Table 3.2)

descr description of the cross-section term (from Magboltz)

e energy loss

It is sometimes useful to know the frequency with which individual levels are excited in an avalanche (or along a drift line). For this purpose, `MediumMagboltz` keeps track of the number

of times the individual levels are sampled in `ElectronCollision`. These counters are accessible through the functions

```
unsigned int GetNumberOfElectronCollisions();
unsigned int GetNumberOfElectronCollisions(int& nElastic, int& nIonising,
                                           int& nAttachment, int& nInelastic,
                                           int& nExcitation, int& nSuperelastic);
unsigned int GetNumberOfElectronCollisions(const unsigned int level);
```

The first function returns total number of electron collisions since the last reset. The second function additionally provides the number of collisions of each cross-section category (elastic, ionising etc.). The third function returns the number of collisions for a specific cross-section term. The counters can be reset using

```
void ResetCollisionCounters();
```

Excitation transfer

Penning transfer can be taken into account in terms of a transfer efficiency r_i, *i. e.* the probability for an excited level i with an excitation energy ϵ_i exceeding the ionisation potential ϵ_{ion} of the mixture to be "converted" to an ionisation. The simulation of Penning transfer can be switched on/off using

```
void EnablePenningTransfer();
void EnablePenningTransfer(const double r, const double lambda,
                           std::string gasname);
void EnablePenningTransfer(const double r, const double lambda);
```

r value of the transfer efficiency

lambda distance characterizing the spatial extent of Penning transfers; except for special studies, this number should be set to zero

gasname name of the gas the excitation levels of which are to be assigned the specified transfer efficiency

The first function, which takes no arguments, calculates the Penning transfer probability for the current gas mixture using pre-implemented parameterisations taken from literature [39, 37, 38, 36], if available.

The second function sets the Penning transfer probability for a specific gas component in the mixture.

The third function (without the `gasname` parameter) activates Penning transfer globally for all gases in the mixture. Note that r is an average transfer efficiency, it is assumed to be the same for all energetically eligible levels ($\epsilon_i > \epsilon_{\text{ion}}$).

Penning transfer can be switched off, globally or for a specific component, using

```
void DisablePenningTransfer();
void DisablePenningTransfer(std::string gasname);
```

If the gas table includes excitation and ionisation rates as function of the electric and magnetic fields, the Townsend coefficient is updated accordingly when calling `EnablePenningTransfer` (or `DisablePenningTransfer`). More precisely, the adjusted Townsend coefficient is given by

$$\alpha = \alpha_0 \frac{\sum_i r_{\text{exc},i} + \sum_i r_{\text{ion},i}}{\sum_i r_{\text{ion},i}},$$

where α_0 is the Townsend coefficient calculated without Penning transfers, $r_{\text{exc},i}$ is the rate of an excited level i with an excitation energy above the ionisation potential of the gas mixture, and $r_{\text{ion},i}$ is the rate of an ionization level i.

Table 3.3. Lattice mobility parameter values.

	electrons		holes	
	μ_L [10^{-6} cm^2 V^{-1} ns^{-1}]	β	μ_L [10^{-6} cm^2 V^{-1} ns^{-1}]	β
Sentaurus [19]	1.417	−2.5	0.4705	−2.5
Minimos [41]	1.43	−2.0	0.46	−2.18
Reggiani [27]	1.32	−2.0	0.46	−2.2

3.3. Semiconductors

3.3.1. Silicon

Like for all `Medium` classes, users have the possibility to specify the transport parameters in tabulated form as function of electric field, magnetic field, and angle. If no such tables have been entered, `MediumSilicon` calculates the electron and hole transport parameters based on empirical parameterizations (as used, for instance, in device simulation programs). Several mobility models are implemented. For the mobility μ_0 at low electric fields, the following options are available:

- Using

```
void SetLowFieldMobility(const double mue, const double mh);
```

  mue electron mobility (in cm^2/(V ns))

  muh hole mobility (in cm^2/(V ns))

  the values of low-field electron and hole mobilities can be specified explicitly by the user.

- The following functions select the model to be used for the mobility due to phonon scattering:

```
void SetLatticeMobilityModelMinimos();
void SetLatticeMobilityModelSentaurus();
void SetLatticeMobilityModelReggiani();
```

  In all cases, the dependence of the lattice mobility μ_L on the temperature T is described by

$$\mu_L(T) = \mu_L(T_0)\left(\frac{T}{T_0}\right)^{\beta}, \qquad T_0 = 300 \text{ K}. \tag{3.1}$$

  The values of the parameters $\mu_L(T_0)$ and β used in the different models are shown in Table 3.3. By default, the "Sentaurus" model is activated.

- The parameterization to be used for modelling the impact of doping on the mobility is specified using

```
void SetDopingMobilityModelMinimos();
void SetDopingMobilityModelMasetti();
```

The first function activates the model used in Minimos 6.1 (see Ref. [41]). Using the second function the model described in Ref. [22] is activated (default setting).

For modelling the velocity as function of the electric field, the following options are available:

- The method for calculating the high-field saturation velocities can be set using

```
void SetSaturationVelocity(const double vsate, const double vsath);
void SetSaturationVelocityModelMinimos();
void SetSaturationVelocityModelCanali();
void SetSaturationVelocityModelReggiani();
```

The first function sets user-defined saturation velocities (in cm/ns) for electrons and holes. The other functions activate different parameterizations for the saturation velocity as function of temperature. In the Canali model [10], which is activated by default,

$$\begin{aligned} v^e_{\rm sat} &= 0.0107 \left(\frac{T_0}{T}\right)^{0.87} \text{ cm/ns}, \\ v^h_{\rm sat} &= 0.00837 \left(\frac{T_0}{T}\right)^{0.52} \text{ cm/ns}, \end{aligned}$$

where $T_0 = 300$ K. The expressions for the other two implemented models can be found in Refs. [27, 31].

- The parameterization of the mobility as function of the electric field to be used can be selected using

```
void SetHighFieldMobilityModelMinimos();
void SetHighFieldMobilityModelCanali();
void SetHighFieldMobilityModelReggiani();
void SetHighFieldMobilityModelConstant();
```

The last function requests a constant mobility (*i. e.* linear dependence of the velocity on the electric field). The models activated by the other functions used the following expressions

$$\mu^e(E) = \frac{2\mu^e_0}{1+\sqrt{1+\left(\frac{2\mu^e_0 E}{v^e_{\rm sat}}\right)^2}}, \qquad \mu^h(E) = \frac{\mu^h_0}{1+\frac{\mu_0}{v^h_{\rm sat}}}, \qquad \text{(Minimos)}$$

$$\mu^{e,h}(E) = \frac{\mu^{e,h}_0}{\left(1+\left(\frac{\mu^{e,h}_0 E}{v_{{\rm sat}^{e,h}}}\right)^{\beta^{e,h}}\right)^{\frac{1}{\beta^{e,h}}}}, \qquad \text{(Canali [10])}$$

$$\mu^e(E) = \frac{\mu^e_0}{\left(1+\left(\frac{\mu^e_0 E}{v^e_{\rm sat}}\right)^{3/2}\right)^{2/3}}, \qquad \mu^h(E) = \frac{\mu^h_0}{\left(1+\left(\frac{\mu^h_0 E}{v^h_{\rm sat}}\right)^{2}\right)^{1/2}}, \qquad \text{(Reggiani [27])}$$

By default, the Canali model is used.

For the impact ionization coefficient, the user has currently the choice between the models of Grant [17], van Overstraeten and de Man [28], Okuto and Crowell [26], and Massey [23].

```
void SetImpactIonisationModelGrant();
void SetImpactIonisationModelVanOverstraetenDeMan();
```

```
void SetImpactIonisationModelMassey();
void SetImpactIonisationModelOkutoCrowell();
```

By default, the model by van Overstraeten and de Man is used.

3.3.2. Gallium arsenide

By default, `MediumGaAs` uses Eq. (3.1) for calculating the low-field lattice mobility, with

$$\mu_L\left(T = 300\,\mathrm{K}\right) = 8 \times 10^{-6}\,\mathrm{cm}^2\,\mathrm{V}^{-1}\,\mathrm{ns}^{-1}, \qquad \beta = -1$$

for electrons and

$$\mu_L\left(T = 300\,\mathrm{K}\right) = 0.4 \times 10^{-6}\,\mathrm{cm}^2\,\mathrm{V}^{-1}\,\mathrm{ns}^{-1}, \beta = -2.1$$

for holes. Alternatively, the low-field mobilities can be set explicitly using

```
SetLowFieldMobility(const double mue, const double muh);
```

For the dependence of the mobility on the electric field E, the parameterizations [5]

$$\mu^e\left(E\right) = \frac{\mu_0^e + \frac{v_{\mathrm{sat}}}{E}\left(\frac{E}{E_c}\right)^4}{1 + \left(\frac{E}{E_c}\right)^4}, \qquad \mu^h\left(E\right) = \frac{\mu_0^h + \frac{v_{\mathrm{sat}}}{E_c}}{1 + \frac{E}{E_c}}, \qquad E_c = 4000\,\mathrm{V}\,/\,\mathrm{cm}$$

are used, and the saturation velocity is calculated using

$$v_{\mathrm{sat}}^{e,h} = 1.13 \times 10^{-2} - 3.6 \times 10^{-3}\left(\frac{T}{T_0}\right)\,\mathrm{cm}\,/\,\mathrm{ns}.$$

The Townsend (impact ionization) coefficient as a function of the electric field E is calculated using

$$\alpha^e = a\exp\left(-\left(\frac{b}{E}\right)^{1.75}\right), \qquad \alpha^h = a\exp\left(-\left(\frac{b}{E}\right)^{1.82}\right).$$

3.3.3. Diamond

Unless the low-field mobility values are set explicitly by the user, `MediumDiamond` calculates them using

$$\mu_0\left(T\right) = \left(\frac{T}{T_0}\right)^{-1.5}\mu_0\left(T_0\right),$$

where $T_0 = 300\,\mathrm{K}$ and $\mu_0\left(T_0\right) = 4.551 \times 10^{-6}\,\mathrm{cm}^2\,\mathrm{V}^{-1}\,\mathrm{ns}^{-1}$ for electrons and $\mu_0\left(T_0\right) = 2.750 \times 10^{-6}\,\mathrm{cm}^2\,\mathrm{V}^{-1}\,\mathrm{ns}^{-1}$ for holes [30]. The mobility as function of the electric field E is calculated using

$$\mu\left(E\right) = \frac{\mu_0}{1 + \frac{\mu_0 E}{v_{\mathrm{sat}}}}$$

with default values of $v_{\mathrm{sat}} = 2.6 \times 10^{-2}\,\mathrm{cm}\,/\,\mathrm{ns}$ for electrons and $v_{\mathrm{sat}} = 1.6 \times 10^{-2}\,\mathrm{cm}\,/\,\mathrm{ns}$ for holes. These default values can be overriden using

```
void SetLowFieldMobility(const double mue, const double muh);
void SetSaturationVelocity(const double vsate, const double vsath);
```

4. Components

The calculation of electric fields is done by classes derived from the abstract base class `Component`. The key functions, used internally in the classes for simulating charge transport, are

```
void ElectricField(const double x, const double y, const double z,
                   double& ex, double& ey, double& ez,
                   Medium*& m, int& status);
void ElectricField(const double x, const double y, const double z,
                   double& ex, double& ey, double& ez, double& v);
Medium* GetMedium(const double& x, const double& y, const double& z);
```

x, y, z position where the electric field (medium) is to be determined

ex, ey, ez, v electric field and potential at the given position

m pointer to the medium at the given position; if there is no medium at this location, a null pointer is returned

status status flag indicating where the point is located (see Table 4.1)

To retrieve only the electrostatic potential, the function

```
double ElectricPotential(const double x, const double y, const double z);
```

can be used, and the function

```
std::array<double, 3> ElectricField(const double x, const double y, const double z);
```

returns the three components of the electric field.

4.1. Defining the geometry

As mentioned above, the purpose of `Component` classes is to provide, for a given location, the electric (and magnetic) field and a pointer to the `Medium` object (if available). For the latter

Table 4.1. Status flags for electric fields.

value	meaning
0	inside a drift medium
> 0	inside a wire
-1 ...-4	on the side of a plane where no wires are
-5	inside the mesh, but not in a drift medium
-6	outside the mesh

task, it is obviously necessary to specify the geometry of the device. In case of field maps, the geometry is already defined in the field solver. It is, therefore, sufficient to associate the materials of the field map with the corresponding `Medium` objects.

For analytic fields, the geometry is, in general, given by the cell layout.

For other components (*e. g.* user-parameterized fields), the geometry has to be defined separately.

Simple structures can be described by the native geometry (`GeometrySimple`), which has only a very restricted repertoire of shapes (solids). At present, the available solids are

- `SolidBox`,
- `SolidTube`,
- `SolidHole`,
- `SolidRidge`,
- `SolidExtrusion`,
- `SolidWire`, and
- `SolidSphere`.

A description of these classes is given in Appendix C.

As an example, we consider a gas-filled tube with a diameter of 1 cm and a length of 20 cm (along the z-axis), centred at the origin:

```
// Create the medium.
MediumMagboltz gas;
// Create the geometry.
GeometrySimple geo;
// Dimensions of the tube
double rMax = 0.5, halfLength = 10.;
SolidTube tube(0., 0., 0., rMax, halfLength);
// Add the solid to the geometry, together with the gas inside.
geo.AddSolid(&tube, &gas);
```

Solids may overlap. When the geometry is scanned (triggered, for instance, by calling `GetMedium`), the first medium found is returned. The sequence of the scan is reversed with respect to the assembly of the geometry. Hence, the last medium added to the geometry is considered the innermost.

For more complex structures, the class `GeometryRoot` can be used which provides an interface to the ROOT geometry (`TGeo`). Using `GeometryRoot`, the above example would look like this:

```
// Create the ROOT geometry.
TGeoManager* geoman = new TGeoManager("world", "geometry");
// Create the ROOT material and medium.
// For simplicity we use the predefined material "Vacuum".
TGeoMaterial* matVacuum = new TGeoMaterial("Vacuum", 0, 0, 0);
TGeoMedium*   medVacuum = new TGeoMedium("Vacuum", 1, matVacuum);
// Dimensions of the tube
double rMin = 0., rMax = 0.5, halfLength = 10.;
// In this simple case, the tube is also the top volume.
```

```
TGeoVolume* top = geoman->MakeTube("TOP", medVacuum, rMin, rMax, halfLength);
geoman->SetTopVolume(top);
geoman->CloseGeometry();
// Create the Garfield medium.
MediumMagboltz* gas = new MediumMagboltz();
// Create the Garfield geometry.
GeometryRoot* geo = new GeometryRoot();
// Pass the pointer to the TGeoManager.
geo->SetGeometry(geoman);
// Associate the ROOT medium with the Garfield medium.
geo->SetMedium("Vacuum", gas);
```

In either case (`GeometrySimple` and `GeometryRoot`), after assembly the geometry is passed to the `Component` as a pointer:

```
void SetGeometry(Geometry* geo);
```

4.1.1. Visualizing the geometry

Geometries described by `GeometrySimple` can be viewed using the class `ViewGeometry`.

```
MediumMagboltz gas;
// Create and setup the geometry.
GeometrySimple geo;
const double r1 = 0.4;
const double r2 = 0.6;
SolidHole hole(0, 0, 0, r1, r2, 1, 1, 1);
hole.SetSectors(4);
geo.AddSolid(&hole, &gas);

// Create a viewer.
ViewGeometry geoView(&geo);
geoView.Plot3d();
```

The snippet above will produce a three-dimensional impression of the geometry. The function `ViewGeometry::Plot2d()` draws a cut through the solids at the current viewing plane, *e. g.* using again the geometry defined above,

```
// ...
ViewGeometry geoView;
geoView.SetPlaneXZ();
geoView.Plot2d();
```

The layout of an arrangement of wires, planes and tubes defined in `ComponentAnalyticField` can be visualized using the member function

```
void PlotCell(TPad* pad);
```

of `ComponentAnalyticField`:

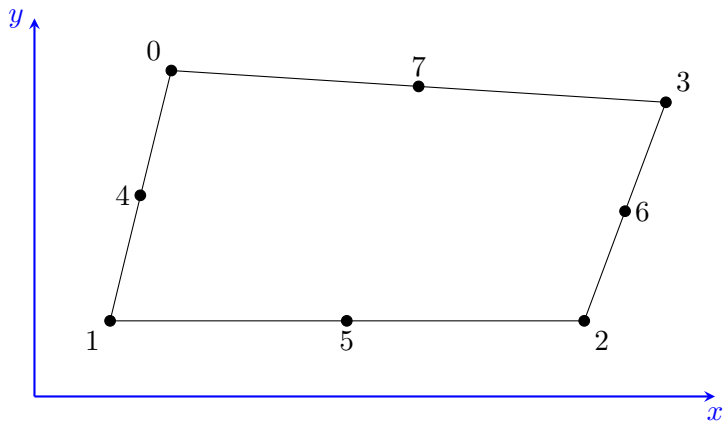

Figure 4.1. Eight-node quadrilateral element used in 2D Ansys field maps.

```
// Create and set up the component.
ComponentAnalyticField cmp;
// ...
TCanvas* canvas = new TCanvas("c", "", 600, 600);
cmp.PlotCell(canvas);
```

This function internally uses the class `ViewCell`. Instead of calling `PlotCell` one can also use `ViewCell` directly:

```
// Create and set up the component.
ComponentAnalyticField cmp;
...
// Create a viewer.
ViewCell view(&cmp);
// Make a two-dimensional plot of the cell layout.
view.Plot2d();
```

By default, `ViewCell` will use marker symbols to indicate the position of the wires (instead of drawing the wires to scale). This behaviour can be switched on or off using the function

```
ViewCell::EnableWireMarkers(const bool on);
```

The function `ViewCell::Plot3d()` paints a three-dimensional view of the cell layout.

4.2. Field maps

4.2.1. Ansys

The interpolation of FEM field maps created with the program Ansys [2] is dealt with by the classes `ComponentAnsys121` and `ComponentAnsys123`. The class names refer to the type of mesh element used by Ansys:

- `ComponentAnsys121` reads two-dimensional field maps with eight-node curved quadrilaterals (Fig. 4.1), known as “plane121” in Ansys).

