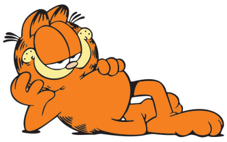
Garfield++ User Guide



Version 2014.1

H. Schindler

March 2014

Contents

1. Introduction

Garfield++ is an object-oriented toolkit for the detailed simulation of particle detectors which use a gas mixture or a semiconductor material as sensitive medium.

For calculating electric fields, three 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 [18].

In the future, an interface to the neBEM field solver [11, 12] (which already exists for Garfield [20]), should be made available.

For calculating the transport properties of electrons in gas mixtures, an interface to the "Magboltz" program [2, 3] is available.

The ionization pattern produced along the track of relativistic charged particles can be simulated using the program "Heed" [17].

The present document aims to provide an overview of the Garfield++ classes and their key functionalities, 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 webpage http://cern.ch/garfieldpp. If you have questions, doubts, comments etc. about the code or this manual, please don't hesitate to contact the authors. Any kind feedback is highly welcome.

1.1. Class Structure

An overview of the different types of classes is given in Fig. 1.1. Two main categories can be distinguished: classes for describing the detector (material properties, geometry, fields), and transport classes which deal with tracing particles through the device. The two class types are linked by the class `Sensor`.

The individual classes are explained in detail in the following chapters.

Readers familiar with the structure of (Fortran) Garfield [20] 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 graphics classes of the ROOT framework [4].

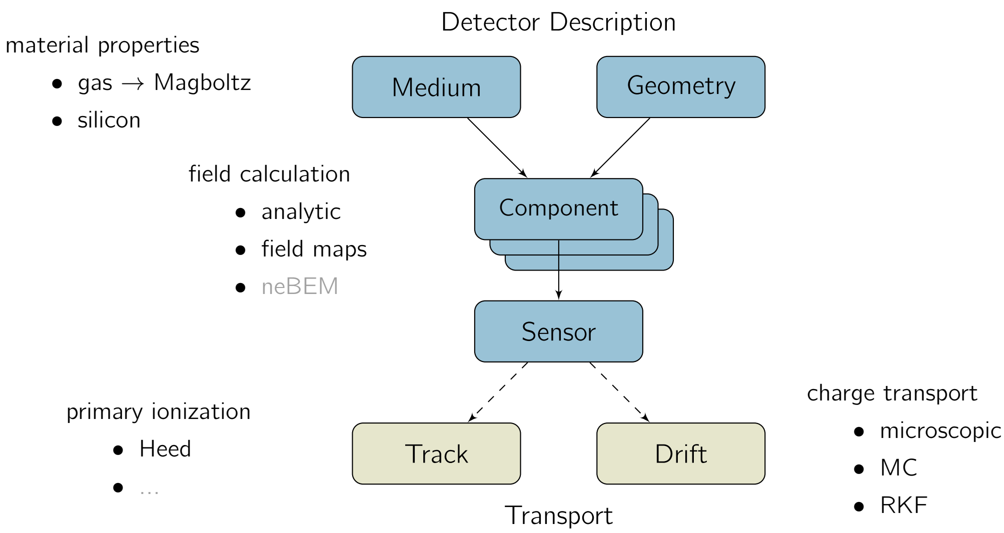


Figure 1.1. Overview of the main classes in Garfield++ and their interplay.

2. Getting Started

2.1. Installation

The source code is hosted on a Subversion[1] (svn) repository managed by the CERN Central SVN service. Web interfaces for browsing the code are available at:

- `http://svnweb.cern.ch/world/wsvn`,
- `http://svnweb.cern.ch/trac/garfield`.

The following instructions describe how to download and build Garfield++ from source.

- Make sure that ROOT is installed. For installation instructions see `http://root.cern.ch/drupal/content/installing-root-source` or `http://root.cern.ch/drupal/content/downloading-root`.
- Define an environment variable `GARFIELD_HOME` pointing to the directory where the Garfield++ classes are to be located. In the following, we assume that we want to install Garfield in a directory `/home/mydir/garfield`. If you are using bash, type

  ```
  export GARFIELD_HOME=/home/mydir/garfield
  ```

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

  For (t)csh-type shells, type

  ```
  setenv GARFIELD_HOME /home/mydir/garfield
  ```

  Include the above lines also in the `.bashrc` (or `.cshrc`) file in your home directory. If unsure which shell you are using, type `echo $SHELL`.
- Download ("check out") the code from the repository. This can be done via SSH access or via HTTP access. For SSH access, give the command

  ```
  svn co svn+ssh://username@svn.cern.ch/reps/garfield/trunk $GARFIELD_HOME
  ```

  where `username` is your CERN afs login. For HTTPS access, give the command

  ```
  svn co https://username@svn.cern.ch/reps/garfield/trunk $GARFIELD_HOME
  ```

  or, if you do not have a CERN account,

[1] For more information about Subversion, have a look at `http://svn.web.cern.ch/svn/docs.php` and the documents listed there.

```
svn co http://svn.cern.ch/guest/garfield/trunk $GARFIELD_HOME
```

Alternatively, you can download the tarballs from the web interface (see the above address) and extract them in the `$GARFIELD_HOME` directory.

- Change to the `$GARFIELD_HOME` directory (`cd $GARFIELD_HOME`).
- If necessary, adapt the `makefile` according to your configuration. By default, `gfortran` is used as Fortran compiler. In order to use a different compiler (e. g. g77), you can modify the definition of the variable `$FC` in the `makefile` accordingly.
- Compile the classes by giving the command `make`.
- Heed requires an environment variable `HEED_DATABASE` to be defined.

```
export HEED_DATABASE=$GARFIELD_HOME/Heed/heed++/database/
```

Add this line also to your `.bashrc`/`.cshrc` as well.

After the initial "check-out", the command

```
svn update
```

followed by `make` (in case of trouble: try `make clean; make`), can be used for downloading the latest version of the code from the repository.

2.2. Examples

Section 2.2.1 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 dealt with in Sec. 2.2.2.

Sample macros and further examples can be found on the webpage (cern.ch/garfieldpp).

2.2.1. Drift Tube

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 **E** (and, in general, also the magnetic field **B** as well as the angle between **E** and **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 = new MediumMagboltz();
gas->SetComposition("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 the 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;
const bool verbose = true;
gas->GenerateGasTable(ncoll, verbose);
```

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");
```

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 = new ComponentAnalyticField();
```

The `Component` requires a description of the geometry, that is a list of volumes and associated media.

```
// Wire radius [cm]
const double rWire = 25.e-4;
// Outer radius of the tube [cm]
const double rTube = 1.46;
// Half-length of the tube [cm]
const double lTube = 10.;
GeometrySimple* geo = new GeometrySimple();
// Make a tube
// (centered at the origin, inner radius: rWire, outer radius: rTube).
SolidTube* tube = new SolidTube(0., 0., 0., rWire, rTube, lTube);
// Add the solid to the geometry, together with the medium inside.
geo->AddSolid(tube, gas);
// Pass a pointer to the geometry class to the component.
cmp->SetGeometry(geo);
```

Next we setup the electric field.

```
// Voltages
const double vWire = 3270.;
const double vTube =  0.;
// Add the wire in the center.
cmp->AddWire(0., 0., 2 * rWire, vWire, "s");
// Add the tube.
cmp->AddTube(rTube, vTube, 0, "t");
```

We want to calculate the signal induced on the wire. Using

```
cmp->AddReadout("s");
```

we tell the `Component` to prepare the solution for the weighting field of the wire (which we have given the label "s" before).

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

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

In this (not very realistic) example, we want to calculate only the electron signal. We set the time interval within which the signal is recorded by the sensor to 2 ns, with a binning of 0.02 ns.

```
const double tMin = 0.;
const double tMax = 2.;
const double tStep = 0.02;
const int nTimeBins = int((tMax - tMin) / tStep);
sensor->SetTimeWindow(0., tStep, nTimeBins);
```

Avalanche

For simulating the electron avalanche we use the class `AvalancheMC` which uses the previously computed tables of transport parameters to calculate drift lines and multiplication.

```
AvalancheMC* aval = new AvalancheMC();
aval->SetSensor(sensor);
// Switch on signal calculation.
aval->EnableSignalCalculation();
// Do the drift line calculation in time steps of 50 ps.
aval->SetTimeSteps(0.05);
// Starting position [cm] and time [ns] of the initial electron.
// The electron is started at 100 micron above the wire.
const double x0 = 0.;
const double y0 = rWire + 100.e-4;
```

```
const double z0 = 0.;
const double t0 = 0.;
// Simulate an avalanche.
aval->AvalancheElectron(x0, y0, z0, t0);
```

Using the class `ViewSignal`, we plot the current induced on the wire by the avalanche simulated in the previous step.

```
ViewSignal* signalView = new ViewSignal();
signalView->SetSensor(sensor);
signalView->PlotSignal("s");
```

2.2.2. GEM

Field Map

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

Next we create a `Sensor` and add the field map component to it

```
Sensor* sensor = new Sensor();
sensor->AddComponent(fm);
```

Gas

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

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

In this example, we calculate electron avalanches using "microscopic" Monte Carlo simulation, based directly on the electron-atom/molecule cross-sections in the Magboltz database.

```
gas->SetMaxElectronEnergy(200.);
const bool verbose = true;
gas->Initialise(verbose);
```

In order to track a particle through the detector we have to tell `ComponentAnsys123` which field map material corresponds to which `Medium`.

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

Avalanche

Microscopic tracking is handled by the class `AvalancheMicroscopic`.