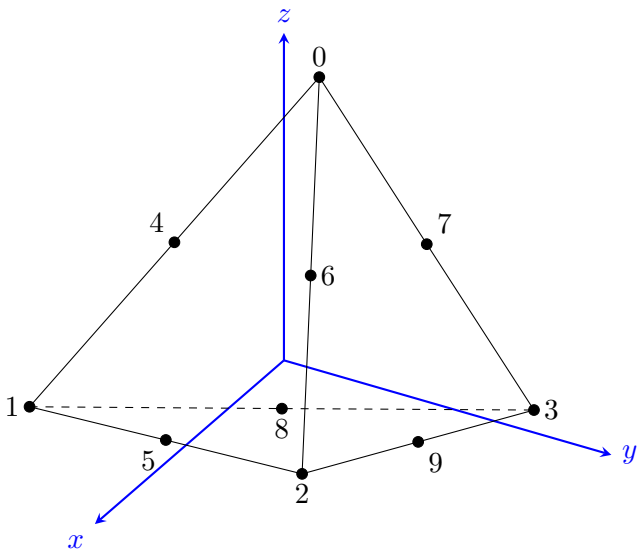

Figure 4.2. Ten-node tetrahedral element used in 3D Ansys field maps. The node numbering scheme shown here is the one used in Ansys and differs slightly from the one used internally in `ComponentAnsys123`.

- `ComponentAnsys123` reads three-dimensional field maps with quadratic curved tetrahedra (Fig. 4.2), known as "solid123" in Ansys).

The field map is imported with the function

```
bool Initialise(const std::string& elist, const std::string& nlist,
                const std::string& mplist, const std::string& prnsol,
                const std::string& unit);
```

elist name of the file containing the list of elements (default: `"ELIST.lis"`)

nlist name of the file containing the list of nodes (default: `"NLIST.lis"`)

mplist name of the file containing the list of materials (default: `"MPLIST.lis"`)

prnsol name of the file containing the nodal solutions (default: `"PRNSOL.lis"`)

unit length unit used in the calculation (default: `"cm"`,
other recognized units are `"mum"`/`"micron"`/`"micrometer"`, `"mm"`/`"millimeter"` and `"m"`/`"meter"`).

The return value is `true` if the map was read successfully.

In order to enable charge transport and ionization, the materials in the map need to be associated with `Medium` objects.

```
// Get the number of materials in the map.
size_t GetNumberOfMaterials();
// Associate a material with a Medium object.
void SetMedium(const size_t imat, Medium* medium);
```

imat index in the list of (field map) materials

medium pointer to the `Medium` object to be associated with this material

The materials in the field map are characterized by the relative dielectric constant ε and the conductivity σ. These parameters are accessible through the functions

```
double GetPermittivity(const size_t imat);
double GetConductivity(const size_t imat);
```

The function

```
void SetGas(Medium* medium);
```

associates a given `Medium` object to all field map materials with $\varepsilon = 1$.

By default, mesh elements in conductors, *i.e.* materials with resistivity equal to zero, are eliminated. This feature can be switched on or off using

```
void EnableDeleteBackgroundElements(const bool on = true);
```

on flag to delete mesh elements in conductors (`true`) or keep them (`false`)

A weighting field map can be imported using

```
bool SetWeightingField(std::string prnsol, std::string label);
```

prnsol name of the file containing the nodal solution for the weighting field configuration

label arbitrary name, used for identification of the electrode/signal

The weighting field map has to use the same mesh as the previously read "actual" field map.

For periodic structures, *e. g.* GEMs, one usually models only the basic cell of the geometry and applies appropriate symmetry conditions to cover the whole problem domain. The available symmetry operations are:

- simple periodicities,
- mirror periodicities,
- axial periodicities, and
- rotation symmetries.

Mirror periodicity means that odd periodic copies of the basic cell are mirror images of even periodic copies. Mirror periodicity and simple periodicity as well as axial periodicity and rotation symmetry are, obviously, mutually exclusive. In case of axial periodicity, the field map has to cover an integral fraction of 2π.

Periodicities can be set and unset using

```
void EnablePeriodicityX(const bool on);
void EnableMirrorPeriodicityX(const bool on);
void EnableAxialPeriodicityX(const bool on);
void EnableRotationSymmetryX(const bool on);
```

on flag to enable (`true`) or disable (`false`) periodicity.

Analogous functions are available for y and z.

In order to assess the quality of the mesh, one can retrieve the dimensions of each mesh element using

```
bool GetElement(const size_t i, double& vol, double& dmin, double& dmax);
```

i index of the element

vol volume/area of the element

dmin, dmax min./max. distance between two node points

In the following example we make histograms of the aspect ratio and element size.

```
ComponentAnsys123 fm;
// ...
TH1F* hAspectRatio = new TH1F("hAspectRatio"; "Aspect␣Ratio", 100, 0., 50.);
TH1F* hSize = new TH1F("hSize", "Element␣Size", 100, 0., 30.);
const size_t nel = fm.GetNumberOfElements();
// Loop over the elements.
double volume, dmin, dmax;
for (size_t i = 0; i < nel; ++i) {
  fm.GetElement(i, volume, dmin, dmax);
  if (dmin > 0.) hAspectRatio->Fill(dmax / dmin);
  hSize->Fill(volume);
}
TCanvas* c1 = new TCanvas();
hAspectRatio->Draw();
TCanvas* c2 = new TCanvas();
c2->SetLogy();
hSize->Draw();
```

Since Ansys uses curved elements, an iterative procedure is used for determining the local coordinates of a point within an element. This search can sometimes fail to achieve the requested precision. In this case, the first-order approximation is used, and (by default) a warning message is printed out. These messages can be switched on or off using

```
void EnableConvergenceWarnings(const bool on);
```

4.2.2. Synopsys TCAD

Electric fields calculated using the device simulation program Synopsys Sentaurus [47] can be imported with the classes `ComponentTcad2d` and `ComponentTcad3d` (derived from the base class `ComponentTcadBase`).

The function to import the field map is

```
bool Initialise(const std::string& gridfilename,
                const std::string& datafilename);
```

gridfilename name of the mesh file, the extension is typically `.grd`

datafilename name of the file containing the nodal solution; the filename typically typically ends with `_des.dat`

Both files have to be exported in DF-ISE format, files in the default TDR format cannot be read. To convert a TDR file to `_.dat` and `.grd` files, the Sentaurus tool `tdx` can be used

```
tdx -dd fieldToConvert.tdr
```

The classes have been tested with meshes created with the application `Mesh` which can produce axis-aligned two- and three-dimensional meshes. The only three-dimensional mesh elements `ComponentTcad3d` can deal with are tetrahedra. A mesh which consists only of simplex elements (triangles in 2D, tetrahedra in 3D), can be generated by invoking `Mesh` with the option `-t`.

After importing the files, the regions of the device where charge transport is to be enabled need to be associated with `Medium` objects.

```
// Get the number of regions in the device
size_t GetNumberOfRegions();
// Associate a region with a Medium object
void SetMedium(const size_t ireg, Medium* medium);
```

ireg index in the list of device regions

medium pointer to the `Medium` object to be associated with this region

To associate all regions with certain material to a given `Medium` object, one can use the function

```
void SetMedium(const std::string& material, Medium* medium);
```

material name of the material in TCAD

medium pointer to the `Medium` to be associated to all regions with this material.

The name of a region can be retrieved with

```
void GetRegion(const size_t i, std::string& name, bool& active);
```

name label of the region as defined in the device simulation

active flag indicating whether charge transport is enabled in this region

Simple periodicities and mirror periodicities along x, y, and – in case of `ComponentTcad3d` – z are supported.

```
void EnablePeriodicityX();
void EnableMirrorPeriodicityX();
```

Using TCAD, the weighting field and potential are typically determined by first computing the electric field $\mathbf{E}_0$ and potential V_0 with all electrodes set at the nominal potentials. The potential at the electrode to be read out is then increased by a small voltage ΔV and the corresponding

electric field E_+ and potential V_+ are calculated. The weighting field and potential are then given by

$$\mathbf{E}_w = \frac{1}{\Delta V}\left(\mathbf{E}_+ - \mathbf{E}_0\right), \qquad V_w = \frac{1}{\Delta V}\left(V_+ - V_0\right).$$

In `ComponentTcad2d` and `ComponentTcad3d` the weighting field and potential loaded using

```
SetWeightingField(const std::string& datfile1, const std::string& datfile2,
                  const double dv, const std::string& label);
```

datfile1 `.dat` file containing the solution $\mathbf{E}_0, V_0$,

datfile2 `.dat` file containing the solution $\mathbf{E}_+, V_+$,

dv potential difference ΔV,

label identifier of the electrode.

The mesh is assumed to be the same as the one for the drift field (imported using `Initialise`).

Using the function

```
bool SetWeightingFieldShift(const std::string& label,
                            const double x, const double y, const double z);
```

one can set an offset to be applied to the map of weighting fields and potentials imported from TCAD.

4.2.3. Elmer

The class `ComponentElmer` (contributed by J. Renner) allows one to import field maps created with the open source field solver Elmer [15] and the mesh tool Gmsh [16]. A detailed tutorial can be found on the webpage.

4.2.4. CST

The class `ComponentCST` (contributed by K. Zenker) reads field maps extracted from CST Studio. More details can be found at http://www.desy.de/ zenker/FLC/garfieldpp.html.

4.2.5. COMSOL

The class `ComponentComsol` can be used for importing field maps computed using COMSOL Multiphysics. The function to import a field map is

```
bool Initialise(std::string header = "mesh.mphtxt",
                std::string mplist = "dielectrics.dat",
                std::string field = "field.txt");
```

header COMSOL Multiphysics text file (`.mphtxt`) containing the mesh data,

mplist file containing the material properties,

field file containing the exported field data.

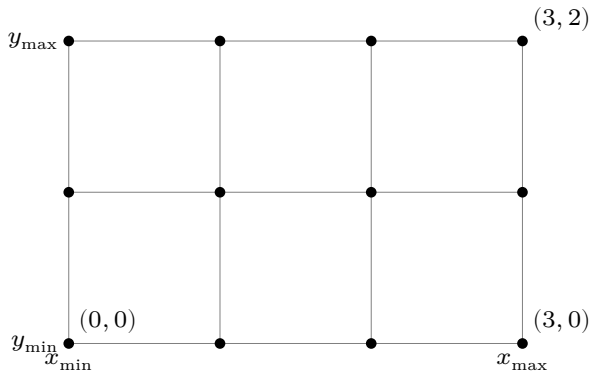

Figure 4.3. Example of a two-dimensional regular mesh (as used in `ComponentGrid`) with 4×3 grid points, corresponding to $n_x = 4, n_y = 3$.

The materials file (second argument) is a simply ASCII file containing

- the number of materials in the field map,
- the relative dielectric constants of each of the materials,
- the number of "domains",
- a list of domain numbers and indices of the corresponding materials.

The domain numbers can be found in the section `# Geometric entity indices` of the mesh data file (`mesh.mphtxt`). For a field map with two materials, one with a dielectric constant of $\varepsilon = 1$ (domain number 1) and the other one with a dielectric constant of $\varepsilon = 4$ (domain number 2), the file should look like the following.

```
2
1. 4.
2
1 0
2 1
```

4.2.6. Regular grids

Electric field values on a regular two-dimensional or three-dimensional grid can be imported from a text file using the class `ComponentGrid`. The electric field values (and potential) for each point on the grid are read in using

```
bool LoadElectricField(const std::string& filename, const std::string& format,
                       const bool withPotential, const bool withFlag,
                       const double scaleX = 1., const double scaleE = 1.,
                       const double scaleP = 1.);
```

filename name of the ASCII file

format description of the file format (see below)

withPotential flag whether the file contains an additional column with the electrostatic potential

withRegion flag whether the file contains an additional column with an integer value indicating whether the point is in an active medium (1) or not (0).

scaleX, scaleE, scaleP scaling factors to be applied to the coordinates, electric field values and potentials

The file should contain one line for each grid point. The available formats are `XY`, `XZ`, `IJ`, `IK`, `XYZ` and `IJK`, the first four for two-dimensional maps, and the last two for three-dimensional maps. In case of `XY` (`XYZ`), the first two (three) columns contain the x, y (and z) coordinates of a given point in the grid, followed by the electric field values (and potential if available) at this point. The class then determines the closest grid point and assigns the electric field and potential accordingly. In case of `IJ` (`IJK`) the indices of the grid point along x, y (and z) are specified directly. The file can include comments (lines starting with `#` or `//`).

Prior to reading the electric field, the limits and spacing of the grid can be set using the function

```
void SetMesh(const unsigned int nx, const unsigned int ny,
             const unsigned int nz, const double xmin, const double xmax,
             const double ymin, const double ymax, const double zmin,
             const double zmax);
```

nx, ny, nz number of grid lines along x, y, z

xmin, xmax, ... boundaries of the grid in x, y, z

An example illustrating the parameters defining the mesh and the numbering of the grid nodes is shown in Fig. 4.3. Alternatively, the grid parameters can be read from the file that contains the electric field/potential. For instance, to specify the limits and number of grid lines along z, a line like

```
# ZMIN = -1., ZMAX = 1., NZ = 3
```

should be included at the beginning of the file. If the user has not specified the grid explicitly (using `SetMesh`) and the information given in the comment lines is incomplete, the class will try to determine the remaining grid parameters from the table of coordinates and corresponding electric fields itself.

When retrieving the field/potential at a given point (x, y, z) the class performs a trilinear interpolation between the values at the grid nodes surrounding this point. The medium in the domain covered by the mesh is set using

```
void SetMedium(Medium* m);
```

A point (x, y, z) is considered in an active region (*i.e.* associated to the medium) if all surrounding grid nodes are flagged as active.

A magnetic field map can be imported using the function

```
bool LoadMagneticField(const std::string& filename, const std::string& format,
                       const double scaleX = 1., const double scaleB = 1.);
```

The available formats are the same as for the electric field (except for the extra columns for potential and active medium flag). Prompt and delayed weighting fields/potentials can be imported using

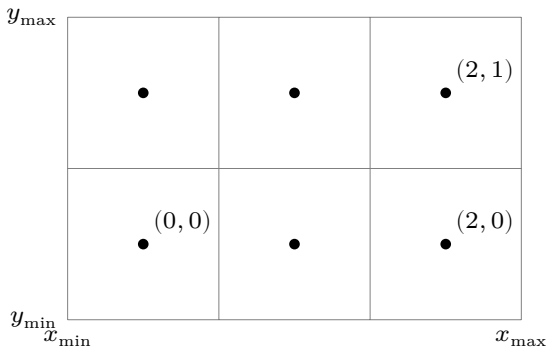

Figure 4.4. Example of a two-dimensional regular mesh (as used in `ComponentVoxel`) with 3×2 cells, corresponding to $n_x = 3, n_y = 2$.

```
bool LoadWeightingField(const std::string& filename, const std::string& format,
                        const bool withPotential,
                        const double scaleX = 1., const double scaleE = 1.,
                        const double scaleP = 1.);
bool LoadWeightingField(const std::string& filename, const std::string& format,
                        const double time, const bool withPotential,
                        const double scaleX = 1., const double scaleE = 1.,
                        const double scaleP = 1.);
```

Calling

```
void SetCylindricalCoordinates();
```

switches to a cylindrical coordinate system instead of the default Cartesian coordinates. The first coordinate ("x") then corresponds to the radial distance and the second coordinate ("y") corresponds to the azimuth (in radian).

The class `ComponentVoxel` is similar to `ComponentGrid` but uses a different definition of the mesh. An example illustrating the parameters of the mesh and the numbering of the voxels/cells is shown in Fig. 4.4. As a first step, the mesh needs to be defined using the function

```
void SetMesh(const unsigned int nx, const unsigned int ny,
             const unsigned int nz, const double xmin, const double xmax,
             const double ymin, const double ymax, const double zmin,
             const double zmax);
```

nx, ny, nz number of rows/columns of cells along x, y, z

xmin, xmax, ... boundaries of the grid in x, y, z

The electric field values (and potential) for each voxel are imported using

```
bool LoadElectricField(const std::string& filename, const std::string& format,
                       const bool withPotential, const bool withRegion,
                       const double scaleX = 1., const double scaleE = 1.,
                       const double scaleP = 1.);
```

filename name of the ASCII file

format description of the file format (see `ComponentGrid`)

withPotential flag whether the file contains an additional column with the electrostatic potential

withRegion flag whether the file contains an additional column with an integer value corresponding to the region index (each region can be associated with a different medium)

scaleX, scaleE, scaleP scaling factors to be applied to the coordinates, electric field values and potentials

The medium to be associated with a given region can be set using

```
void SetMedium(const unsigned int i, Medium* m);
```

i index of the region

If the regions are not assigned explicitly when importing the electric field, all voxels are assumed to belong to the same region (index 0).

By default, the field and potential are assumed to be constant within a voxel. Alternatively, the fields/potentials given in the field map file can be interpreted to be the values at the voxel centres and the fields/potentials at intermediate points be determined by trilinear interpolation. This feature can be activated using the function

```
void EnableInterpolation(const bool on = true);
```

4.2.7. Visualizing the mesh

For visualizing the mesh imported from a FEM field map (or a TCAD field map), the class `ViewFEMesh` (written by J. Renner) is available. Using

```
void ViewFEMesh::SetViewDrift(ViewDrift* driftView);
```

a `ViewDrift` object can be attached to the mesh viewer. The function

```
bool ViewFEMesh::Plot(const bool twod = true, const bool outline = false);
```

then draws the drift lines stored in the `ViewDrift` class together with the mesh. If the flag `twod` is set to `true`, a two-dimensional projection of the drift lines and a cut view of the mesh at the currently set viewing plane are drawn. If `twod` is `false`, the drift lines and mesh elements are drawn in 3D. If `outline` is set to `true`, only the facets separating two regions are drawn. The plot can be customized using

```
void SetColor(int matid, int colorid);
void SetFillColor(int matid, int colorid);
void SetFillMesh(bool fill);
```

matid material index in the field map

colorid index of the ROOT color with which all elements of material `matid` are to be drawn

fill flag indicating whether to draw a wireframe mesh (`false`) or filled elements

As an illustration consider the following example (suppose that `driftView` is a pointer to a `ViewDrift` object)

```
ComponentAnsys123 fm;
// ...
TCanvas* c1 = new TCanvas();
ViewFEMesh* meshView = new ViewFEMesh(&fm);
meshView->SetCanvas(c1);
// Set the viewing plane.
meshView->SetPlane(0, -1, 0, 0, 0, 0);
meshView->SetFillMesh(false);
meshView->SetViewDrift(driftView);
meshView->SetArea(-0.01, -0.01, -0.03, 0.01, 0.01, 0.01);
meshView->Plot();
```

```
ComponentAnsys123 fm;
// ...
TCanvas* c1 = new TCanvas();
ViewFEMesh* meshView = new ViewFEMesh(&fm);
meshView->SetCanvas(c1);
// Set the viewing plane.
meshView->SetPlane(0, -1, 0, 0, 0, 0);
meshView->SetFillMesh(false);
meshView->SetViewDrift(driftView);
meshView->SetArea(-0.01, -0.01, -0.03, 0.01, 0.01, 0.01);
meshView->Plot();
```

4.3. Analytic fields

For two-dimensional geometries consisting of wires, planes and tubes, semi-analytic calculation techniques are implemented. The equations to be solved to find the wire charges are known as capacitance equations, they are obtained by expressing the (known) potential of wire i in the (unknown) charges per unit length q_j on the wires $j = 1 \ldots n$,

$$V_i = \sum_{j=1}^{n} C_{ij}^{-1} q_j + V_{\text{ref}}$$

When planes are absent, the freedom to choose a reference potential V_{ref} can be exploited to ensure that the sum of all charges is zero, by adding

$$\sum_{j=1}^{n} q_j = 0$$

to the set of equations. If there is at least one equipotential plane, the sum of all charges is automatically zero (the charges and mirror charges cancel) and the reference potential is set equal to zero in this case.