```
AvalancheMicroscopic* aval = new AvalancheMicroscopic();
aval->SetSensor(aval);
```

We are now ready to track an electron through the GEM.

```
// 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);
```

3. Media

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

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

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

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

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

3.1. Transport Parameters

`Medium` classes provide functions for calculating the macroscopic transport parameters of electrons, holes, and ions 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})

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

Table 3.1. Pressure scaling relations for gases.

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

Analogous functions are available for holes (although of course not meaningful for gases), and also for ions (except for the Townsend and attachment coefficients). A function specific to ions is

```
bool IonDissociation(const double ex, const double ey, const double ez,
                     const double bx, const double by, const double bz,
                     double& diss);
```

It returns the dissociation coefficient (in cm^{-1}).

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

- the electric field **E**,
- the component of the magnetic field **B** transverse to **E**,
- $\mathbf{E} \times \mathbf{B}$.

The longitudinal diffusion is measured along **E**. The transverse diffusion is the average of the diffusion coefficients along the two remaining axes.

3.1.1. Transport 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 **E**, **B**, and the angle θ between **E** and **B**). If only a one-dimensional table is present and the drift velocity at $B \neq 0$ is requested, the Laue-Langevin equation is used.

In the present version of the code, 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
- $\mathbf{B} = 0$, $\theta = 0$

For specifying the field grid, two functions are available:

```
void SetFieldGrid(double emin, double emax, int ne, bool logE,
                  double bmin, double bmax, int nb,
                  double amin, double amax, int 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 rad.

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

```
void SetInterpolationMethodVelocity(const int intrp);
void SetInterpolationMethodDiffusion(const int intrp);
void SetInterpolationMethodTownsend(const int intrp);
void SetInterpolationMethodAttachment(const int intrp);
void SetInterpolationMethodIonMobility(const int intrp);
void SetInterpolationMethodIonDissociation(const 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 3-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.

For plotting the transport parameters, the class `ViewMedium` can be used.

collision type	index
elastic collision	0
ionisation	1
attachment	2
inelastic collision	3
excitation	4
superelastic collision	5

Table 3.2. Classification of electron collision processes.

3.2. Electron Scattering Rates

For calculating electron drift lines using "microscopic tracking" (see Sec. 6.3), the preparation of an electron transport table is not necessary, since this method is based directly on the electron-atom/molecule scattering rates.

The following functions which are meant to be called from within the class `AvalancheMicroscopic` are available in `Medium`:

- ```
  double GetElectronCollisionRate(const double e, const int band = 0);
  ```

  returns the total scattering rate of an electron with energy `e` (in eV) in the `Medium`. The band index is relevant only for semiconducting media.

- ```
  bool GetElectronCollision(const double e, int& type, int& level,
                            double& e1, double& dx, double& dy, double& dz,
                            int& nion, int& ndx, int& band);
  ```

  e electron energy prior to the collision

  type category of the collision process (see Tab. 3.2)

  level index of the collision process

  e1 electron energy after the collision

  dx, dy, dz incoming and outgoing direction

  nion number of "ionisation products" (i. e. electrons and ions) created in the collision

  ndxc number of "deexcitation products" created in the collision

  band band index (irrelevant for gases)

3.3. Gases

There are currently two classes implemented which 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 [3] – allows 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 fraction of the respective molecule/atom

A valid gas mixture comprises at least one and at most six different species. A list of the presently available gases and their identifiers can be found in the appendix. The fractions have to be strictly positive and may add up to any non-zero value; internally they will be normalized to 1.

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

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

Note that the density is 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 = new MediumMagboltz();
// Set the composition
gas->SetComposition("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).

3.3.1. Ion Transport

The `$GARFIELD_HOME/Data` directory includes a few files (e. g. `IonMobility_Ar+_Ar.txt` for Ar^+ ions in argon) which contain ion mobility data in form of a table of reduced electric fields E/N (in Td[1]) vs. mobilities (in $cm^2\ V^{-1}\ s^{-1}$). These mobility files can be imported using

[1] 1 Td = 10^{-17} V cm^2

```
bool MediumGas::LoadIonMobility(const std::string filename);
```

filename path and filename of the mobility file

Extensive compilations of ion mobilities and diffusion coefficients can be found in Refs. [6–8, 21].

3.3.2. Magboltz

Magboltz, written by Steve Biagi, is a program [3] for the calculation of electron transport properties in gas mixtures using semi-classical Monte Carlo simulation. It includes a database of electron-atom/molecule cross-sections for a large number of detection gases.

`MediumMagboltz` allows running Magboltz for a given electric field, magnetic field and field angle:

```
void RunMagboltz(const double e, const double b, const double btheta,
                 const int ncoll, bool verbose,
                 double& vx, double& vy, double& vz,
                 double& dl, double& dt, double& alpha, double& eta,
                 double& vxerr, double& vyerr, double& vzerr,
                 double& dlerr, double& dterr,
                 double& alphaerr, double& etaerr,
                 double& alphatof);
```

e, b, btheta **E** field, **B** field, and angle

ncoll number of collisions (in multiples of 10^7) over which the electron is tracked

verbose flag switching on/off full output from Magboltz

vx, vy, vz drift velocity along **E** (vz), along $\mathbf{B}_t$ (vy), and along $\mathbf{E} \times \mathbf{B}$ (vy)

dl, dt diffusion coefficients

alpha, eta Townsend and attachment coefficient calculated using the SST technique or, at low fields, the ionization/loss rate

vxerr, vyerr, ..., etaerr statistical error of the calculation in %

alphatof alternative estimate of the effective Townsend coefficient $\alpha - \eta$ based on the Time-Of-Flight method

The max. energy of the cross-section table is chosen automatically by Magboltz. 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.

In order to calculate the electron transport parameters for all values of **E**, **B**, and θ included in the current field grid, the function

```
void GenerateGasTable(const int numCollisions);
```

can be used.

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.

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

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

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

e max. electron energy (in eV)

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

```
int GetNumberOfLevels();
bool GetLevel(const 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 `GetElectronCollision`. These counters are accessible through the functions

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

The first function returns total number of electron collisions (i. e. calls to GetElectronCollisions) 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(const double r, const double lambda);
void EnablePenningTransfer(const double r, const double lambda,
                           std::string gasname);
void DisablePenningTransfer();
void DisablePenningTransfer(std::string gasname);
```

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 functions without the gasname parameter switch on/off 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 ($\epsilon_i > \epsilon_{\text{ion}}$).

3.4. Semiconductors

MediumSilicon is the only semiconductor-type Medium class implemented so far.

3.4.1. Transport Parameters

Like for all Medium classes the user has 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, the transport parameters are calculated based on empirical parameterizations (as used, for instance, in device simulation programs). Several mobility models are implemented. For the

	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 [10]	1.417	−2.5	0.4705	−2.5
Minimos [16]	1.43	−2.0	0.46	−2.18
Reggiani [14]	1.32	−2.0	0.46	−2.2

Table 3.3. Lattice mobility parameter values.

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

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

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. [16]). Using the second function the model described in Ref. [13] 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 [5], which is activated by default,

$$\begin{aligned} v_{\text{sat}}^{e} &= 0.0107 \left(\frac{T_0}{T}\right)^{0.87} \text{cm/ns}, \\ v_{\text{sat}}^{h} &= 0.00837 \left(\frac{T_0}{T}\right)^{0.52} \text{cm/ns}, \end{aligned}$$

where $T_0 = 300$ K. The expressions for the other two implemented models can be found in Refs. [14, 15].

- 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

$$\mu^{e}(E) = \frac{2\mu_0^{e}}{1 + \sqrt{1 + \left(\frac{2\mu_0^{e} E}{v_{\text{sat}}^{e}}\right)^2}}, \qquad \mu^{h}(E) = \frac{\mu_0^{h}}{1 + \frac{\mu_0}{v_{\text{sat}}^{h}}}, \qquad \text{(Minimos)}$$

$$\mu^{e,h}(E) = \frac{\mu_0^{e,h}}{\left(1 + \left(\frac{\mu_0^{e,h} E}{v_{\text{sat}}{}^{e,h}}\right)^{\beta^{e,h}}\right)^{\frac{1}{\beta^{e,h}}}}, \qquad \text{(Canali [5])}$$

$$\mu^{e}(E) = \frac{\mu_0^{e}}{\left(1 + \left(\frac{\mu_0^{e} E}{v_{\text{sat}}^{e}}\right)^{3/2}\right)^{2/3}}, \qquad \mu^{h}(E) = \frac{\mu_0^{h}}{\left(1 + \left(\frac{\mu_0^{h} E}{v_{\text{sat}}^{h}}\right)^{2}\right)^{1/2}}, \qquad \text{(Reggiani [14])}$$

By default, the Canali model is used.

For the impact ionization coefficient, the user has currently the choice between the model of Grant [9] and the model of van Overstraeten and de Man [19].

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

The latter model is used by default.

On an experimental basis, electron collision rates for use with microscopic tracking are also included.

4. Components

The calculation of electric fields is done by classes derived from the abstract base class `ComponentBase`. The key functions 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);
bool GetMedium(const double x, const double y, const double z,
               Medium*& m);
```

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)

The function `GetMedium` returns `true` if a medium was found at the given location.

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` (if available). For the latter 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` classes.

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

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

Table 4.1. Status flags for electric fields.

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`, and
- `SolidTube`.