The equipotential planes can be treated as if they were grounded if the linear potential generated by the planes alone is subtracted from all wire-potentials before the charges are calculated and added separately when the potential and electrostatic field are evaluated.

4.3.1. Describing the cell

The medium to be associated with the active region of the cell is set using

```
SetMedium(Medium* medium);
```

Wires, tubes and planes can be added to the cell layout by means of the following functions:

```
// Add a wire
void AddWire(const double x, const double y, const double d,
             const double v, const std::string& label = "",
             const double length = 100.,
             const double tension = 50., const double rho = 19.3);
// Add a tube
void AddTube(const double r, const double v,
             const int nEdges, const std::string& label = "");
// Add a plane at constant x
void AddPlaneX(const double x, const double v, const std::string& label = "");
// Add a plane at constant y
void AddPlaneY(const double y, const double v, const std::string& label = "");
```

All of these functions take the potential `v` (in V) and a label as arguments. The `label` argument is optional. If the default value (empty string) is used, the calculation of the weighting field and potential for this wire, tube or plane will be skipped (*i. e.* one cannot compute the signal induced in this element).

For wires, the center of the wire (`x, y`) and its diameter (`d`) need to be specified. Optional parameters are the wire length, the tension (more precisely, the mass in g of the weight used to stretch the wire during the assembly) and the density (in $\mathrm{g\,/\,cm^3}$) of the wire material. These parameters have no influence on the electric field. The number of wires that can be added is not limited.

Tube-specific parameters are the radius[1] (`r`) and the number of edges, which determines the shape of the tube:

- $n = 0$: cylindrical pipe
- $3 \leq n \leq 8$: regular polygon

There can be only one tube in a cell. The tube is always centered at the origin $(0, 0)$.

Planes are described by their coordinates. A cell can have at most two x and two y planes. Planes and tubes cannot be used together in the same cell.

The geometry can be reset (thereby deleting all wires, planes and tubes) by

```
void Clear();
```

Before assembling and inverting the capacitance matrix, a check is performed whether the provided geometry matches the requirements. If necessary, the planes and wires are reordered. Wires outside the tube or the planes as well as overlapping wires are removed.

[1] For non-circular tubes, this parameter is the distance between the origin and any of the edges.

4.3.2. Cylindrical geometries

By default, the wire coordinates are specified in Cartesian coordinates and the planes are parallel to the x or y axis. Calling

```
void SetPolarCoordinates();
```

switches to a polar coordinate system. This can be useful for certain types of cells which are more conveniently described in polar coordinates (r, ϕ), for instance because of the presence of a circular plane.

Internally, the class uses log-polar coordinates (ρ, ϕ), where the angular coordinate ϕ is the same as in polar coordinates, and $\rho = \ln r$. The transformation[2]

$$(x, y) = \mathrm{e}^{\rho} (\cos\phi, \sin\phi)$$

is a conformal mapping[3] which translates circles to lines at constant ρ. After calculating the field in internal coordinates, it is transformed back to Cartesian coordinates using

$$\begin{pmatrix} E_x \\ E_y \end{pmatrix} = \mathrm{e}^{\rho} \begin{pmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{pmatrix} \begin{pmatrix} E_\rho \\ E_\phi \end{pmatrix}.$$

By calling

```
void SetCartesianCoordinates();
```

one can change back to Cartesian coordinates. Mixed coordinates are not permitted; when switching from Cartesian to polar coordinates or vice versa the description of the cell is reset.

In order to add a wire to the cell, the same function is used for both Cartesian and polar coordinates. In the latter case, the first coordinate corresponds to the radius of the wire (in cm), and the second coordinate corresponds to the angle (in degrees). A wire must not be positioned at the origin if polar coordinates are being used.

Planes at constant radius or at constant angle are specified using

```
void AddPlaneR(const double r, const double voltage, const std::string& label);
void AddPlanePhi(const double phi, const double voltage, const std::string& label);
```

r radius of the plane (in cm),

phi angle of the plane (in degrees).

The following simple example generates a pie wedge with a wire inside.

```
ComponentAnalyticField cmp;
cmp.SetPolarCoordinates();
const double r = 2.;
cmp.AddPlaneR(r, 0., "r");
cmp.AddPlanePhi( 0., 0., "phi1");
cmp.AddPlanePhi(60., 0., "phi2");
cmp.AddWire(0.5 * r, 30., 50.e-4, 500., "w");
```

[2] Using complex numbers, the transformation can be written as $x + \mathrm{i}y = \exp(\rho + \mathrm{i}\phi)$.
[3] This implies that the Laplace equation in log-polar coordinates has the same form as in Cartesian coordinates.

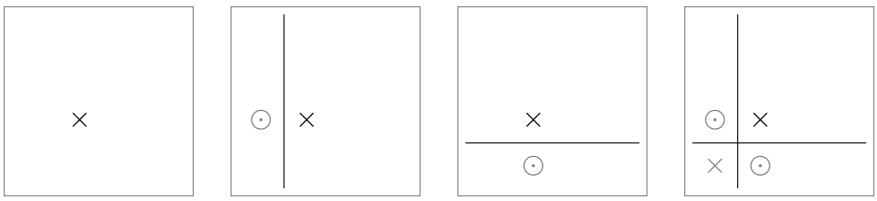

Figure 4.5. Examples of A-type cells (isolated wires with at most one x-plane and one y-plane).

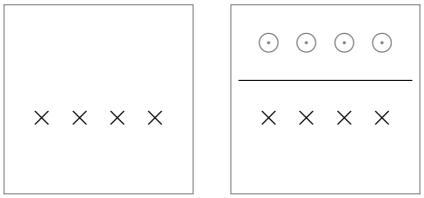

Figure 4.6. Examples of B1X-type cells (x-periodic array of wires with at most one y-plane).

4.3.3. Periodicities

The class supports simple periodicity in x and y direction. The periodic lengths are set using

```
void SetPperiodicityX(const double s);
void SetPeriodicityY(const double s);
```

When working in polar coordinates, one can set the ϕ periodicity using

```
void SetPeriodicityPhi(const double s);
```

s angular period (in degrees).

If the ϕ periodicity is not set explicitly by the user, a cyclic boundary condition with a period of 2π is imposed.

Radial periodicity is not supported since the internal coordinate ρ is not linear in r.

4.3.4. Cell types

Internally, cells are classified as belonging to one of these types:

A non-periodic cells with at most one x and one y plane

B1X x-periodic cells without x planes and at most one y plane

B1Y y-periodic cells without y planes and at most one x plane

B2X cells with two x planes and at most one y plane

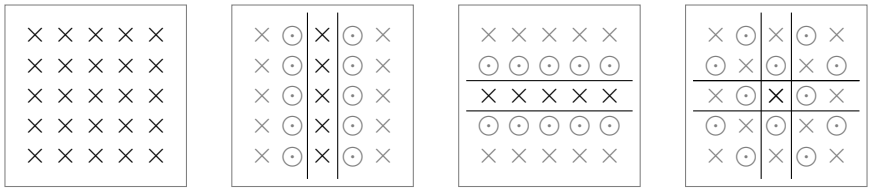

Figure 4.7. Examples of C-type cells (doubly periodic arrays of wires).

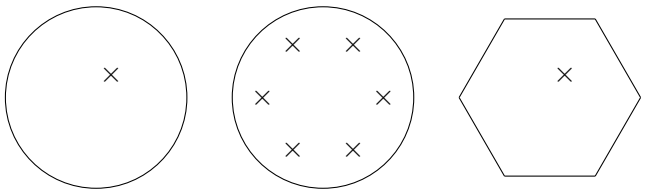

Figure 4.8. Examples of D-type cells. Left: isolated wire in a circular tube (D1). Middle: ϕ-periodic ring of wires in a circular tube (D2). Right: wire in a polygon (D3).

B2Y cells with two y planes and at most one x plane

C1 doubly periodic cells without planes

C2X doubly periodic cells with x planes

C2Y doubly periodic cells with y planes

C3 doubly periodic cells with x and y planes

D1 round tubes without axial periodicity

D2 round tubes with axial periodicity

D3 polygonal tubes without axial periodicity

After the cell has been assembled and initialized, the cell type can be retrieved by the function

```
std::string GetCellType();
```

4.3.5. Dipole moments

By default, `ComponentAnalyticField` uses the thin-wire approximation for computing the electric field and potential. In this approach, dipole and higher-order terms are neglected which is usually a good approximation if the wire spacing is large compared to the diameter. One can request dipole terms to be included in the calculation using

```
void EnableDipoleTerms(const bool on = true);
```

Dipole terms are currently implemented only for cell types A, B1X, B1Y, B2X and B2Y. To investigate whether dipole and higher order terms are significant, one can use the function

```
bool MultipoleMoments(const unsigned int iw, const unsigned int order = 4,
                      const bool print = false, const bool plot = false,
                      const double rmult = 1.,
                      const double eps = 1.e-4, const unsigned int nMaxIter = 20);
```

iw index of the wire for which the multipole decomposition should be done,

order order of the highest multipole moment to be taken into account,

print flag to request verbose output during the minimisation step,

plot flag to create a plot of the result of the multipole fit,

rmult distance (in units of the wire radius) at which the potential is to be calculated,

eps "small number" used by the minimisation function to calculate the derivative matrix,

nMaxIter max. number of iteration in the minimisation.

4.3.6. Weighting fields

By default, weighting fields and potentials will be calculated for all elements (wires, planes, *etc.*) which were assigned a non-empty string as a label.

In addition to the weighting fields of the elements used for the calculation of the (actual) electric field, the weighting field for a strip segment of a plane can also be calculated. Strips can be defined using

```
void AddStripOnPlaneX(const char direction, const double x,
                      const double smin, const double smax,
                      const std::string& label, const double gap = -1.);
void AddStripOnPlaneY(const char direction, const double y,
                      const double smin, const double smax,
                      const std::string& label, const double gap = -1.);
```

direction orientation of the strip (`'y'` or `'z'` in case of x-planes, `'x'` or `'z'` in case of y-planes

x, y coordinate of the plane on which the strip is located

smin, smax min. and max. coordinate of the strip

The strip weighting field is calculated using an analytic expression for the field between two infinite parallel plates which are kept at ground potential except for the strip segment, which is raised to 1 V. The anode-cathode distance d to be used for the evaluation of this expression can be set by the user (variable `gap` in `AddStripOnPlaneX`, `AddStripOnPlaneY`). If this variable is not specified (or set to a negative value), the following default values are used:

- if two planes are present in the cell, d is assumed to be the distance between those planes;
- if only one plane is present, d is taken to be the distance to the nearest wire.

Similarly, pixels can be defined using

```
void AddPixelOnPlaneX(const double x, const double ymin, const double ymax,
                      const double zmin, const double zmax,
                      const std::string& label, const double gap = -1.,
                      const double rot = 0.);
void AddPixelOnPlaneY(const double y, const double xmin, const double xmax,
                      const double zmin, const double zmax,
                      const std::string& label, const double gap = -1.,
                      const double rot = 0.);
```

The last (optional) parameter specifies the rotation angle (in rad) of the pixel in the plane. Pixel weighting fields are calculated using the expressions given in Ref. [34].

When working in polar coordinates, strips and pixels are defined using

```
void AddStripOnPlaneR(const char direction, const double r, const double smin    ,
                      const double smax, const std::string& label,
```

```
                        const double gap = -1.);
void AddStripOnPlanePhi(const char direction, const double phi, const double     smin,
                          const double smax, const std::string& label,
                          const double gap = -1.);
void AddPixelOnPlaneR(const double r,
                        const double phimin, const double phimax,
                        const double zmin, const double zmax,
                        const std::string& label, const double gap = -1.);
void AddPixelOnPlanePhi(const double phi,
                          const double rmin, const double rmax,
                          const double zmin, const double zmax,
                          const std::string& label, const double gap = -1.);
```

Valid strip directions are `'p'` (ϕ) or `'z'` for circular planes and `'r'` or `'z'` for ϕ planes.

In periodic chambers, the electric field is identical in all copies of the basic cell, but the weighting fields of the wires of one copy are as a rule not the same as the weighting fields of the wires of another copy. `ComponentAnalyticField` will, by default, compute the weighting fields and potentials by considering only the basic cell – suppressing all periodicities. This is a good approximation if the wires being read out are surrounded by many other wires in the basic cell.

This behaviour can be controlled using the function

```
void SetNumberOfCellCopies(const unsigned int nfourier);
```

The argument (`nfourier`) must be 0 or an integral power of 2.

If the argument is 0, the weighting field and potential will be computed using the same functions that are used for the electric field, *i. e.* they will exhibit the same periodicity. This should be used if one is interested only in the signals induced in the planes (and not in the wire signals) or if all copies of the read-out wires are interconnected.

If the argument is non-zero, `SetNumberOfCellCopies` sets the number of copies of the basic cell to be included in the weighting field calculation. Requesting a large number of copies is meaningful only in chambers which contain equipotential planes - convergence is poor otherwise.

4.3.7. Wire displacements

The forces acting on a wire and the wire displacement that results from these forces can be computed using the methods

```
bool ForcesOnWire(const unsigned int iw,
                  std::vector<double>& xMap, std::vector<double>& yMap,
                  std::vector<std::vector<double> >& fxMap,
                  std::vector<std::vector<double> >& fyMap);
```

and

```
bool WireDisplacement(const unsigned int iw, const bool detailed,
                      std::vector<double>& csag, std::vector<double>& xsag,
                      std::vector<double>& ysag, double& stretch,
                      const bool print = true);
```

of `ComponentAnalyticField`. In both methods, the argument `iw` is the index of the wire (you can use the function `PrintCell` to print a list of all wires in the cell layout). By default, both gravitational force and electrostatic force are taken into account for computing the displacement. This can be changed using

```
void EnableGravity(const bool on = true);
void EnableElectrostaticForce(const bool on = true);
```

The direction in which gravity acts on the wires can be set using

```
void SetGravity(const double dx, const double dy, const double dz);
```

The three arguments are the components of a vector indicating the direction in which gravity pulls on the wires. Only the direction of the vector is used, not its normalisation.

The function `WireDisplacement` fills the vector `csag` with a set of coordinates along the wire and the vectors `xsag`, `ysag` with the x and y components of the sag profile at these coordinates, and sets the variable `stretch` to the relative elongation. It offers two levels of accuracy.

- With the flag `detailed` set to `false`, only the force acting on the wire in its nominal position is used to compute the sag. The wire sag that results from such a force is parabolic, since it results from an elastic elongation. The shape is not a hyperbolic cosine, this would be the case of a freely hanging wire. This approach is incorrect if the wire is nominally in an almost stable position while there are substantial forces acting on the wire in neighbouring positions.
- If the flag `detailed` is set to `true`, the method also considers the force acting on the wire in the vicinity of its nominal position. The electrostatic forces are computed by solving the capacitance equations for a set of wire locations on a regular grid around the nominal location and interpolating the resulting table of forces. The number of grid lines can be set using

  ```
  void SetScanningGrid(const unsigned int nX, const unsigned int nY);
  ```

  The default is 11 lines in both x and y. By default, the range of wire shifts for which the forces are computed, is selected automatically by enlarging by a scaling factor (default: 2) the zeroth order estimates of the shift, and restricting this to the largest area around the wire which is free of other cell elements. The scaling factor can be set using

  ```
  void SetScanningAreaFirstOrder(const double scale = 2.);
  ```

  If the wire movements are expected to be very large, then one may wish to set the scanning area to the largest area around the wire which is free from other cell elements:

  ```
  void SetScanningAreaLargest();
  ```

  You may also manually set the scanning area by specifying a lower x, an upper x, a lower y and an upper y which together describe a rectangular area relative to the nominal position of the wire under consideration.

```
void SetScanningArea(const double xmin, const double xmax,
                     const double ymin, const double ymax);
```

By default, the calculation stops if a point of the wire is found at a position not covered by the scanning area, whether set manually or automatically. If you enable extrapolation using

```
void EnableExtrapolation(const bool on = true);
```

then the force on the wire at such a point will be computed by extrapolating the force table. It is usually a better strategy to pick a scanning area that covers all wire positions – if this is not possible, then the wire is probably not in a stable position (*i.e.* it will move against other electrodes).

In the detailed approach, the differential equation that governs the wire sag is numerically solved using a multiple shooting method in which each shot is traced with a Runge-Kutta-Nystroem method, and in which the boundary and matching conditions are minimised with a Newton method with Broyden rank-1 updates of the derivative matrix. The number of shots and the number of integration steps within each shot can be set by the user

```
void SetNumberOfShots(const unsigned int n);
void SetNumberOfSteps(const unsigned int n);
```

The function `ForcesOnWire` fills the vectors `xMap`, `yMap` with the coordinates of the scanning grid lines (relative to the nominal wire position) and the vectors `fxMap`, `fyMap` with the x and y components of the electrostatic force at the grid points. The scanning grid can be set using the same methods as for `WireDisplacement`.

4.3.8. Optimisation

`ComponentAnalyticField` includes a set of methods that vary the potentials of a set of electrodes in an attempt to match as closely as possible a function of the potential and field (field function) to a given target value.