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 = new MediumMagboltz();
// Create the geometry.
GeometrySimple* geo = new GeometrySimple();
// Dimensions of the tube
double rMin = 0., rMax = 0.5, halfLength = 10.;
SolidTube* tube = new SolidTube(0., 0., 0., rMin, 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 fo 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(GeometryBase* geo);
```

4.1.1. Visualizing the Geometry

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

```
// Create and setup the geometry.
GeometrySimple* geo = new GeometrySimple();
...
// Create a viewer.
ViewGeometry* view = new ViewGeometry();
// Set the pointer to the geometry.
view->SetGeometry(geo);
view->Plot();
```

ROOT geometries can be viewed by calling the `Draw()` function of `TGeoManager`.

The layout of an arrangement of wires, planes and tubes defined in `ComponentAnalyticField` can be inspected using the class `ViewCell`.

```
// Create and setup the component.
ComponentAnalyticField* cmp = new ComponentAnalyticField();
...
// Create a viewer.
ViewCell* view = new ViewCell();
// Set the pointer to the component.
view->SetComponent(cmp);
// Make a two-dimensional plot of the cell layout.
view->Plot2d();
```

Similarly, 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 [1] is dealt with by the classes `ComponentAnsys121` and `ComponentAnsys123`. The class names refer to the type of mesh element used by Ansys:

- `ComponentAnys121` reads two-dimensional field maps with 8-node curved quadrilaterals (known as "plane121" in Anys).
- `ComponentAnsys123` reads three-dimensional field maps with quadric curved tetrahedra (known as "solid123" in Ansys).

The field map is imported with the function

```
bool Initialise(std::string elist, std::string nlist,
                std::string mplist, std::string prnsol,
                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 successfully read.

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

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

imat index in the list of (field map) materials

medium pointer to the Medium class 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 int imat);
double GetConductivity(const int imat);
```

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 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();    void DisablePeriodicityX();
void EnableMirrorPeriodicityX(); void DisableMirrorPeriodicityX();
void EnableAxialPeriodicityX(); void DisableAxialPeriodicityX();
void EnableRotationSymmetryX(); void DisableRotationSymmetryX();
```

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

```
GetElement(const int 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 = new ComponentAnsys123();
...
TH1F* hAspectRatio = new TH1F("hAspectRatio"; "Aspect␣Ratio", 100, 0., 50.);
TH1F* hSize = new TH1F("hSize", "Element␣Size", 100, 0., 30.);
const int nel = fm->GetNumberOfElements();
// Loop over the elements.
double volume, dmin, dmax;
for (int 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();
```

4.2.2. Synopsys TCAD

Electric fields calculated using the device simulation program Synopsys Sentaurus [18] can be imported with the classes `ComponentTcad2d` and `ComponentTcad3d`.

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. 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` classes.

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

ireg index in the list of device regions

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

The name of a region can be retrieved with

```
void GetRegion(const int 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();
```

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 and the mesh tool Gmsh. 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/garfieldpp.html.

4.2.5. Visualizing the Mesh

For visualizing the mesh imported from a FEM 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();
```

then allows draws a two-dimensional projection of the drift lines stored in the `ViewDrift` class together with the mesh. 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 `fm` is a pointer to a field map component and `driftView` is a pointer to a `ViewDrift` class)

```
TCanvas* c1 = new TCanvas();
ViewFEMesh* meshView = new ViewFEMesh();
meshView->SetCanvas(c1);
// Set the component.
meshView->SetComponent(fm);
// 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 – based essentially on the capacitance matrix method – are implemented.

4.3.1. Describing the Cell

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);
```

In all of these functions, the potential `v` (in V) and a label (used for signal calculation) have to be supplied as parameters.

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 g/cm^3) of the wire material. These parameters have no influence on the electric field. The number of wires which 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.

4.3.2. Periodicities

The class supports simple periodicity in x and y direction. The periodic lengths are set by means of

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

4.3.3. Cell Types

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

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

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

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

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();
```

4.3.4. Weighting Fields

The weighting field calculation for a readout group – i. e. all elements (wires, planes, etc.) with the same label – is activated by the function

```
void AddReadout(const std::string 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 char std::string, 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.

4.4. 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 can be set using

```
void SetWeightingField(const double wx, const double wy, const double wz,
                       const std::string label);
```

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

```
void SetElectricField(void (*f)(const double, const double, const double,
                                double& double&, double&));
void SetPotential(void (*f)(const double, const double, const double,
                            double&));
```

f pointer to the user function

As an example, let us consider the 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.