- The method

```
bool OptimiseOnTrack(const std::vector<std::string>& groups,
    const std::string& field_function, const double target,
    const double x0, const double y0, const double x1, const double y1,
    const unsigned int nP = 20, const bool print = true);
```

compares target and field function on `nP` sampling points along a straight line between (x_0, y_0) and (x_1, y_1). This function can be used *e. g.* to adjust the potentials on a drift electrode in a TPC to get the proper drift field.

-

```
bool OptimiseOnGrid(const std::vector<std::string>& groups,
    const std::string& field_function, const double target,
    const double x0, const double y0, const double x1, const double y1,
    const unsigned int nX = 10, const unsigned int nY = 10,
    const bool print = true);
```

compares target and field function on a regular $x-y$ (or $r-\phi$) grid. The extent of the grid area is defined by the two corners $(x_0, y_0), (x_1, y_1)$ and the grid density is defined by `nX`, `nY` (number of lines in x and y).

- ```
  bool OptimiseOnWires(const std::vector<std::string>& groups,
      const std::string& field_function, const double target,
      const std::vector<unsigned int>& wires, const bool print = true);
  ```

  compares target and field function on the surfaces of a set of wires (the vector `wires` contains the wire indices). This function can be used *e. g.* to adjust the voltages on the anode wires such that the field in their proximity has a given value.

The vector `groups` contains the labels of the electrodes for which to adjust the potentials. Electrodes that are put together in a group are shifted collectively. In the expression `field_function` the following variables may be used:

- `x` or `r` (x or r coordinate),
- `y` or `phi` (y or ϕ coordinate),
- `ex` or `er` (x or r-component of the electric field),
- `ey` or `ephi` (y or ϕ-component of the electric field),
- `e` (norm of the electric field),
- `v` (electric potential).

More variables can be added on demand.

If the flag `print` is set to `true`, some optimisation information is printed at each cycle.

The method used is that of repeated Householder steps minimising (in the Euclidean norm) the difference between target and field function. Several conditions can cause the iteration to be stopped:

- the user defined maximum number of iterations is reached,
- the relative change in Euclidean distance between the target and field function (on the sampling points) drops below threshold ε,
- the maximum distance (of all sampling points) between target and field function becomes smaller than a threshold distance.

These parameters can be set using the method

```
void SetOptimisationParameters(const double dist = 1.,
                               const double eps = 1.e-4,
                               const unsigned int nMaxIter = 10);
```

The parameter ε is also used for computing the numerical derivatives needed for the covariance matrix. A larger value (say 1) should be chosen when you know you are far from the optimised value and a smaller value (say 10^{-4}) when your initial guess is quite good.
```

4.4. neBEM

The nearly exact Boundary Element Method (neBEM) solver discretizes the boundary of the domain in triangular and rectangular elements and computes the surface charge density on each of these elements that is required to satisfy the applied boundary conditions. Once the charge density distribution has been obtained, the potential and field can be evaluated at any point in the domain. A detailed description of the technique can be found in Refs. [25, 21, 24].

The `ComponentNeBem3d` interface class requires as input a `GeometrySimple` object with its list of solids and associated media.

```
void ComponentNeBem3d::SetGeometry(Geometry* geo);
```

The boundary conditions to be applied to each solid can be set using

```
void Solid::SetBoundaryPotential(const double v);
```

in case of a conductor at fixed potential (Dirichlet boundary conditions) or

```
void Solid::SetBoundaryDielectric();
```

in case of a dielectric-dielectric interface. In the latter case (Neumann boundary conditions), the normal component of the displacement field $\mathbf{D} = \varepsilon\mathbf{E}$ is required to be continuous at the boundary of the solid,

$$\mathbf{n} \cdot (\varepsilon^{+}\mathbf{E}^{+} - \varepsilon^{-}\mathbf{E}^{-}) \, .$$

During initialisation, `ComponentNeBem3d`

- retrieves the surface panels from the `Solid` objects present in the geometry,
- searches for and eliminates overlaps between the panels,
- and splits the resulting polygons into rectangular and triangular "primitives".

These "primitives" are then passed on to neBEM, where they are further subdivided into "elements" (rectangles and right-angled triangles). A special case are `SolidWire` objects which are represented as one-dimensional straight line primitives and elements (corresponding to the thin-wire approximation used also in `ComponentAnalyticField`). The elements should be small enough such that the distribution of the charge density on them be approximated as uniform. The size of the elements can be controlled using the functions

```
void SetTargetElementSize(const double length);
void SetMinMaxNumberOfElements(const unsigned int nmin, const unsigned int nmax);
```

length preferred linear size of the elements, measured along their edges.

nmin,nmax smallest and largest number of elements produced along either axis of a single primitive.

One can also request different target element sizes for each `Solid` object, using

```
void Solid::SetDiscretisationLevel(const double dis);
```

After splitting the primitives into elements, neBEM determines the influence matrix K and the right-hand side vector[4] of the system of equations

$$K\rho = \mathbf{b},$$

inverts the matrix and computes the surface charge density ρ_i of every element,

$$\rho_i = K^{-1}_{ij} b_j.$$

4.4.1. Weighting fields

If a `Solid` object has been given a label using

```
void Solid::SetLabel(const std::string& label);
```

neBEM will compute its weighting field at the same time as the electric field. One can assign the same label to multiple `Solid` objects. The weighting field corresponding to this label will then be computed by setting the potential of all elements associated to solids with this label to 1 V (and grounding all other conducting elements).

4.5. Parameterisations

The class `ComponentUser` takes the electric field and potential from a user-defined function.

```
void SetElectricField(std::function<void(const double, const double, const double,
                                         double& double&, double&)> f);
void SetPotential(std::function<double(const double, const double, const double)> f);
```

f user function

The signature of the electric field function is `void(const double, const double, const double, double&, double&, double&)`, the first three arguments being the coordinates x, y, z and the last three (reference) arguments being the components of the electric field (to be assigned in the function). The potential function takes the coordinates x, y, z as arguments and returns the potential. Similar functions are available to set the weighting field and potential, and the magnetic field.

Alternatively, `ComponentUser` also allows one to define the electric field and potential in string expressions, which are then just-in-time compiled internally using the ROOT C++ interpreter.

```
void SetElectricField(const std::string& expression);
void SetPotential(const std::string& expression);
```

expression a compileable C++ expression.

Valid parameter names are `x`, `y`, `z`. The expression used for evaluating the electric field shall assign the variables `ex`, `ey`, `ez` (corresponding to the three field components). If a field component is not assigned in the expression, it is assumed to be zero.

If the active volume is a box, its limits can be set using the functions

[4] In a system with only Dirichlet boundary conditions, the right-hand side vector is given by the potentials at the elements, $b_i = V_i$.

```
void SetArea(const double xmin, const double ymin, const double zmin,
             const double xmax, const double ymax, const double zmax);
```

In this case, the medium associated to the active volume is set using

```
void SetMedium(Medium* medium);
```

Alternatively, the geometry can be defined using

```
void SetGeometry(Geometry* geo);
```

As an example, let us consider the electric field in the bulk of an overdepleted planar silicon sensor, given by

$$E(x) = \frac{V - V_{\text{dep}}}{d} + 2x\frac{V_{\text{dep}}}{d^2},$$

where V is the applied bias voltage, V_{dep} is the depletion voltage, and d is the thickness of the diode.

```
MediumSilicon si;
// Detector thickness
const double d = 0.1;

ComponentUser cmp;
cmp.SetArea(0., -2 * d, -2 * d, d, 2 * d, 2 * d);
cmp.SetMedium(&si);
auto efield = [](const double x, const double y, const double z,
                 double& ex, double& ey, double& ez) {

  // Depletion voltage
  const double vdep = 160.;
  // Applied voltage
  const double v = 200.;

  ey = ez = 0.;
  ex = (v - vdep) / d + 2 * x * vdep / (d * d);
};
cmp.SetElectricField(efield);
```

Using the JIT-compilation approach, the above example would look like this.

```
MediumSilicon si;
// Detector thickness
const double d = 0.1;

ComponentUser cmp;
cmp.SetArea(0., -2 * d, -2 * d, d, 2 * d, 2 * d);
cmp.SetMedium(&si);
// Depletion voltage
const double vdep = 160.;
// Applied voltage
const double v = 200.;
```

```
std::string efield = "ex␣=␣" + std::to_string((v - vdep) / d) +
                     "2␣*␣x␣*␣" + std::to_string(vdep / (d * d));
cmp.SetElectricField(efield);
```

4.6. Other components

For simple calculations, the class `ComponentConstant` can be used. As the name implies, it provides a uniform electric field. The electric field and potential can be specified using

```
void SetElectricField(const double ex, const double ey, const double ez);
void SetPotential(const double x, const double p, const double z, const double v);
```

ex, ey, ez components of the electric field

x, y, z coordinates where the potential is specified

v voltage at the given position

The weighting field and potential can be set using

```
void SetWeightingField(const double wx, const double wy, const double wz,
                       const std::string label);
void SetWeightingPotential(const double x, const double y, const double z,
                           const double v);
```

wx,wy,wz components of the weighting field

label identifier of the weighting field/electrode

x, y, z coordinates where the weighting potential is specified

v weighting potential at the given position

As is the case with `ComponentUser`, the active volume can be set using `SetArea` and `SetMedium` (if it is a box), or using `SetGeometry` in case of more complex geometries.

4.7. Visualizing the field

The class `ViewField` provides some basic functions for plotting the potential and field of a component/sensor.

The `Component` or `Sensor` from which to retrieve the field/potential to be plotted is set through the constructor of `ViewField` or by means of

```
void SetComponent(Component* c);
void SetSensor(Sensor* s);
```

By default, the voltage range is retrieved from the minimum and maximum values of the potential in the component/sensor, and the range of the electric and weighting fields is "guessed" by taking random samples. This feature can be switched on or off using the function

```
void EnableAutoRange(const bool on = true, const bool samplePotential = true);
```

The flag `samplePotential` indicates whether the range of the potential should be determined by random sampling or if `ViewField` should first try to retrieve it from the component/sensor.

If the "auto-range" feature is disabled, the range of the function to be plotted needs to be set using

```
void SetVoltageRange(const double vmin, const double vmax);
void SetElectricFieldRange(const double emin, const double emax);
void SetWeightingFieldRange(const double wmin, const double wmax);
```

4.7.1. One-dimensional plots

The function

```
void PlotProfile(const double x0, const double y0, const double z0,
                 const double x1, const double y1, const double z1,
                 const std::string& option = "v",
                 const bool normalised = true);
```

plots the potential or field (depending on the parameter `option`) along the line $(x_0, y_0, z_0) \rightarrow (x_1, y_1, z_1)$. With the flag `normalised` set to `true` (default), normalised coordinates $[0, 1]$ are used for the x-axis.

Similar functions are available for visualizing weighting potentials and fields.

```
void PlotContourWeightingField(const std::string& label, const std::string& option);
void PlotWeightingField(const std::string& label, const std::string& option,
                        const std::string& drawopt);
void PlotProfileWeightingField(const std::string& label,
                   const double x0, const double y0, const double z0,
                   const double x1, const double y1, const double z1,
                   const std::string& option = "v",
                   const bool normalised = true);
```

label identifier of the electrode for which to plot the weighting field/potential.

4.7.2. Two-dimensional plots

The functions

```
void PlotContour(const std::string& option = "v");
void Plot(const std::string& option = "v", const std::string& drawopt = "");
```

create a contour plot or another two-dimensional plot in the chosen viewing plane. The quantity to be plotted is set using the parameter `option` (see Table 4.2). The parameter `drawopt` is passed on to the function `Draw()` of the ROOT `TF2` class and sets the plotting options. For instance,

```
ViewField view;
view.Plot("v", "SURF4");
```

Table 4.2. `ViewField` option strings and corresponding quantities.

Quantity	`option`
Electrostatic potential	`"v"`, `"p"`, `"phi"`
Magnitude of the electric field ($\|\mathbf{E}\|$)	`"e"`, `"emag"`, `"norm"`
x-component of the electric field (E_x)	`"ex"`
y-component of the electric field (E_y)	`"ey"`
z-component of the electric field (E_z)	`"ez"`
Magnitude of the magnetic field ($\|\mathbf{B}\|$)	`"bmag"`
x-component of the magnetic field (B_x)	`"bx"`
y-component of the magnetic field (B_y)	`"by"`
z-component of the magnetic field (B_z)	`"bz"`

will create a surface plot of the potential.

The viewing plane and the region to be drawn can be specified using

```
void SetArea(const double xmin, const double ymin, const double xmax, const double ymax);
void SetPlane(const double fx, const double fy, const double fz,
              const double x0, const double y0, const double z0);
void Rotate(const double angle);
```

xmin, ymin, xmax, ymax plot range in "local coordinates" (in the current viewing plane).

fx, fy, fz normal vector of the plane.

x0, y0, z0 in-plane point.

angle rotation angle (in radian).

By default, the viewing plane is the $x-y$ plane (at $z = 0$) and the plot range is retrieved from the bounding box of the component/sensor. The default viewing plane can be restored using

```
void SetPlaneXY();
```

and the feature to determine the plot area from the bounding box can be activated using

```
void SetArea();
```

The density of the plotting grid can be set using

```
void SetNumberOfSamples1d(const unsigned int n);
void SetNumberOfSamples2d(const unsigned int nx, const unsigned int ny);
```

n, nx, ny number of points in x and y direction (default for one-dimensional plots: $n = 1000$; default for two-dimensional plots: $n_x = n_y = 200$)

The number of contour levels can be set using

```
void SetNumberOfContours(const unsigned int n);
```

4.7.3. Field lines

The function

```
void PlotFieldLines(const std::vector<double>& x0,
                    const std::vector<double>& y0,
                    const std::vector<double>& z0,
                    const bool electron = true, const bool axis = true,
                    const short col = kOrange - 3)
```

computes and draws electric field lines from a set of starting points, given by the vectors `x0`, `y0`, `z0`. The flag `electron` specifies whether the field lines should be calculated for a negative (electron-like) test charge or for a positive test charge.

A useful helper function for determining the starting points is

```
bool EqualFluxIntervals(const double x0, const double y0, const double z0,
                        const double x1, const double y1, const double z1,
                        std::vector<double>& xf, std::vector<double>& yf,
                        std::vector<double>& zf,
                        const unsigned int nPoints = 20) const;
```

which fills the vectors `xf`, `yf`, `zf` with the coordinates of `nPoints` along the line $(x_0, y_0, z_0) - -(x_1, y_1, z_1)$ which are spaced by equal flux intervals. The flux is computed by integrating the electric field component that is in the viewing plane and perpendicular to the line.

If the flux changes sign over the track, then points are only generated over the parts of the line where the flux is positive if the total flux over the line is positive. Conversely, if the total flux is negative, points are generated only in areas where the flux is negative.

A similar function is

```
bool FixedFluxIntervals(const double x0, const double y0, const double z0,
                        const double x1, const double y1, const double z1,
                        std::vector<double>& xf, std::vector<double>& yf,
                        std::vector<double>& zf,
                        const double interval = 10.) const;
```

It fills the vectors `xf`, `yf`, `zf` with the coordinates of points that are spaced by a given flux interval (in units of V).

4.8. Inspecting the field

The `Component` base class provides functions for inspecting the field, in particular for determining the electric flux over a surface.

```
double IntegrateFluxSphere(const double xc, const double yc, const double zc,
                           const double r, const unsigned int nI = 20);
```

calculates the charge (in fC) enclosed in a sphere of radius r centred at (x_c, y_c, z_c) using Gauss's law, *i.e.* by integrating the normal component of the electric field over the surface of the

sphere,

$$Q = \varepsilon_0 \oint \mathbf{E} \cdot \mathrm{d}\mathbf{A}.$$

Similarly,

```
double IntegrateFluxCircle(const double xc, const double yc,
                           const double r, const unsigned int nI = 50);
```

calculates the line charge (in fC / cm) contained in a circle of radius r centred at (x_c, y_c). The integrations are performed using six-point Gaussian quadrature. The number of integration intervals can be set as a parameter.

The integral of the flux over a parallelogram can be calculated using the function

```
double IntegrateFluxParallelogram(const double x0, const double y0, const double z0,
                                  const double dx1, const double dy1, const double dz1,
                                  const double dx2, const double dy2, const double dz2,
                                  const unsigned int nU = 20, const unsigned int nV = 20);
```

x0,y0,z0 coordinates of one of the corners of the parallelogram,

dx1,dy1,dz1 direction vector from (x_0, y_0, z_0) to one of the adjacent corners,

dx2,dy2,dz2 direction vector to the other adjacent corner,

nU,nV number of integration intervals along the two directions.

The result is given in units of V cm.

4.9. Sensor

The `Sensor` class can be viewed as a composite of components. In order to obtain a complete description of a detector, it is sometimes useful to combine fields from different `Component` objects. For instance, one might wish to use a field map for the electric field, calculate the weighting field using analytic methods, and use a parameterized B field. Superpositions of several electric, magnetic and weighting fields are also possible.

Components are added using

```
void AddComponent(Component* comp);
void AddElectrode(Component* comp, std::string label);
```

While `AddComponent` tells the `Sensor` that the respective `Component` should be included in the calculation of the electric and magnetic field, `AddElectrode` requests the weighting field named `label` to be used for computing the corresponding signal.

To deactivate (or activate) a component after having added it, the function

```
void EnableComponent(const unsigned int i, const bool on);
```

can be used. Components that have been deactivated are not taken into account when calculating the electric field but are not removed from the list. Similarly,

```
void EnableMagneticField(const unsigned int i, const bool on);
```

can be used to deactivate (or activate) the magnetic field of a given component.

To reset the sensor, thereby removing all components and electrodes, use

```
void Clear();
```

The total electric and magnetic fields (sum over all components) at a given position are accessible through the functions `ElectricField` and `MagneticField`. The syntax is the same as for the corresponding functions of the `Component` classes. Unlike the fields, materials cannot overlap. The function `Sensor::GetMedium`, therefore, returns the first valid drift medium found.

The `Sensor` acts as an interface to the transport classes.

For reasons of efficiency, it is sometimes useful to restrict charge transport, ionization and similar calculations to a certain region of the detector. This "user area" can be set by

```
void SetArea(const double xmin, const double ymin, const double zmin,
             const double xmax, const double ymax, const double zmax);
```

xmin, ..., zmax corners of the bounding box within which transport is enabled.

Calling `SetArea()` (without arguments) sets the user area to the envelope of all components (if such an envelope exists).

In addition, the `Sensor` class takes care of signal calculations (Chapter 7).

5. Tracks

The purpose of `Track...` classes is to simulate ionization patterns produced by charged particles traversing the detector.

The type of the primary particle is set by the function

```
void SetParticle(std::string particle);
```

particle name of the particle

Only particles which are sufficiently long lived to leave a track in a detector are considered. A list of the available particles is given in Table 5.1.

The kinematics of the charged particle can be defined by means of a number of equivalent methods:

- the total energy (in eV) of the particle,
- the kinetic energy (in eV) of the particle,
- the momentum (in eV/c) of the particle,
- the (dimension-less) velocity $\beta = v/c$, the Lorentz factor $\gamma = 1/\sqrt{1-\beta^2}$ or the product $\beta\gamma$ of these two variables.

The corresponding functions are

```
void SetEnergy(const double e);
void SetKineticEnergy(const double ekin);
void SetMomentum(const double p);
void SetBeta(const double beta);
void SetGamma(const double gamma);
void SetBetaGamma(const double bg);
```

A track is initialized by means of

```
void NewTrack(const double x0, const double y0, const double z0, const double t0,
              const double dx0, const double dy0, const double dz0);
```

x0, y0, z0 initial position (in cm)

t0 starting time

dx0, dy0, dz0 initial direction vector

The starting point of the track has to be inside an ionizable medium. Depending on the type of `Track` class, there can be further restrictions on the type of `Medium`. If the specified direction vector has zero length, an isotropic random vector will be generated.

After successful initialization, the "clusters" produced along the track can be retrieved using

Table 5.1. Available charged particles.

particle		mass [MeV/c^2]	charge
e	`electron, e-`	0.510998910	-1
e^+	`positron, e+`	0.510998910	$+1$
μ^-	`muon, mu-`	105.658367	-1
μ^+	`mu+`	105.658367	$+1$
π^-	`pion, pi, pi-`	139.57018	-1
π^+	`pi+`	139.57018	$+1$
K^-	`kaon, K, K-`	493.677	-1
K^+	`K+`	493.677	$+1$
p	`proton, p`	938.272013	$+1$
$\overline{p}$	`anti-proton, antiproton, p-bar`	938.272013	-1
d	`deuteron, d`	1875.612793	$+1$

```
const std::vector<Cluster>& GetClusters();
```

In this context, "cluster" refers to the energy loss in a single ionizing collision of the primary charged particle and the secondary electrons produced in this process. The concrete implementation of the cluster objects depends on the `Track` class.

5.1. Heed

The program Heed [44] is an implementation of the photo-absorption ionization (PAI) model. It was written by I. Smirnov. An interface to Heed is available through the class `TrackHeed`.

The `Cluster` objects returned by `TrackHeed::GetClusters` contain the position and time of the ionizing collision (member variables `x, y, z, t`), the transferred energy (member variable `energy`), and a `vector` of `Electron` objects corresponding to the conduction electrons associated to the cluster.

In the following snippet, we iterate over the clusters along a track and the conduction electrons in each cluster.

```
TrackHeed track;
// ...
double x0 = 0., y0 = 0., z0 = 0., t0 = 0.;
track.NewTrack(x0, y0, z0, t0, 0., 0., 0.);
// Loop over the clusters along the track.
for (const auto& cluster : track.GetClusters()) {
  // Loop over the conduction electrons in the cluster.
  for (const auto& electron : cluster.electrons) {
    // Get the coordinates of the electron.
    const double xe = electron.x;
    const double ye = electron.y;
    const double ze = electron.z;
    const double te = electron.t;
  }
}
```

5.1.1. Delta electron transport

Heed simulates the energy degradation of δ electrons and the production of secondary ("conduction") electrons using a phenomenological algorithm described in Ref. [44].

`TrackHeed` retrieves the necessary input parameters - the asymptotic W value (eV) and the Fano factor - from the relevant `Medium` object. If these parameters are set to zero, Heed uses internal default values. The default value for the Fano factor is $F = 0.19$.

The transport of δ electrons can be activated or deactivated using

```
void EnableDeltaElectronTransport();
void DisableDeltaElectronTransport();
```

If δ electron transport is disabled, the number of electrons returned by `GetCluster` is the number of "primary" ionisation electrons, *i. e.* the photo-electrons and Auger electrons. Their kinetic energies and locations are accessible through the function `GetElectron`.

If δ electron transport is enabled (default setting), the function `GetElectron` returns the locations of the "conduction" electrons as calculated by the internal δ transport algorithm of Heed. Since this method does not provide the energy and direction of the secondary electrons, the corresponding parameters in `GetElectron` are not meaningful in this case.

5.1.2. Photon transport

Heed can also be used for simulating x-ray photoabsorption.

```
Cluster TransportPhoton(const double x0, const double y0, const double z0,
                        const double t0, const double e0,
                        const double dx0, const double dy0, const double dz0);
```

x0, y0, z0, t0 initial position and time of the photon

e0 photon energy in eV

dx0, dy0, dz0 direction of the photon

5.1.3. Magnetic fields

If the sensor has a non-zero magnetic field, `TrackHeed` will, by default, take the magnetic field into account for calculating the charged-particle trajectory. In order to explicitly enable or disable the use of the magnetic field in the stepping algorithm, the functions

```
EnableMagneticField();
DisableMagneticField();
```

can be called before simulating a track. Depending on the strength of the magnetic field, it might be necessary to adapt the limits/parameters used in the stepping algorithm in order to obtain a smoothly curved trajectory as, for instance, in the example below.

```
TrackHeed track;
// Get the default parameters.
double maxrange = 0., rforstraight = 0., stepstraight = 0., stepcurved = 0.;
```

```
track.GetSteppingLimits(maxrange, rforstraight, stepstraight, stepcurved);
// Reduce the step size [rad].
stepcurved = 0.02;
track.SetSteppingLimits(maxrange, rforstraight, stepstraight, stepcurved);
```

5.2. SRIM

SRIM[1] is a program for simulating the energy loss of ions in matter. It produces tables of stopping powers, range and straggling parameters that can subsequently be imported in Garfield using the class `TrackSrim`. The function

```
bool ReadFile(const std::string& file)
```

returns `true` if the SRIM output file was read successfully. The SRIM file contains the following data

- a list of kinetic energies at which losses and straggling have been computed;
- average energy lost per unit distance via electromagnetic processes, for each energy;
- average energy lost per unit distance via nuclear processes, for each energy;
- projected path length, for each energy;
- longitudinal straggling, for each energy;
- transverse straggling, for each energy.

These can be visualized using the functions

```
void PlotEnergyLoss();
void PlotRange();
void PlotStraggling();
```

and printed out using the function `TrackSrim::Print()`. In addition to these tables, the file also contains the mass and charge of the projectile, and the density of the target medium. These properties are also imported and stored by `TrackSrim` when reading in the file. Unlike in case of Heed, the particle type therefore does not need to be specified by the user. The user does however need to set the kinetic energy of the projectile.

`TrackSrim` tries to generate individual tracks which statistically reproduce the average quantities calculated by SRIM. Starting with the energy specified by the user, it iteratively

- computes (by interpolating in the tables) the electromagnetic and nuclear energy loss per unit length at the current particle energy,
- calculates a step with a length over which the particle will produce on average a certain number of electrons,
- updates the trajectory based on the longitudinal and transverse scatter at the current particle energy,

[1] Stopping and Range of Ions in Matter, www.srim.org

Table 5.2. Fluctuation models in `TrackSrim`.

Model	Description
0	No fluctuations
1	Untruncated Landau distribution
2	Vavilov distribution (provided the kinematic parameters are within the range of applicability, otherwise fluctuations are disabled)
3	Gaussian distribution
4	Combination of Landau, Vavilov and Gaussian models, each applied in their alleged domain of applicability

- calculates a randomised actual electromagnetic energy loss over the step and updates the particle energy.

This is repeated until the particle has no energy left or leaves the geometry. The model for randomising the energy loss over a step can be set using the function

```
void SetModel(const int m);
```

m fluctuation model to be used (Table 5.2); the default setting is model 4.

The generation of Vavilov distributed random numbers is based on a C++ implementation of the CERNLIB G115 procedures for the fast, approximate calculation of functions related to the Vavilov distribution. The description of the algorithm can be found in Ref. [35].

For sampling the energy loss, `TrackSrim` needs the electron density of the target material. By default it is retrieved from the relevant `Medium` object (and scaled to the mass density given in the SRIM output file). Alternatively, the user can specify the effective atomic number Z and mass number A of the target using `TrackSrim::SetAtomicMassNumbers`.

Transverse and longitudinal straggling can be switched on or off using

```
void EnableTransverseStraggling(const bool on);
void EnableLongitudinalStraggling(const bool on);
```

If energy loss fluctuations are used, longitudinal straggling should be disabled. By default, transverse straggling is switched on and longitudinal straggling is switched off.

SRIM is aimed at low energy nuclear particles which deposit large numbers of electrons in a medium. The grouping of electrons to a cluster is therefore somewhat arbitrary. By default, `TrackSrim` will adjust the step size such that there are on average 100 clusters on the track. If the user specifies a target cluster size, using

```
void SetTargetClusterSize(const int n);
```

the step size will be chosen such that a cluster comprises on average `n` electrons. Alternatively, if the user specifies a maximum number of clusters, using

```
void SetClustersMaximum(const int n);
```

the step size will be chosen such that on average there are `n / 2` clusters on the track.

To calculate the number of electrons for given amount of deposited energy, `TrackSrim` needs the work function W (in eV) and the Fano factor of the target material. By default, `TrackSrim` retrieves these parameters from the relevant `Medium` object. Alternatively, they can be set explicitly using `TrackSrim::SetWorkFunction` and `TrackSrim::SetFanoFactor`.

The `Cluster` objects returned by `TrackSrim::GetClusters` contain the location and time of the cluster (variables `x, y, z, t`), the energy spent to make the cluster (`energy`), the ion energy when the cluster was created (`kinetic`), and the number of electrons in the cluster (`n`).

5.3. TRIM

TRIM[2] is a Monte Carlo simulation program from the same collection of software packages as SRIM. It simulates individual ion trajectories in a target (which can be made of several layers) and the processes following the ion's energy loss (recoil cascades, displacement damage, *etc.*). TRIM typically produces a number of output files. One of them is a file called `EXYZ.txt` which lists the position and electronic stopping power for each simulated ion at regular steps in the ion's kinetic energy. The energy interval is set by the user.

The class `TrackTrim` allows one to import these data in Garfield and use them for simulating tracks. The function for reading ion trajectories from an `EXYZ.txt` file is

```
bool ReadFile(const std::string& file, const unsigned int nIons = 0,
              const unsigned int nSkip = 0);
```

file name/path of the `EXYZ.txt` file to be loaded,

nIons number of ion trajectories to be loaded from the file,

nSkip number of ion trajectories to be skipped at the beginning of the file.

If the value of `nIons` is zero, `TrackTrim` will import all ion trajectories included in the file.

When the function `NewTrack` is called, `TrackTrim` will generate clusters according to one of the ion trajectories imported from `EXYZ.txt` (starting with the first one). Each cluster will correspond to one energy interval. The number of clusters along a track and the deposited energy per cluster is therefore controlled by the energy interval specified when running TRIM. The number of electrons in a cluster is sampled using an algorithm that reproduces the requested work function (W value) and Fano factor. At the next call to `NewTrack`, `TrackTrim` moves to the next ion trajectory in the list (if it reaches the end of the list, it rewinds to the first one).

The variables contained in the `Cluster` objects returned by `TrackTrim::GetClusters` are the same as for `TrackSrim`.

5.4. Degrade

The class `TrackDegrade` simulates ionisation by primary electrons in a gas and the subsequent degradation of δ electrons, Auger electrons and photoelectrons, using an interface to the program Degrade, developed by S. Biagi. Degrade has many commonalities with Magboltz, in particular the database of electron-atom/molecule cross-sections.

[2] TRansport of Ions in Matter

While the Degrade program can also be used for simulating X-rays, β decay and double β decay, but these features are not (yet) accessible through the `TrackDegrade` interface at the moment. It is also worth noting, that the current version of `TrackDegrade` does not consider the electric and magnetic fields in the detector.

The `Cluster` objects returned by `TrackDegrade::GetClusters` contain the position and time of the ionizing collision (member variables `x, y, z, t`), a `vector` of `Electron` objects corresponding to the thermalised electrons associated to the cluster. In addition, it also contains a `vector` of the δ electrons and Auger electrons.

```
TrackDegrade track;
// ...
double x0 = 0., y0 = 0., z0 = 0., t0 = 0.;
double dx0 = 1., dy0 = 0., dz0 = 0.
track.NewTrack(x0, y0, z0, t0, dx0, dy0, dz0);
// Loop over the clusters along the track.
for (const auto& cluster : track.GetClusters()) {
  // Loop over the thermalised electrons in the cluster.
  for (const auto& electron : cluster.electrons) {
    // Get the coordinates and kinetic energy of the electron.
    double xe = electron.x;
    double ye = electron.y;
    double ze = electron.z;
    double te = electron.t;
    double ee = electron.energy;
  }
}
```

By default, electrons are tracked until their kinetic energy falls below 2 eV. This threshold can be changed using the function

```
void SetThresholdEnergy(const double ethr);
```

If the function

```
void StoreExcitations(const bool on = true, const double ethr);
```

is called prior to `NewTrack`, the excitations (with excitation energy above `ethr`) produced by the primary and secondary electrons are also stored in the `Cluster` object.

6. Charge transport

On a phenomenological level, the drift of charge carriers under the influence of an electric field $\mathbf{E}$ and a magnetic field $\mathbf{B}$ is described by the first order equation of motion

$$\dot{\mathbf{r}} = \mathbf{v}_{\mathrm{d}}(\mathbf{E}\left(\mathbf{r}\right), \mathbf{B}\left(\mathbf{r}\right)), \tag{6.1}$$

where $\mathbf{v}_{\mathrm{d}}$ is the drift velocity. For the solution of (6.1), two methods are available in Garfield++:

- Runge-Kutta-Fehlberg integration (`DriftLineRKF`), and
- Monte Carlo integration (`AvalancheMC`).

For accurate simulations of electron trajectories in small-scale structures (with characteristic dimensions comparable to the electron mean free path), and also for detailed calculations of ionisation and excitation processes, transporting electrons on a microscopic level – *i. e.* based on the second-order equation of motion – is the method of choice. Microscopic tracking of electrons is dealt with by the class `AvalancheMicroscopic`.

6.1. Runge-Kutta-Fehlberg integration

This method, implemented in the class `DriftLineRKF`, calculates a drift line by iterating over the following steps.

1. Given a starting point $\mathbf{x}_0$, the velocity at the starting point, and a time step Δt, compute two estimates of the step to the next point on the drift line,

   - $\Delta v_{\mathrm{I}} = \sum_{k=0}^{2} C_{\mathrm{I},k}\mathbf{v}_d\left(\mathbf{x}_k\right)$, accurate to second order, and
   - $\Delta v_{\mathrm{II}} = \sum_{k=0}^{3} C_{\mathrm{II},k}\mathbf{v}_d\left(\mathbf{x}_k\right)$, accurate to third order.

   These two estimates are based on the drift velocity at the starting point and the velocity at three new locations

   $$\mathbf{x}_k = \mathbf{x}_0 + \Delta t \sum_{i=0}^{k-1} \beta_{k,i}\mathbf{v}_d\left(\mathbf{x}_i\right).$$

   The values of the coefficients are shown in Table 6.1.

Table 6.1. Coefficients of the Runge-Kutta-Fehlberg formula [46].

k \ i	$\beta_{k,i}$ 0	1	2
1	$\frac{1}{4}$		
2	$\frac{-189}{800}$	$\frac{729}{800}$	
3	$\frac{214}{891}$	$\frac{1}{33}$	$\frac{650}{891}$

k	$C_{I,k}$	$C_{II,k}$
0	$\frac{214}{891}$	$\frac{533}{2106}$
1	$\frac{1}{33}$	
2	$\frac{650}{891}$	$\frac{800}{1053}$
3		$\frac{-1}{78}$

2. The time step is updated by comparing the second and third order estimates with the requested accuracy
$$\Delta t' = \sqrt{\frac{\varepsilon \Delta t}{|\Delta v_{\mathrm{I}} - \Delta v_{\mathrm{II}}|}}.$$
3. The step is repeated if
   - the time step shrinks by more than a factor five,
   - the step size exceeds the maximum step length allowed (if such a limit is set),
4. The position is updated with the second order estimate.
5. The velocity is updated according to the end-point velocity of the step, which is one of the three velocity vectors that were computed under 1.

The initial time step is estimated using $\Delta t = \varepsilon / |\mathbf{v}_d(\mathbf{x}_0)|$. The parameter ε used in this initial estimate and in step 2 of the above algorithm, can be set using

```
void SetIntegrationAccuracy(const double eps);
```

When traversing a large area with a very smooth field, the step size becomes large. If this is not desired, for instance because there is a fine structure behind the smooth area, then one should limit the step size using

```
void SetMaximumStepSize(const double ms);
```

The maximum step size is recommended to be of order $1/10 - 1/20$ of the distance to be traversed. The default behaviour (no limit on the step size) can be reinstated using

```
void UnsetMaximumStepSize();
```

By default, the drift line calculation is aborted if the drift line makes a bend sharper than 90°. Such bends rarely occur in smooth fields, the most common case is a drift line that tries to cross a saddle point. This check can be switched on or off using

```
void RejectKinks(const bool on = true);
```

During the evaluation of the velocities $\mathbf{v}_d(\mathbf{x}_k)$ for the next step, checks are made to ensure that none of the probe points $\mathbf{x}_k$ are outside the active area and that no wire was crossed. If a wire has been crossed during the step, another algorithm for stepping towards a wire takes over. The stepping towards the wire also starts when the distance between the particle position and a wire is less than n times the wire radius. The factor n ("trap radius") can be set in `ComponentAnalyticField` when defining the wire. If one of the points along the step is outside the active area, the drift line is terminated by doing a last linear step towards the boundary.

Drift line calculations are started using

```
bool DriftElectron(const double x0, const double y0, const double z0, const double t0);
bool DriftHole(const double x0, const double y0, const double z0, const double t0);
bool DriftIon(const double x0, const double y0, const double z0, const double t0);
```

x0, y0, z0, t0 initial position and time

The function

```
bool DriftPositron(const double x0, const double y0, const double z0, const double t0);
```

computes an electron drift line, but assuming that the electron has positive charge (which can be useful for determining isochrons). Analogously, the function

```
bool DriftNegativeIon(const double x0, const double y0, const double z0, const double t0);
```

computes an ion drift line, assuming that the ion has negative charge.

After calculating a drift line, the multiplication and loss factors

$$\exp\left(\int \alpha \mathrm{d}s\right), \exp\left(-\int \eta \mathrm{d}s\right)$$

along the drift line can be obtained using

```
void GetGain(const double eps = 1.e-4);
void GetLoss(const double eps = 1.e-4);
```

eps parameter determining the accuracy of the integration.

Both functions use an adaptive Simpson-style integration.

Similarly, the function

```
void GetArrivalTimeSpread(const double eps = 1.e-4);
```

computes the σ of the arrival time distribution by integrating quadratically the ratio of longitudinal diffusion coefficient and drift velocity over the drift line,

$$\sigma^2 = \int \left(D_L/v_\mathrm{D}\right)^2 \mathrm{d}s.$$

The points along the most recent drift line are accessible through the functions