```
void 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.;
  // Detector thickness
  const double d = 0.1;

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

```
}

ComponentUser* component = new ComponentUser();
component->SetElectricField(efield);
```

A user-defined weighting field can be set using

```
void SetWeightingField(void (*f)(const double, const double, const double,
                              double&, double&, double&, const std::string));
```

4.5. Visualizing the Field

The class `ViewField` provides a number of functions for plotting the potential and field of a component.

```
void PlotContour(const std::string option = "v");
void PlotSurface(const std::string option = "v");
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");
```

x0, . . . , z1 coordinates of the start and end points of the line along which the potential is to be evaluated

option quantity to be plotted: potential (`"v"`/`"p"`/`"phi"`), magnitude of the electric field (`"e"`/`"field"`), or individual components of the field (`"ex"`, `"ey"`, `"ez"`)

The first two functions create a contour and surface plot, respectively, in the selected viewing plane. The latter function plots the potential/field along the line $(x_0, y_0, z_0) \rightarrow (x_1, y_1, z_1)$.

The component or sensor of which the field is to be plotted is set by means of

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

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

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

By default, the area is set to the bounding box of the component/sensor.

The density of the plotting grid can be set using

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

The range of the function to be plotted is set using

```
void SetVoltageRange(const double minval, const double maxval);
void SetElectricFieldRange(const double minval, const double maxval);
void SetWeightingFieldRange(const double minval, const double maxval);
```

For the voltage range, the (estimated) minimum and maximum values of the potential in the component/sensor are used as defaults. The range of the electric and weighting fields should always be set by the user.

4.6. 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 classes. 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(ComponentBase* comp);
void AddElectrode(ComponentBase* 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 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(double xmin, double ymin, double zmin,
             double xmax, double ymax, 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 available).

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

5. Tracks

The purpose of classes of the type `Track` is to simulate ionization patterns produced by fast 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

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	$+2$

Table 5.1. Available charged particles.

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 norm, an isotropic random vector will be generated.

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

```
bool GetCluster(double& xcls, double& ycls, double& zcls, double& tlcs,
                int& n, double& e, double& extra);
```

xcls, ycls, zcls, tcls position (and time) of the ionizing collision

n number of electrons produced in the collision

e transferred energy (in eV)

The function returns `false` if the list of clusters is exhausted or if there is no valid track.

The concept of a "cluster" deserves some explanation. In the present context it refers to the energy loss in a single ionizing collision of the primary charged particle and the secondary electrons produced in this process.

5.1. Heed

The program Heed [17] 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`.

After calling `GetCluster`, one can retrieve details about the electrons in the present cluster using

```
bool GetElectron(const int i, double& x, double& y, double& z,
                 double& t, double& e,
                 double& dx, double& dy, double& dz);
```

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 [17].

The asymptotic W value (eV) and the Fano factor of a `Medium` can be specified by the user by means of the functions

```
void Medium::SetW(const double w);
void Medium::SetFanoFactor(const double f);
```

If these parameters are not set, 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 the simulation of x-ray photoabsorption.

```
void 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,
                     int& nel);
```

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

e0 photon energy in eV

dx0, dy0, dz0 direction of the photon

nel number of photoelectrons and Auger-electrons produced in the photon conversion

6. Charge Transport

On a phenomenological level, the drift of charge carriers under the influence of an electric field **E** and a magnetic field **B** is described by the first order equation of motion

$$\dot{\mathbf{r}} = \mathbf{v}_{\mathrm{d}}\left(\mathbf{E}\left(\mathbf{r}\right), \mathbf{B}\left(\mathbf{r}\right)\right), \tag{6.1}$$

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, 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 is implemented in the class `DriftLineRKF`.

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 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 the specified multiple of the "collision time"

$$\tau = \frac{mv_d}{qE}.$$

The third method is activated by default.

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 trajectory can be retrieved using

```
int GetNumberOfDriftLinePoints() const;
void GetDriftLinePoint(const int 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 x0, const double y0, const double z0,
                       const double t0, const bool hole = false);
bool AvalancheHole(const double x0, const double y0, const double z0,
                   const double t0, const bool electron = false);
bool AvalancheElectronHole(const double x0, const double y0, const double z0,
                           const double t0);
```

The flags `hole` and `electron` specify whether multiplication of holes and electrons, respectively, should be taken into account in the simulation. In case of gas-based detectors, only `AvalancheElectron` with `hole = false` is meaningful.

The starting and endpoints of electrons in the avalanche can be retrieved using

```
int GetNumberOfElectronEndpoints();
void GetElectronEndpoint(const int i,
                         double& x0, double& y0, double& z0, double& t0,
                         double& x1, double& y1, double& z1, double& t1,
                         int& status) const;
```

i index of the electron

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

x1, y1, z1, t1 final position and time of the electron

status status code indicating why the tracking of the electron was stopped.

Analogous functions are available for holes and ions.

The functions

```
void EnableMagneticField();
```

instructs the class to consider not only the electric but also the magnetic field in the evaluation of the transport parameters. By default, magnetic fields are not taken into account.

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

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. The time window can be removed using

```
void UnsetTimeWindow();
```

6.3. Microscopic Tracking

Microscopic tracking is (at present) only possible for electrons. It is implemented in the class `AvalancheMicroscopic`. A calculation is started by means of

```
void AvalancheElectron(const double x0, const double y0, const double z0,
                       const double t0, const double e0,
                       const double dx0 = 0., const double dy0 = 0., const double dz0 = 0.);
```

x0, y0, z0, t0 initial position and time

e0 initial energy (eV)

dx0, dy0, dz0 initial direction

If the norm 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);
```

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
-16	energy below transport cut
-17	outside the time window

Table 6.1. Status codes for the termination of drift lines.

Information about the "history" of each avalanche electron can be retrieved by

```
int GetNumberOfElectronEndpoints();
void GetElectronEndpoint(const int i,
              double& x0, double& y0, double& z0, double& t0, double& e0,
              double& x1, double& y1, double& z1, double& t1, double& e1,
                int& status);
```

i index of the electron

x0, y0, z0, t0, e0 initial position, time and energy of the electron

x1, y1, z1, t1, e1 final position, time and energy of the electron

status status code indicating why the tracking of the electron was stopped.

A list of status codes is given in Table 6.1.

The function

```
bool DriftElectron(const double x0, const double y0, const double z0,
                   const double t0, const double e0,
                   const double dx0, const double dy0, const double dz0);
```

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 = new AvalancheMicroscopic();
// Make a histogram (100 bins between 0 and 100 eV).
TH1F* hEnergy = new TH1F("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.

The effect of magnetic fields can be included in the stepping algorithm using the function

```
void EnableMagneticField();
```

By default, magnetic fields are not taken into account in the calculation.

Using

```
void EnableAvalancheSizeLimit(const int size);
```

the size of an electron avalanche can be limited. 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, so-called "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 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" 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 following example we want all excitations which occur to undergo a special treatment.

```
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 = new AvalancheMicroscopic();
  ...
  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);
void AvalancheMC::EnablePlotting(ViewDrift* view);
void Track::EnablePlotting(ViewDrift* view);
```

`ViewDrift` stores the trajectories which are calculated by the transport class. The drawing of the trajectories is triggered by the function

```
void ViewDrift::Plot();
```

In case of AvalancheMicroscopic, it is usually not advisable to plot every single collision. The number of collisions to be skipped for plotting can be set using

```
void AvalancheMicroscopic::SetCollisionSteps(const int n);
```

n number of collisions to be skipped

Note that this setting does not affect the transport of the electron as such, the electron is always tracked rigorously through single collisions.

7. Signals

Signals are calculated using the Shockley-Ramo theorem. 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

$$i(t) = -q\mathbf{v} \cdot \mathbf{E}_w(\mathbf{r}),$$

where $\mathbf{E}_w$ is the so-called weighting field for the electrode to be read out.

The basic steps for calculating the current induced by the drift of electrons and ions/holes are:

1. Prepare the weighting field for the electrode to be read out. This step depends on the field calculation technique (i. e. the type of `Component`) which is used (see Chapter 4).
2. Tell the `Sensor` that you want to use this weighting field for the signal calculation.

   ```
   void Sensor::AddElectrode(ComponentBase* cmp, std::string label);
   ```

   where `cmp` is a pointer to the `Component` which calculates the weighting field, and `label` (in our example `"readout"`) is the name you have assigned to the weighting field in the previous step.
3. Setup 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. Switch on signal calculation in the transport classes using

   ```
   void AvalancheMicroscopic::EnableSignalCalculation();
   void AvalancheMC::EnableSignalCalculation();
   ```

   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.

For plotting the signal, the class `ViewSignal` can be used. 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 = new ComponentAnsys123();
...
fm->SetWeightingField("WPOT.lis", label);

Sensor* sensor = new Sensor();
sensor->AddComponent(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 = new AvalancheMicroscopic();
aval->SetSensor(sensor);
aval->EnableSignalCalculation();
// Calculate some drift lines.
...
// Plot the induced current.
ViewSignal* signalView = new ViewSignal(tStart, tStep, nSteps);
signalView->SetSensor(sensor);
signalView->Plot(label);
```

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". The transfer function to be applied can be set using

```
void Sensor::SetTransferFunction(double (*f)(double t));
```

where `double f(double t)` is a function provided by the user, or using

```
void Sensor::SetTransferFunction(std::vector<double> times,
                                 std::vector<double> values);
```

in which case the transfer function will be calculated by interpolation of the values provided in the table.

In order to convolute the presently stored signal with the transfer function (specified using the above function), the function

```
bool Sensor::ConvoluteSignal();
```

can be called.