```
size_t GetNumberOfDriftLinePoints() const;
void GetDriftLinePoint(const size_t i, double& x, double& y, double& z, double& t) const;
```

i index of the point.

x, y, z, t coordinates and time of the point.

If one is simply interested in the end point and status flag of the current drift line, the function

```
void GetEndPoint(double& x, double& y, double& z, double& t, int& st) const;
```

can be used. A list of the status codes is given in Table 6.2.

By default, the induced current is calculated for each drift line. This can be activated or deactivated using

```
void EnableSignalCalculation(const bool on = true);
```

For electron drift lines, multiplication is by default taken into account in the signal calculation. For this purpose, after calculating a drift line the number of electrons and ions at each point of the line is calculated by integrating the Townsend and attachment coefficient along the line. For a given starting point, the number of electrons at the end of the drift line is thus given by

$$n_e = \exp\left(\int (\alpha - \eta)\,\mathrm{d}s\right).$$

The multiplication factor can be set explicitly using

```
void SetGainFluctuationsFixed(const double gain = -1.);
```

gain multiplication factor to be used. If the provided value is < 1, the multiplication factor obtained by integrating $\alpha - \eta$ along the drift line is used instead (as is the default).

Using

```
EnableAvalanche(const bool on);
```

the simulation of avalanches for electron drift lines can be switched on or off.

In order to take fluctuations of the avalanche size into account in the signal calculation, the number of electrons in the avalanche can be sampled from a Pólya distribution [1]

$$\overline{n} P_n = \frac{(\theta+1)^{\theta+1}}{\Gamma(\theta+1)} \left(\frac{n}{\overline{n}}\right)^{\theta} \mathrm{e}^{-(\theta+1)n/\overline{n}},$$

where P_n is the probability that the avalanche comprises n electrons, $\overline{n}$ is the mean avalanche size, and θ is a parameter controlling the shape of the distribution. For $\theta = 0$ one obtains an exponential distribution, while with increasing θ the distribution becomes more and more "rounded". The simulation of gain fluctuations can be enabled using the function

```
void SetGainFluctuationsPolya(const double theta, const double mean = -1.);
```

theta shape parameter θ of the Pólya distribution.

mean mean avalanche size $\overline{n}$. If the provided value is < 1, the multiplication factor obtained by integrating $\alpha - \eta$ along the drift line is used instead (as is the default).

By default, `DriftLineRKF` also simulates the ion tail, *i. e.* the signal due to the ions created in an avalanche, provided that the drift medium has ion mobility data. Using the function

```
EnableIonTail(const bool on);
```

the simulation of the ion tail can be switched on or off explicitly.

The signal component due to negative ions created in the avalanche is not simulated by default, and needs to be requested explicitly using

```
EnableNegativeIonTail(const bool on = true);
```

6.2. Monte Carlo integration

In the class `AvalancheMC`, Eq. (6.1) is integrated in a stochastic manner:

- a step of length $\Delta s = v_{\mathrm{d}} \Delta t$ in the direction of the drift velocity $\mathbf{v}_{\mathrm{d}}$ at the local electric and magnetic field is calculated (with either the time step Δt or the distance Δs being specified by the user);
- a random diffusion step is sampled from three uncorrelated Gaussian distributions with standard deviation $\sigma_L = D_L\sqrt{\Delta s}$ for the component parallel to the drift velocity and standard deviation $\sigma_T = D_T\sqrt{\Delta s}$ for the two transverse components;
- the two steps are added vectorially and the location is updated.

The functions for setting the step size are

```
void SetTimeSteps(const double d = 0.02);
void SetDistanceSteps(const double d = 0.001);
void SetCollisionSteps(const int n = 100);
```

In the first case the integration is done using fixed time steps (default: 20 ps), in the second case using fixed distance steps (default: 10 µm). Calling the third function instructs the class to do the integration with exponentially distributed time steps with a mean equal to a multiple of the "collision time"

$$\tau = \frac{m v_d}{qE}.$$

The third method is activated by default.

Instead of making simple straight-line steps (using the drift velocity vector at the starting point of a step), the end point of a step can be calculated using a (second-order) Runge-Kutta-Fehlberg method. This feature can be activated using

```
void EnableRKFSteps(const bool on = true);
```

The average velocity $\mathbf{v}$ over a step Δt is then calculated using

$$\mathbf{v} = \sum_{k=0}^{2} C_k \mathbf{v}_d\left(\mathbf{x}_k\right), \quad \mathbf{x}_1 = \mathbf{x}_0 + \Delta t \beta_{1,0} \mathbf{v}_d\left(\mathbf{x}_0\right), \quad \mathbf{x}_2 = \mathbf{x}_0 + \Delta t\left(\beta_{2,0}\mathbf{v}_d\left(\mathbf{x}_0\right) + \beta_{2,1}\mathbf{v}_d\left(\mathbf{x}_1\right)\right),$$

where $\mathbf{x}_0$ is the starting point of the step. The values of the coefficients C_k and $\beta_{k,i}$ are given in Table 6.1.

If the electric field or drift speed is zero, the algorithm switches to diffusion-only steps based on the low-field mobility.

Drift line calculations are started using

```
bool DriftElectron(const double x, const double y, const double z, const double t);
bool DriftHole(const double x, const double y, const double z, const double t);
bool DriftIon(const double x, const double y, const double z, const double t);
```

x, y, z, t initial position and time

The trajectory can be retrieved using

```
size_t GetNumberOfDriftLinePoints() const;
void GetDriftLinePoint(const size_t i, double& x, double& y, double& z, double& t);
```

The calculation of an avalanche initiated by an electron, a hole or an electron-hole pair is done using

```
bool AvalancheElectron(const double x, const double y, const double z,
                       const double t, const bool hole = false);
bool AvalancheHole(const double x, const double y, const double z,
                   const double t, const bool electron = false);
bool AvalancheElectronHole(const double x, const double y, const double z,
                           const double t);
```

The flags `hole` and `electron` specify whether the drift and multiplication of the holes/ions (electrons) created in the avalanche should be simulated.

The trajectories of the electrons in the avalanche can be retrieved using

```
const std::vector<EndPoint>& GetElectrons();
```

where `EndPoint` is a `struct` containing an integer status code indicating why the tracking of the electron was stopped and a vector of points along the drift line:

```
struct EndPoint {
  int status;
  std::vector<Point> path;
};
```

Analogous functions are available for holes and (positive and negative) ions.

For debugging purposes, attachment and diffusion can be switched off using

```
void DisableAttachment();
void DisableDiffusion();
```

A time interval can be set using

```
void SetTimeWindow(const double t0, const double t1);
```

t0 lower limit of the time window

t1 upper limit of the time window

If a time window is set, only charge carriers with a time coordinate $t \in [t_0, t_1]$ are tracked. If the time coordinate of a particle crosses the upper limit, it is stopped and assigned the status code -17. Slicing the calculation into time steps can be useful for instance for making a movie of the avalanche evolution or for calculations involving space charge. Another useful function for that purpose is

```
bool ResumeAvalanche(const bool electron = true, const bool hole = true);
```

electron,hole flags to switch off the electron (hole) component of the avalanche

which instructs `AvalancheMC` to continue the avalanche simulation from the most recent set of end points. Before calling `ResumeAvalanche`, one can "manually" add electrons, holes, or ions to the list of charge carriers to be transported.

```
void AddElectron(const double x, const double y, const double z, const double t);
void AddHole(const double x, const double y, const double z, const double t);
void AddIon(const double x, const double y, const double z, const double t);
```

The time window can be removed using

```
void UnsetTimeWindow();
```

Using the function

```
void EnableProjectedPathIntegration(const bool on = true);
```

one can request the Townsend and attachment coefficients to be projected onto the local drift velocity vector when integrating them over drift path segments. By default, this feature is switched on. The function

```
void EnableAvalancheSizeLimit(const unsigned int size);
```

sets an upper limit to the number of electrons in an avalanche can be imposed.

6.3. Microscopic tracking

In the microscopic tracking approach – implemented at present only for electrons – a particle is followed from collision to collision. As input, it requires a table of the collision rates $\tau_i^{-1}(\epsilon)$ for each scattering process i as function of the electron energy ϵ. For gases, these data are provided by the class `MediumMagboltz`. Between collisions, an electron is traced on a classical vacuum trajectory according to the local electric (and optionally magnetic) field. The duration Δt of a free-flight step is controlled by the total collision rate $\tau^{-1}(\epsilon) = \sum_i \tau_i^{-1}(\epsilon)$. The sampling of Δt of a free-flight step is done using the "null-collision" method [43], which accounts for the change in electron energy during the step. After the step, the energy, direction, and position of the electron are updated and the scattering process to take place is sampled based on the relative collision rates at the new energy ϵ'. The energy and direction of the electron are subsequently updated according to the type of collision.

In Garfield++, the microscopic tracking method is implemented in the class `AvalancheMicroscopic`. A calculation is started by means of

```
void AvalancheElectron(const double x, const double y, const double z,
                       const double t, const double e,
                       const double dx = 0., const double dy = 0., const double dz = 0.);
```

x, y, z, t initial position and time

e initial energy (eV)

Table 6.2. Status codes for the termination of drift lines.

status code	meaning
-1	particle left the drift area
-3	calculation abandoned (error, should not happen)
-5	particle not inside a drift medium
-7	attachment
-8	sharp kink (only for RKF)
-16	energy below transport cut
-17	outside the time window

dx, dy, dz initial direction

If the length of the direction vector is zero, the initial direction is randomized.

After the calculation is finished, the number of electrons (`ne`) and ions (`ni`) produced in the avalanche can be retrieved using

```
void GetAvalancheSize(int& ne, int& ni);
```

Information about the "history" of each avalanche electron can be retrieved using

```
const std::vector<Electron>& GetElectrons();
```

where `Electron` is a `struct` containing an integer status code indicating why the simulation of the electron drift line was terminated, a vector of points along the trajectory and the total path length.

```
struct Electron {
  int status = 0;
  std::vector<Point> path;
  double pathLength = 0.;
};
```

A list of status codes is given in Table 6.2. The `Point` objects contain the location (`x, y, z`), time (`t`), kinetic energy (`energy`), and direction vector (`kx, ky, kz`) of the electron.

The function

```
bool DriftElectron(const double x, const double y, const double z, const double t,
                   const double e, const double dx, const double dy, const double dz);
```

traces only the initial electron but not the secondaries produced along its drift path (the input parameters are the same as for `AvalancheElectron`).

The electron energy distribution can be extracted in the following way:

```
AvalancheMicroscopic aval;
// Make a histogram (100 bins between 0 and 100 eV).
TH1F hEnergy("hEnergy", "Electron energy", 100, 0., 100.);
// Pass the histogram to the avalanche class.
```

```
aval.EnableElectronEnergyHistogramming(&hEnergy);
```

After each collision, the histogram is filled with the current electron energy.

If the sensor has a non-zero magnetic field, `AvalancheMicroscopic` will by default use a more complicated stepping algorithm which takes the effect of the B field on the electron trajectory into account. In order to explicitly switch the use of magnetic fields on or off one can use the function

```
void EnableMagneticField(const bool on);
```

The default stepping algorithm evaluates the electric field only at the start of a free-flight step, assuming that it is constant throughout the step. This is an exact solution for uniform fields, and usually a good approximation if the electric field does not change drastically over a step. To improve the accuracy of the free-flight calculation, in particular at low gas pressure, a stepping algorithm based on Runge-Kutta-Nyström integration can be requested using

```
void EnableRKNSteps(const bool on = true);
```

The error tolerance and minimum step size in the algorithm can be set using

```
void SetRKNTolerance(const double sTol = 1.e-10,
                     const double sMinStep = 1.e-5);
```

Using

```
void EnableAvalancheSizeLimit(const unsigned int size);
```

an upper limit to the size of an electron avalanche can be imposed. After the avalanche has reached the specified max. size, no further secondaries are added to the stack of electrons to be transported.

Like in `AvalancheMC` a time window can be set/unset using

```
void SetTimeWindow(const double t0, const double t1);
void UnsetTimeWindow();
```

An energy threshold for transporting electrons can be applied using

```
void SetElectronTransportCut(const double cut);
```

cut energy threshold (in eV)

The tracking of an electron is aborted if its energy falls below the transport cut. This option can be useful for δ electron studies in order to stop the calculation once the energy of an electron is below the ionization potential of the gas. The transport cut can be removed by setting the threshold to a negative value. By default, no cut is applied.

In order to extract information from the avalanche on a collision-by-collision basis, a number of callback functions (“user handles”) are available.

```
void SetUserHandleStep(void (*f)(double x, double y, double z,
                                 double t, double e,
                                 double dx, double dy, double dz,
                                 bool hole));
void UnsetUserHandleStep();
void SetUserHandleCollision(void (*f)(double x, double y, double z, double     t,
                                      int type, int level, Medium* m,
                                      double e0, double e1,
                                      double dx0, double dy0, double dz0,
                                      double dx1, double dy1, double dz1));
void UnsetUserHandleCollision();
void SetUserHandleAttachment(void (*f)(double x, double y, double z,
                                       double t,
                                       int type, int level, Medium* m));
void UnsetUserHandleAttachment();
void SetUserHandleInelastic(void (*f)(double x, double y, double z,
                                      double t,
                                      int type, int level, Medium* m));
void UnsetUserHandleInelastic();
void SetUserHandleIonisation(void (*f)(double x, double y, double z,
                                       double t,
                                       int type, int level, Medium* m));
void UnsetUserHandleIonisation();
```

The function specified in `SetUserHandleStep` is called prior to each free-flight step. The parameters passed to this function are

x, y, z, t position and time,

e energy before the step

dx, dy, dz direction,

hole flag indicating whether the particle is an electron or a hole.

The "user handle" function set via `SetUserHandleCollision` is called every time a real collision (as opposed to a null collision) occurs. The "user handle" functions for attachment, ionisation, and inelastic collisions are called each time a collision of the respective type occurs. In this context, inelastic collisions also include excitations. The parameters passed to these functions are

x, y, z, t the location and time of the collision,

type the type of collision (see Table 3.2),

level the index of the cross-section term (as obtained from the `Medium`),

m a pointer to the current `Medium`.

In the function set using `SetUserHandleCollision`, the energy and the direction vector before and after the collision are available in addition.

In the following example we want to retrieve all excitations happening in the avalanche.

```
void userHandle(double x, double y, double z, double t,
                int type, int level, Medium* m) {
```

```
  // Check if the collision is an excitation.
  if (type != 4) return;
  // Do something (e. g. fill a histogram, simulate the emission of a VUV photon)
  ...
}

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

  // Setup gas, geometry, and field
  ...
  AvalancheMicroscopic aval;
  ...
  aval.SetUserHandleInelastic(userHandle);
  double x0 = 0., y0 = 0., z0 = 0., t0 = 0.;
  double e0 = 1.;
  aval.AvalancheElectron(x0, y0, z0, t0, e0, 0., 0., 0.);
  ...

}
```

6.4. Visualizing drift lines

For plotting drift lines and tracks the class `ViewDrift` can be used. After attaching a `ViewDrift` object to a transport class, *e. g.* using

```
void AvalancheMicroscopic::EnablePlotting(ViewDrift* view, const size_t nColl = 100);
void AvalancheMC::EnablePlotting(ViewDrift* view);
void DriftLineRKF::EnablePlotting(ViewDrift* view);
void Track::EnablePlotting(ViewDrift* view);
```

it will store the trajectories which are calculated by the respective transport class.

To actually draw the trajectories, the function

```
void ViewDrift::Plot();
```

needs to be called.

In case of `AvalancheMicroscopic`, the second argument of `EnablePlotting` (`nColl`) sets the number of collisions to be skipped between successive points on the plot (by default, every 100th collision is plotted). Note that this setting does not affect the transport of the electron as such, the electron is always tracked rigorously through single collisions.

6.5. Visualizing isochrons

For drift chambers, it is useful to determine the contours of equal drift time to a wire. These so-called isochrons can be calculated and drawn using the class `ViewIsochrons`. The component or sensor from which to retrieve the field is set using

```
void SetComponent(Component* c);
```

or

```
void SetSensor(Sensor* s);
```

The function

```
void DriftElectrons(const bool positive = false);
```

instructs the class to compute isochrons using electron drift lines. If the flag `positive` is set to `true`, the electrons are drifted with positive charge, which is useful for calculating isochrons of wires that attract electrons.

By calling

```
void DriftIons(const bool negative = false);
```

one requests drift lines of (positive or negative) ions.

The calculation of the drift lines and equal time contours and their visualization is done by the function

```
void PlotIsochrons(const double tstep,
    const std::vector<std::array<double, 3> >& points, const bool reverse = false,
    const bool colour = false, const bool markers = false, const bool plotDriftLines = true);
```

tstep time interval between isochron lines,

points list of starting points from which to simulate drift lines,

reverse flag to measure the drift time from the end points of the drift lines (`true`) or from the starting points (`false`),

colour requests drawing the contour lines using the currently active colour palette,

markers flag to draw markers at the points on the isochrons (`true`) or draw the isochron as lines (`false`),

plotDriftLines requests plotting of the drift lines together with the isochrons.

The calculation of the drift lines is done using `DriftLineRKF`.

The appearance of the isochrons is affected by a number of additional parameters. By default, the algorithm tries to order the points at equal time such that the isochrons appear as reasonably smooth lines. This sorting step can be switched off or on using

```
void EnableSorting(const bool on = true);
```

When an isochron appears to be more or less circular, its points are ordered by increasing angle with respect to the centre of gravity. If the isochron, on the other hand, seems to be more or less linear, the points are ordered along the longest principal axis of the distribution. Whether the set is circular or linear is decided by computing the RMS in the two principal axes of the point distribution. If the ratio of these two numbers exceeds a threshold then the isochron is assumed to be linear, otherwise circular. This parameter (which defaults to 3) can be set using

```
void SetAspectRatioSwitch(const double ar);
```

Isochrons that appear to be circular are closed if the largest distance between two points does not exceed a certain fraction of the total length of the isochron. This threshold (initially set to 0.2) can be modified using

```
void SetLoopThreshold(const double thr);
```

Points on an isochron are only joined if they are less than a certain fraction away from each other on the screen. Points that can not be connected are shown by a marker. This fraction (initial value: 0.2) can be set using

```
void SetConnectionThreshold(const double thr);
```

By default, points on an isochron are also not joined if their connecting line crosses a drift line. This feature can be switched on or off using

```
void CheckCrossings(const bool on = true);
```

7. Signals

Signals are calculated using the Shockley-Ramo theorem [42, 32]. The current $i(t)$ induced by a particle with charge q at a position $\mathbf{r}$ moving at a velocity $\mathbf{v}$ is given by

$$i(t) = -q\mathbf{v} \cdot \mathbf{E}_w(\mathbf{r}), \tag{7.1}$$

where $\mathbf{E}_w$ is the so-called weighting field for the electrode to be read out and the charge induced by particle moving from $\mathbf{r}_1$ to $\mathbf{r}_2$ is given by

$$\int_{t_1}^{t_2} i(t)\,\mathrm{d}t = q\left[\phi_w(\mathbf{r}_2) - \phi_w(\mathbf{r}_1)\right], \tag{7.2}$$

where ϕ_w is the weighting potential.

The basic steps for calculating the current induced by the drift of electrons and ions/holes are:

1. Prepare the weighting field and/or weighting potential for the electrode to be read out. This step depends on the field calculation technique (*i. e.* the type of `Component`) that is used (see Chapter 4).
2. Tell the `Sensor` that you want to use this weighting field/potential for the signal calculation.

   ```
   void Sensor::AddElectrode(Component* cmp, std::string label);
   ```

   where `cmp` is a pointer to the `Component` which calculates the weighting field/potential, and `label` (in our example `"readout"`) is the name you have assigned to the weighting field/potential in the previous step.
3. Set the binning for the signal calculation.

   ```
   void Sensor::SetTimeWindow(const double tmin, const double tstep, const int nbins);
   ```

   The first parameter in this function is the lower time limit (in ns), the second one is the bin width (in ns), and the last one is the number of time bins.
4. The `Sensor` then records and accumulates the signals of all avalanches and drift lines which are simulated.
5. The calculated signal can be retrieved using

   ```
   double Sensor::GetSignal(const std::string label, const int bin);
   double Sensor::GetElectronSignal(const std::string label, const int bin);
   double Sensor::GetIonSignal(const std::string label, const int bin);
   ```

The functions `GetElectronSignal` and `GetIonSignal` return the signal induced by negative and positive charges, respectively. `GetSignal` returns the sum of both electron and hole signals.

6. After the signal of a given track is finished, call

```
void Sensor::ClearSignal();
```

to reset the signal to zero.

The method

```
void Sensor::PlotSignal(const std::string& label, TPad* pad);
```

plots the (total) signal for the electrode with identifier `label` on a pad. Internally, this method uses the class `ViewSignal`. By default, the function

```
void ViewSignal::PlotSignal(const std::string& label,
                            const std::string& optTotal = "t",
                            const std::string& optPrompt = "",
                            const std::string& optDelayed = "",
                            const bool same = false);
```

produces the same plot as `Sensor::PlotSignal`. The three option strings can be used for specifying the components of the signal to be included in the plot. If `optTotal` is set to `tei`, the electron-induced signal, the signal induced by ions (or holes) and the sum of the two will be shown. Analogously, the option strings `optPrompt` and `optDelayed` define which components (if any) of the prompt and delayed part of the signal should be shown.

As an illustration of the above recipe consider the following example.

```
// Electrode label
const std::string label = "readout";
// Setup the weighting field.
// In this example we use a FEM field map.
ComponentAnsys123 fm;
...
fm.SetWeightingField("WPOT.lis", label);

Sensor sensor(&fm);
sensor.AddElectrode(&fm, label);
// Setup the binning (0 to 100 ns in 100 steps).
const double tStart =   0.;
const double tStop  = 100.;
const int nSteps = 100;
const double tStep = (tStop - tStart) / nSteps;

AvalancheMicroscopic aval(&sensor);
// Calculate some drift lines.
...
// Plot the induced current.
TCanvas* cS = new TCanvas("cS", "Induced␣current", 600, 600);
sensor.PlotSignal(label, cS);
```

All three transport classes (`DriftLineRKF`, `AvalancheMC`, `AvalancheMicroscopic`) offer the choice between using the weighting potential or the weighting field for computing the induced current. The method to be used can be selected using

```
void DriftLineRKF::UseWeightingPotential(const bool on = true);
void AvalancheMC::UseWeightingPotential(const bool on = true);
void AvalancheMicroscopic::UseWeightingPotential(const bool on = true);
```

The weighting potential method (which is the default) should be used if one wants to ensure that the integral of the current is equal to the collected charge. It takes ϕ_w at the start and end of each step $j \to j+1$ along the drift path and calculates the average current $q\Delta\phi_w / \left(t_{j+1} - t_j\right)$. For electron drift lines calculated using `DriftLineRKF`, q is weighted by the avalanche size at this drift line step.

The implementations of the weighting field method are slightly different for `DriftLineRKF`, `AvalancheMC`, and `AvalancheMicroscopic`. For drift lines calculated using `AvalancheMicroscopic`, the signal is assumed to be constant between subsequent drift line points and the average velocity $\mathbf{v} = \left(\mathbf{x}_{j+1} - \mathbf{x}_j\right) / \left(t_{j+1} - t_j\right)$ along a drift line segment is used. By default, the weighting field is evaluated at the mid-point of a drift line segment. Using

```
void AvalancheMicroscopic::EnableWeightingFieldIntegration(const bool on = true);
```

one can request 6-point Gaussian integration of the weighting field over a drift line segment.

For drift lines calculated using `DriftLineRKF` or `AvalancheMC`, the times t_j, coordinates $\mathbf{r}_j$ and drift velocities $\mathbf{v}_j$ at each point along the drift line are taken and the induced current

$$i_j = -q\mathbf{E}_w\left(\mathbf{r}_j\right) \cdot \mathbf{v}_j$$

at these points is computed. In order to calculate the average current in each time bin, the array of $\left(t_j, i_j\right)$ is interpolated (linearly) and then integrated using Simpson's rule over $2n_{\text{avg}} + 1$ points. The parameter n_{avg} defaults to 2 for `DriftLineRKF` and 1 for `AvalancheMC` and can be set using

```
void DriftLineRKF::SetSignalAveragingOrder(const unsigned int navg);
void AvalancheMC::SetSignalAveragingOrder(const unsigned int navg);
```

7.1. Readout electronics

In order to model the signal-processing by the front-end electronics, the "raw signal" – *i.e.* the induced current – can be convoluted with a so-called "transfer function" (often also referred to as delta response function). The transfer function to be applied can be set using

```
void Sensor::SetTransferFunction(std::function<double(double)> f);
```

where `f` is a function provided by the user which takes a `double` as an argument and returns a `double`, or using

```
void Sensor::SetTransferFunction(const std::vector<double>& times,
                                 const std::vector<double>& values);
```

in which case the transfer function will be calculated by interpolation of the values provided in the table. A third option is to use a predefined expression, implemented in the helper class `Shaper`. Its constructor,

```
Shaper(const unsigned int n, const double tau, const double g, std::string shaperType);
```

takes four arguments: n is the order of the shaper, τ is the time constant, g is the gain factor, and `shaperType` is either `"unipolar"` or `"bipolar"`. In the first case (unipolar shaper), the transfer function is given by

$$f(t) = g \exp(n) \left(\frac{t}{t_p}\right)^n \exp(-t/\tau), \qquad t_p = n\tau,$$

while for a bipolar shaper the expression

$$f(t) = g \frac{\exp(r)}{\sqrt{n}} \left(n - \frac{t}{\tau}\right) \left(\frac{t}{t_p}\right)^{n-1} \exp(-t/\tau), \qquad t_p = r\tau, \qquad r = n - \sqrt{n}.$$

is used. The normalization of these expressions is chosen such that the value of the transfer function at the peaking time $t = t_p$ is unity. In order to use a transfer function provided by a `Shaper` class, one should call

```
Sensor::SetTransferFunction(Shaper& shaper);
```

The presently stored signal can be convoluted with the transfer function (specified using one of the methods above) using

```
bool Sensor::ConvoluteSignal(const std::string& label);
```

label name of the electrode

The function

```
bool Sensor::ConvoluteSignals();
```

convolutes the signals of all electrodes with the transfer function.

As an example, consider the following transfer function

$$f(t) = \frac{t}{\tau} \exp(1 - t/\tau), \qquad \tau = 25\,\text{ns},$$

i.e. a unipolar shaper with $n = 1$. The two code snippets below illustrate different methods for applying this transfer function to the induced signal. In the first one, we pass a pointer to a C-style function to the `Sensor`.

```
double transfer(double t) {
  constexpr double tau = 25.;
  return (t / tau) * exp(1 - t / tau);
}

int main(int argc, char* argv[]) {
```

```
  // Setup component, media, etc.
  // ...
  Sensor sensor;
  sensor.SetTransferFunction(transfer);
  // Calculate the induced current.
  // ...
  // Apply the transfer function.
  sensor.ConvoluteSignals();
  // ...
}
```

In the second one, we use a `Shaper` object.

```
int main(int argc, char* argv[]) {

  // ...
  Shaper shaper(1, 25., 1., "unipolar");
  Sensor sensor;
  sensor.SetTransferFunction(shaper);
  // ...
  sensor.ConvoluteSignals();
  // ...
}
```

7.1.1. Noise

Prior to convoluting the induced current with a transfer function, one can add a random noise component to the signal using

```
void AddWhiteNoise(const double enc, const bool poisson = true, const double q0 = 1.);
```

enc desired equivalent noise charge (ENC) of the convoluted signal,

poisson flag whether to sample the number of noise pulses from a Poisson distribution, or to sample the noise charge in each bin from a Gaussian distribution,

q0 amplitude of the noise delta pulses (in electrons).

The algorithm is based on the fact that white current noise is equivalent to a random sequence of delta current pulses with large frequency ν and with constant amplitude q_0. When processing this signal by an amplifier with transfer function $f(t)$, with a peak normalized to unity, the variance of the output signal becomes

$$\nu q_0^2 \int f(t)^2 \, \mathrm{d}t.$$

We want this to be equivalent to the ENC^2, which defines

$$\nu = \frac{1}{q_0^2} \frac{\mathrm{ENC}^2}{\int f(t)^2 \, \mathrm{d}t}.$$

The total number of current delta pulses in a period of time Δt is Poisson distributed with a mean $\nu \Delta t$ and a standard deviation $\sqrt{\nu \Delta t}$. With the flag `poisson` set to `true`, a Poisson-distributed number of current pulses is added to the signal in the time window.

For large frequencies, the Poisson distribution becomes a Gaussian distribution, so the standard deviation of charge in a time bin Δt is

$$\sigma_q = q_0 \sqrt{\nu \Delta t}.$$

With the flag `poisson` set to `false`, a Gaussian-distributed noise charge is added to each signal bin.

7.2. Delayed signals

For detectors with resistive elements, the induced signal as function of time is not only given by the movement of the charges in the drift medium but also by the time dependent reaction of the resistive materials. Computing the induced current in such a configuration requires an extended version of the Ramo-Shockley theorem,

$$i(t) = -q \int_0^t \mathrm{d}t' \mathbf{H}\left[\mathbf{x}(t'), t - t'\right] \cdot \dot{\mathbf{x}}(t'), \tag{7.3}$$

where

$$\mathbf{H}_w(\mathbf{x}, t) = -\nabla \frac{\partial \phi_w(\mathbf{x}, t)}{\partial t}$$

is the time-dependent weighting vector.

The time-dependent weighting potential $\phi_w(\mathbf{x}, t)$ can be written as the sum of a static prompt component and a dynamic delayed component

$$\phi_w(\mathbf{x}, t) = \phi_w^p(\mathbf{x}) + \phi_w^d(\mathbf{x}, t), \qquad \text{where } \phi_w^d(\mathbf{x}, 0) = 0. \tag{7.4}$$

We can then write Eq. (7.3) as

$$i(t) = \underbrace{-q\mathbf{E}_w(\mathbf{x}(t)) \cdot \dot{\mathbf{x}}(t)}_{\text{direct induction}} \underbrace{- q \int_0^t \mathrm{d}t' \mathbf{H}_w^d\left[\mathbf{x}(t'), t - t'\right] \cdot \dot{\mathbf{x}}(t')}_{\text{reaction from resistive material}}. \tag{7.5}$$

The computation of the delayed component of the induced current, *i. e.* the second term in Eq. 7.5, can be activated or deactivated using the function

```
void Sensor::EnableDelayedSignal(const bool on);
```

To be able to compute the delayed component, the dynamic weighting potential or weighting field of the electrode must be defined. The method for doing so depends on the type of `Component`. At present, `ComponentComsol`, `ComponentGrid`, `ComponentTcad2d`, `ComponentTcad3d` and `ComponentUser` support time-dependent weighting fields/potentials.

A. Units and constants

The basic units are cm for distances, g for (macroscopic) masses, and ns for times. Particle energies, momenta and masses are expressed in eV, eV/c and eV/c^2, respectively. For example, the electron mass is given in eV/c^2, whereas the mass density of a material is given in g/cm^3. The mass of an atom is specified in terms of the atomic mass number A.

There are a few exceptions from this system of units, though.

- The unit for the magnetic field **B** corresponding to the above system of units (10^{-5} Tesla) is impractical. Instead, magnetic fields are expressed in Tesla.
- Pressures are specified in Torr.
- Electric charge is expressed in fC.

A summary of commonly used quantities and their units is given in Table A.1.

The values of the physical constants used in the code are defined in the file `FundamentalConstants.hh`.

Table A.1. Physical units.

physical quantity	unit
length	cm
mass	g
time	ns
temperature	K
electric potential	V
electric charge	fC
energy	eV
pressure	Torr
electric field	V / cm
magnetic field	Tesla
electric current	fC / ns
angle	rad

B. Gases

Table B.1 shows a list of the gases available in the current version of Magboltz. The star rating represents an estimate of the reliability of the cross-section data for the respective gas. A rating of "5*" indicates a detailed, well-validated description of the cross-sections, while "2*" indicates a low quality, that is a coarse modelling of the cross-sections associated with large uncertainties.

chem. symbol	name	rating
^{4}He	helium	5*
^{3}He	helium-3	5*
Ne	neon	5*
Ar	argon	5*
Kr	krypton	4*
Xe	xenon	4*
Cs	caesium	2*
Hg	mercury	2*
H_2	hydrogen	5*
para H_2	para hydrogen	5*
D_2	deuterium	5*
ortho D_2	ortho deuterium	4*
N_2	nitrogen	5*
O_2	oxygen	5*
F_2	fluorine	2*
CO	carbon monoxide	5*
NO	nitric oxide	2*
H_2O	water	5*
CO_2	carbon dioxide	5*
N_2O	nitrous oxide	4*
O_3	ozone	3*
H_2S	hydrogen sulfide	2*
COS	carbonyl sulfide	2*
CS_2	carbon disulfide	2*
CH_4	methane	5*
CD_4	deuterated methane	4*
C_2H_6	ethane	5*
C_3H_8	propane	4*
nC_4H_{10}	n-butane	4*
iC_4H_{10}	isobutane	4*
nC_5H_{12}	n-pentane	4*
neo-C_5H_{12}	neopentane	4*
C_2H_4	ethene	5*

C_2H_2	acetylene	4*
C_3H_6	propene	4*
cC_3H_6	cyclopropane	4*
CH_3OH	methanol	4*
C_2H_5OH	ethanol	4*
C_3H_7OH	isopropanol	3*
nC_3H_7OH	n-propanol	3*
$C_3H_8O_2$	methylal	2*
$C_4H_{10}O_2$	DME	5*
CF_4	tetrafluoromethane	5*
CHF_3	fluoroform	3*
C_2F_6	hexafluoroethane	4*
$C_2H_2F_4$	tetrafluoroethane	4*
C_3F_8	octafluoropropane	4*
SF_6	sulfur hexafluoride	4*
BF_3	boron trifluoride	4*
CF_3Br	bromotrifluoromethane	3*
NH_3	ammonia	4*
$N(CH_3)_3$	TMA	3*
SiH_4	silane	4*
GeH_4	germane	3*
CCl_4	carbon tetrachloride	4*

Table B.1. Gases available in Magboltz 11.19.

C. Solids

The constructors of the `Solid` classes usually come in two versions: one where the solid is kept in its default orientation (*i.e.* the local frame of the solid and the global frame are identical except for a translation), and one where the solid's orientation is set explicitly. This is done by specifying the direction of the solid's local z axis in the global frame.

C.1. Box

`SolidBox` describes a rectangular cuboid (Fig. C.1). The simplest version of the constructor creates a box the edges of which are aligned with the axes of the coordinate system.

```
SolidBox(const double cx, const double cy, const double cz,
         const double lx, const double ly, const double lz);
```

cx,cy,cz Coordinates of the centre of gravity of the box.

lx,ly,lz Half-widths of the box along x, y, and z.

A box with a non-default orientation in space is created using

```
SolidBox(const double cx, const double cy, const double cz,
         const double lx, const double ly, const double lz,
         const double dx, const double dy, const double dz);
```

C.2. Tube

`SolidTube` describes a cylinder. The simplest constructor requires the location of the centre, the (outer) radius and the half-length.

```
SolidTube(const double cx, const double cy, const double cz, const double r,
          const double lz);
```

cx,cy,cz Coordinates of the centre of gravity of the cylinder.

r Outer radius of the cylinder.

lz Half-length of the cylinder.

By default, the central axis of the cylinder is collinear with the z-axis, as illustrated in Fig. C.2. To create a cylinder with a different orientation, the constructor

```
SolidTube(const double cx, const double cy, const double cz, const double r,
          const double lz, const double dx, const double dy, const double dz);
```

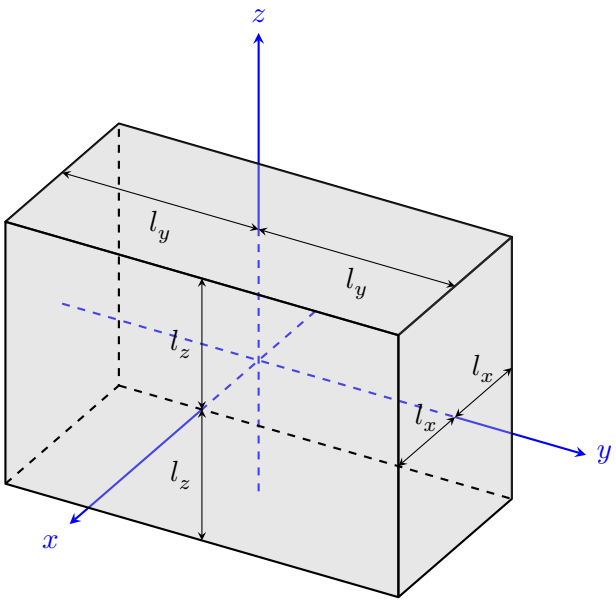

Figure C.1. `SolidBox` centred at $(0, 0, 0)$, defined by its half-widths l_x, l_y, l_z.

can be used.

When determining the surface panels (*e.g.* for use in neBEM), the cylinder is approximated as a polygon with a finite number of panels. The type of polygon to be used can be set using

```
void SetSectors(const unsigned int n);
```

The number of corners of the polygon equals $4(n-1)$. Thus, $n = 2$ will produce a square, $n = 3$ an octagon. By default, the polygon used for approximating the cylinder is inscribed in a circle of the specified radius. If the "average-radius" flag is activated using

```
void SetAverageRadius(const bool average);
```

the radius will be interpreted as the mean radius of the polygon that approximates the cylinder. By default, the list of surface panels will include the "lids" at $\pm z$. This can be switched off using the functions

```
void SetTopLid(const bool closed);
void SetBottomLid(const bool closed);
```

C.3. Sphere

`SolidSphere` has two constructors. The first one takes the location of the centre and the outer radius of the sphere.