As an example, consider the following transfer function

$$f(t) = e\frac{t}{\tau}e^{1-t/\tau}, \qquad \tau = 25 \text{ ns}$$

```
double transfer(double t) {

  const double tau = 25.;
  return (t / tau) * exp(1 - t / tau);

}

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

  // Setup component, media, etc.
  ...
  Sensor* sensor = new Sensor();
  sensor->SetTransferFunction(transfer);
  // Calculate the induced current.
  ...
  // Apply the transfer function.
  sensor->ConvoluteSignal();
  ...
}
```

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`.

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

Table A.1. Physical units.

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	cesium	2*
Hg	mercury	2*
H_2	hydrogen	5*
D_2	deuterium	5*
N_2	nitrogen	5*
O_2	oxygen	4*
F_2	fluorine	2*
CO	carbon monoxide	5*
NO	nitric oxide	4*
H_2O	water	4*
CO_2	carbon dioxide	5*
N_2O	nitrous oxide	2*
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	4*
C_2H_2	acetylene	4*
C_3H_6	propene	4*

cC_3H_6	cyclopropane	4*
CH_3OH	methanol	3*
C_2H_5OH	ethanol	3*
C_3H_7OH	isopropanol	3*
$C_3H_8O_2$	methylal	2*
$C_4H_{10}O_2$	DME	4*
CF_4	tetrafluoromethane	5*
CHF_3	fluoroform	3*
C_2F_6	hexafluoroethane	4*
$C_2H_2F_4$	tetrafluoroethane	2*
C_3F_8	octafluoropropane	3*
SF_6	sulfur hexafluoride	3*
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*

Table B.1. Gases available in Magboltz 8

Bibliography

[1] *Ansys*, `http://www.ansys.com`.

[2] S. F. Biagi, *Magboltz 8*, `http://magboltz.web.cern.ch/magboltz`.

[3] ———, *Monte Carlo simulation of electron drift and diffusion in counting gases under the influence of electric and magnetic fields*, Nucl. Instr. Meth. A **421** (1999), 234–240.

[4] R. Brun, F. Rademakers, et al., *ROOT: An Object-Oriented Data Analysis Framework*, `http://root.cern.ch`.

[5] C. Canali, G. Majni, R. Minder, and G. Ottaviani, *Electron and Hole Drift VelocityMeasurements in Silicon and Their Empirical Relation to Electric Field and Temperature*, IEEE Trans. Electron Devices **22** (1975), 1045–1047.

[6] H. W. Ellis et al., *Transport properties of gaseous ions over a wide energy range*, At. Data Nucl. Data Tables **17** (1976), 177–210.

[7] ———, *Transport properties of gaseous ions over a wide energy range. Part II*, At. Data Nucl. Data Tables **22** (1978), 179–217.

[8] ———, *Transport properties of gaseous ions over a wide energy range. Part III*, At. Data Nucl. Data Tables **31** (1984), 113–151.

[9] W. N. Grant, *Electron and Hole Ionization Rates in Epitaxial Silicon at High Fields*, Solid State Electronics **16** (1973), 1189–1203.

[10] C. Lombardi, S. Manzini, A. Saporito, and M. Vanzi, *A Physically Based Mobility Model for Numerical Simulation of Nonplanar Devices*, IEEE Trans. CAD **7** (1988), 1164–1171.

[11] N. Majumdar and S. Mukhopadhyay, *Simulation of three-dimensional electrostatic field configuration in wire chambers: a novel approach*, Nucl. Instr. Meth. A **566** (2006).

[12] ———, *Simulation of 3D electrostatic configuration in gaseous detectors*, JINST **2** (2007), no. 9.

[13] G. Masetti, M. Severi, and S. Solmi, *Modeling of Carrier Mobility Against Carrier Concentration in Arsenic-, Phosphorus-, and Boron-Doped Silicon*, IEEE Trans. Electron Devices **30** (1983), 764–769.

[14] M. A. Omar and L. Reggiani, *Drift and diffusion of charge carriers in silicon and their empirical relation to the electric field*, Solid State Electronics **30** (1987), 693–697.

[15] R. Quay, C. Moglestue, V. Palankovski, and S. Selberherr, *A temperature dependent model for the saturation velocity in semiconductor materials*, Materials Science in Semiconductor Processing **3** (2000), 149–155.

[16] S. Selberherr, W. Hänsch, M. Seavey, and J. Slotboom, *The Evolution of the MINIMOS Mobility Model*, Solid State Electronics **33** (1990), 1425–1436.

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

[18] *Synopsys Sentaurus Device*, `http://www.synopsys.com/products/tcad/tcad.html`.

[19] R. van Overstraeten and H. de Man, *Measurement of the Ionization Rates in Diffused Silicon p-n Junctions*, Solid State Electronics **13** (1970), 583–608.

[20] R. Veenhof, *Garfield - simulation of gaseous detectors*, `http:://cern.ch/garfield`.

[21] L. Viehland and E. A. Mason, *Transport properties of gaseous ions over a wide energy range, IV*, At. Data Nucl. Data Tables **60** (1995), 37–95.