```
SolidSphere(const double cx, const double cy, const double cz,
            const double r);
```

cx,cy,cz Coordinates of the centre of the sphere.

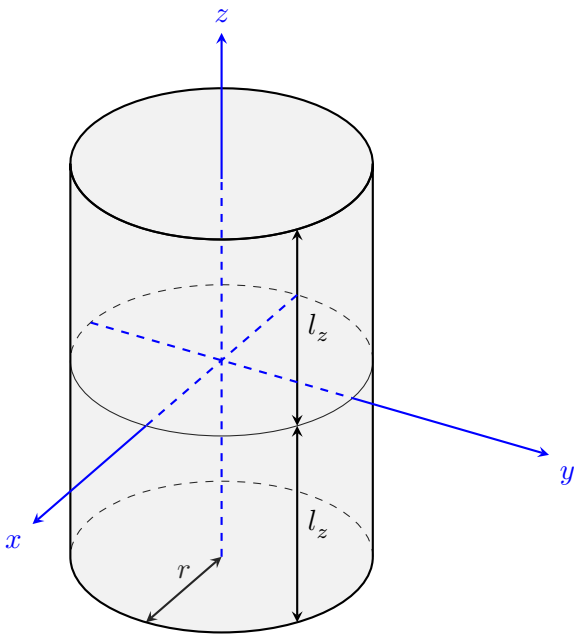

Figure C.2. `SolidTube` centred at $(0, 0, 0)$.

r Radius of the sphere.

The second one takes the coordinates of the centre and the inner and outer radii.

```
SolidSphere(const double cx, const double cy, const double cz,
            const double rmin, const double rmax);
```

When calculating surface panels (*e.g.* for use in neBEM), the sphere is approximated by a set of parallelograms. The parameter n, set using

```
void SetMeridians(const unsigned int n);
```

specifies the number of meridians and also the number of parallels.

C.4. Hole

`SolidHole` describes an, optionally tapered, cylindrical hole in a box. Mandatory parameters are the location of the centre, the radii and the dimensions of the box (see Fig. C.4 for an illustration of the parameters).

```
SolidHole(const double cx, const double cy, const double cz,
          const double rup, const double rlow,
          const double lx, const double ly, const double lz);
```

cx,cy,cz Location of the point that is on the central axis of the hole, half-way between the two planes of the box perpendicular to the central axis of the hole.

rup,rlow Radius of the hole as measured at the "upper" and at the "lower" surface of the box.

lx,ly,lz Half-lenghts of the box.

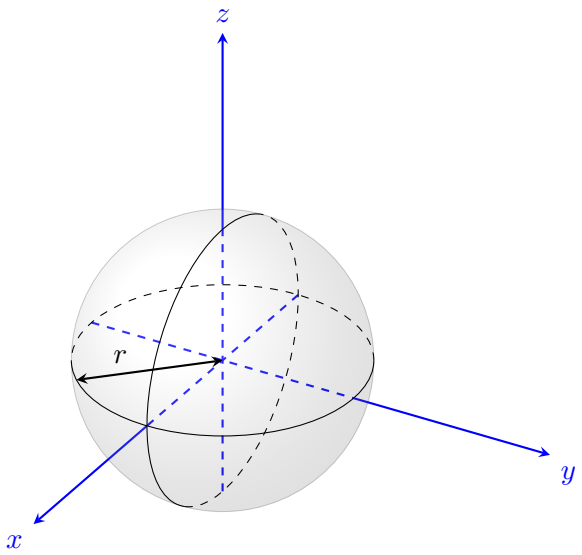

Figure C.3. `SolidSphere` centred at $(0, 0, 0)$.

The central axis of the hole is collinear with the z-axis. To create a hole with a non-default orientation, the constructor

```
SolidHole(const double cx, const double cy, const double cz,
          const double rup, const double rlow,
          const double lx, const double ly, const double lz,
          const double dx, const double dy, const double dz);
```

is available.

Like in `SolidTube`, the type of polygon used for approximating the hole when calculating the surface panels can be specified using

```
void SetSectors(const unsigned int n);
```

C.5. Ridge

`SolidRidge` describes a ridge, similar to a Toblerone bar. The constructor takes the location of the centre and size of the floor, and the position of the ridge proper.

```
SolidRidge(const double cx, const double cy, const double cz,
           const double lx, const double ly, const double hz,
           const double hx);
```

cx,cy,cz Centre of the floor of the ridge in the (x, y) plane at $z = 0$.

lx,ly Half-lenghts of the floor.

hz Height of the ridge measured from the floor.

hx Offset in the x-direction of the ridge. If the offset is set to 0, then the ridge will be symmetric.

An illustration is given in Fig. C.5. By default, the ridge is taken to be parallel with the y-axis.

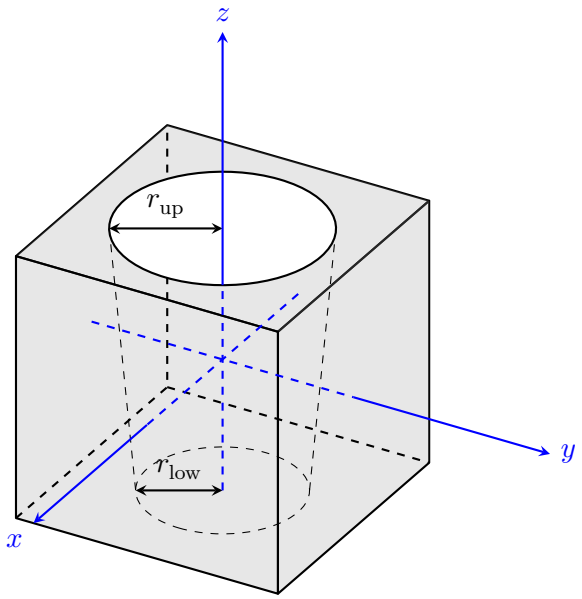

Figure C.4. `SolidHole` centred at $(0, 0, 0)$.

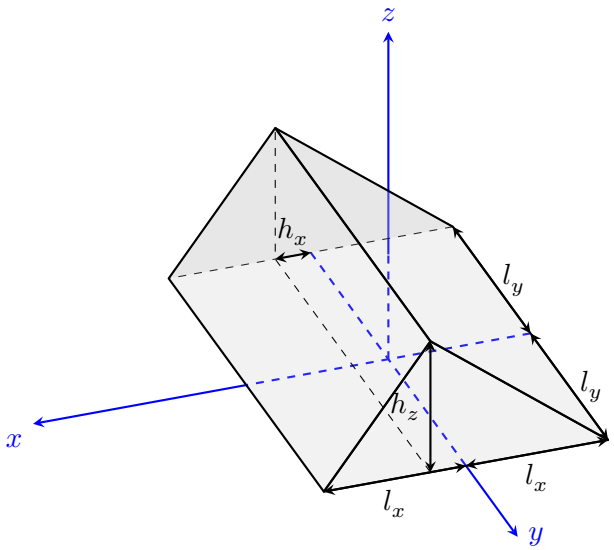

Figure C.5. `SolidRidge` in its default orientation, centred at $(0, 0, 0)$.

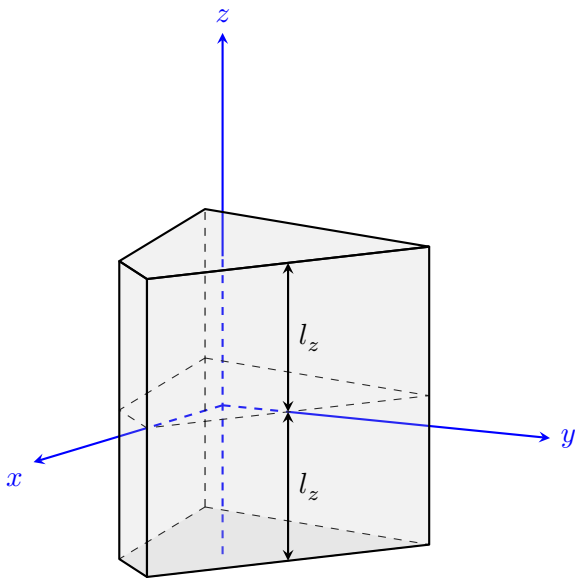

Figure C.6. `SolidExtrusion` in its default orientation.

C.6. Extrusion

`SolidExtrusion` (illustrated in Fig. C.6) describes a volume generated by extruding a polygon along its normal axis (default: z-axis). The constructor takes the half-length along z, and the x, y coordinates of the polygon defining the extrusion.

```
SolidExtrusion(const double lz,
               const std::vector<double>& xp, const std::vector<double>& yp);
```

lz Half-length of the extrusion along z.

xp,yp x, y coordinates of the point defining the extrusion profile.

To create an extrusion with an orientation or offset different from the default one, the constructor

```
SolidExtrusion(const double lz,
               const std::vector<double>& xp, const std::vector<double>& yp,
               const double cx, const double cy, const double cz,
               const double dx, const double dy, const double dz);
```

can be used.

Bibliography

[1] G. D. Alkhazov. "Statistics of electron avalanches and ultimate resolution of proportional counters". In: *Nucl. Instr. Meth.* 89 (1970), pp. 155–165.

[2] *ANSYS*. http://www.ansys.com.

[3] *Atomic and Molecular Data for Radiotherapy and Radiation Research*. IAEA TECDOC 799. IAEA, 1995.

[4] *Average energy required to produce an ion pair*. ICRU Report 31. Washington, DC: International Commission on Radiation Units and Measurements, 1979.

[5] J. J. Barnes, R. J. Lomax, and G. I. Haddad. "Finite-element simulation of GaAs MESFETs with lateral doping profiles and submicron gates". In: *IEEE Transactions on Electron Devices* 23 (1976), pp. 1042–1048. doi: 10.1109/T-ED.1976.18533.

[6] S. F. Biagi. "Magboltz 11". http://magboltz.web.cern.ch/magboltz.

[7] S. F. Biagi. "Monte Carlo simulation of electron drift and diffusion in counting gases under the influence of electric and magnetic fields". In: *Nucl. Instr. Meth. A* 421 (1999), pp. 234–240.

[8] W. Blum, W. Riegler, and L. Rolandi. *Particle Detection with Drift Chambers*. Springer, 2008.

[9] R. Brun, F. Rademakers, et al. *ROOT: An Object-Oriented Data Analysis Framework*. http://root.cern.ch.

[10] C. Canali et al. "Electron and Hole Drift VelocityMeasurements in Silicon and Their Empirical Relation to Electric Field and Temperature". In: *IEEE Trans. Electron Devices* 22 (1975), pp. 1045–1047.

[11] *COMSOL Multiphysics*. https://www.comsol.com.

[12] H. W. Ellis et al. "Transport properties of gaseous ions over a wide energy range". In: *At. Data Nucl. Data Tables* 17 (1976), pp. 177–210.

[13] H. W. Ellis et al. "Transport properties of gaseous ions over a wide energy range. Part II". In: *At. Data Nucl. Data Tables* 22 (1978), pp. 179–217.

[14] H. W. Ellis et al. "Transport properties of gaseous ions over a wide energy range. Part III". In: *At. Data Nucl. Data Tables* 31 (1984), pp. 113–151.

[15] *Elmer*. https://www.csc.fi/web/elmer.

[16] *Gmsh – A three-dimensional finite element mesh generator with built-in pre- and post-processing facilities*. http://gmsh.info.

[17] W. N. Grant. "Electron and Hole Ionization Rates in Epitaxial Silicon at High Fields". In: *Solid State Electronics* 16 (1973), pp. 1189–1203.

[18] I. Krajcar Bronić. "On a relation between the W value and the Fano factor". In: *J. Phys. B* 25 (1992), p. L215. doi: 10.1088/0953-4075/25/8/004.

[19] C. Lombardi et al. "A Physically Based Mobility Model for Numerical Simulation of Nonplanar Devices". In: *IEEE Trans. CAD* 7 (1988), pp. 1164–1171.

[20] N. Majumdar and S. Mukhopadhyay. “Simulation of 3D electrostatic configuration in gaseous detectors”. In: *JINST* 2.9 (2007).

[21] N. Majumdar and S. Mukhopadhyay. “Simulation of three-dimensional electrostatic field configuration in wire chambers: a novel approach”. In: *Nucl. Instr. Meth. A* 566 (2006).

[22] G. Masetti, M. Severi, and S. Solmi. “Modeling of Carrier Mobility Against Carrier Concentration in Arsenic-, Phosphorus-, and Boron-Doped Silicon”. In: *IEEE Trans. Electron Devices* 30 (1983), pp. 764–769.

[23] D. J. Massey, J. P. R. David, and G. J. Rees. “Temperature Dependence of Impact Ionization in Submicrometer Silicon Devices”. In: *IEEE Transactions on Electron Devices* 53 (2006), pp. 2328–2334. doi: 10.1109/TED.2006.881010.

[24] S. Mukhopadhyay and N. Majumdar. “A study of three-dimensional edge and corner problems using the neBEM solver”. In: *Engineering Analysis with Boundary Elements* 33 (2009).

[25] S. Mukhopadhyay and N. Majumdar. “Computation of 3D MEMS electrostatics using a nearly exact BEM solver”. In: *Engineering Analysis with Boundary Elements* 30 (2006).

[26] Y. Okuto and C. R. Crowell. “Threshold Energy Effect on Avalanche Breakdown Voltage in Semiconductor Junctions”. In: *Solid State Electronics* 18 (1975), pp. 161–168.

[27] M. A. Omar and L. Reggiani. “Drift and diffusion of charge carriers in silicon and their empirical relation to the electric field”. In: *Solid State Electronics* 30 (1987), pp. 693–697.

[28] R. van Overstraeten and H. de Man. “Measurement of the Ionization Rates in Diffused Silicon p-n Junctions”. In: *Solid State Electronics* 13 (1970), pp. 583–608.

[29] A. Pansky, A. Breskin, and R. Chechik. “Fano factor and the mean energy per ion pair in counting gases, at low x-ray energies”. In: *J. Appl. Phys.* 82 (1997), p. 871. doi: 10.1063/1.365787.

[30] M. Pomorski. “Electronic Properties of Single Crystal CVD Diamond and Its Suitability for Particle Detection in Hadron Physics Experiments”. PhD thesis. Johann Wolfgang Goethe University, Frankfurt am Main, 2008.

[31] R. Quay et al. “A Temperature Dependent Model for the Saturation Velocity in Semiconductor Materials”. In: *Materials Science in Semiconductor Processing* 3 (2000), pp. 149–155.

[32] S. Ramo. “Currents Induced by Electron Motion”. In: *Proceedings of the I.R.E.* 27 (1939), pp. 584–585.

[33] G. F. Reinking, L. G. Christophorou, and S. R. Hunter. “Studies of total ionization in gases/mixtures of interest to pulsed power applications”. In: *J. Appl. Phys.* 60 (1986), p. 499. doi: 10.1063/1.337792.

[34] W. Riegler and G. Aglieri Rinella. “Point charge potential and weighting field of a pixel or pad in a plane condenser”. In: *Nucl. Instr. Meth. A* 767 (2014), pp. 267–270.

[35] A. Rotondi and P. Montagna. “Fast calculation of Vavilov distribution”. In: *Nucl. Instr. Meth. B* 47 (1990), p. 215. doi: 10.1016/0168-583X(90)90749-K.

[36] O. Sahin. “Excitation energy transfer model for Ne-CO_2 and Ne-N_2 mixtures”. In: *JINST* 16 (2021), P03026.

[37] O. Sahin and T. Z. Kowalski. “A comprehensive model of Penning energy transfers in Ar - CO_2 mixtures”. In: *JINST* 12 (2017), p. C01035.

[38] O. Sahin and T. Z. Kowalski. “Measurements and calculations of gas gain in Xe-5% TMA mixture-pressure scaling”. In: *JINST* 13 (2018), P10032.

[39] O. Sahin et al. “Penning transfer in argon-based gas mixtures”. In: *JINST* 5 (2010), P05002.

[40] F. Sauli. “The gas electron multiplier (GEM): Operating principles and applications”. In: *Nucl. Instr. Meth. A* 805 (2016), pp. 2–24.

[41] S. Selberherr et al. “The Evolution of the MINIMOS Mobility Model”. In: *Solid State Electronics* 33 (1990), pp. 1425–1436.

[42] W. Shockley. “Currents to Conductors Induced by a Moving Point Charge”. In: *J. Appl. Phys.* 9 (1938), pp. 635–636.

[43] H. R. Skullerud. “The stochastic computer simulation of ion motion in a gas subjected to a constant electric field”. In: *Brit. J. Appl. Phys.* 1 (series 2) (1968), pp. 1567–1577.

[44] I. B. Smirnov. “Modeling of ionization produced by fast charged particles in gases”. In: *Nucl. Instr. Meth. A* 554 (2005), pp. 474–493.

[45] D. Srdoc, B. Obelic, and I. Krajcar Bronić. “Statistical fluctuations in the ionisation yield of low-energy photons absorbed in polyatomic gases”. In: *J. Phys. B* 20 (1987), p. 4473. doi: 10.1088/0022-3700/20/17/025.

[46] J. Stoer and R. Bulirsch. *Introduction to Numerical Analysis*. Springer, 1993.

[47] *Synopsys Sentaurus Device*. http://www.synopsys.com/products/tcad/tcad.html.

[48] R. Veenhof. *Garfield - simulation of gaseous detectors*. http:://cern.ch/garfield.

[49] L. Viehland and E. A. Mason. “Transport properties of gaseous ions over a wide energy range, IV”. In: *At. Data Nucl. Data Tables* 60 (1995), pp. 37–95.

[50] J. F. Ziegler, J. Biersack, and U. Littmark. *The Stopping and Range of Ions in Matter*. Pergamon Press, 1985.