Allpix Squared User Manual

Paul Schütze (paul.schuetze@desy.de)
Simon Spannagel (simon.spannagel@cern.ch)
Koen Wolters (koen.wolters@cern.ch)
Stephan Lachnit (stephan.lachnit@cern.ch)

2023-05-04

Version v3.0.0

Contents

1 Introduction

Allpix Squared is a generic simulation framework for semiconductor tracker and vertex detectors written in modern C++, following the C++17 standard. The goal of the framework is to provide an easy-to-use package for simulating the performance of semiconductor detectors, starting with the passage of ionizing radiation through the sensor and finishing with the digitization of hits in the readout chip.

The framework builds upon other packages to perform tasks in the simulation chain, most notably Geant4 [1] for the deposition of charge carriers in the sensor and ROOT [2] for producing histograms and storing the produced data. The core of the framework focuses on the simulation of charge transport in semiconductor detectors and the digitization to hits in the frontend electronics.

Allpix Squared is designed as a modular framework, allowing for an easy extension to more complex and specialized detector simulations. The modular setup also allows to separate the core of the framework from the implementation of the algorithms in the modules, leading to a framework which is both easier to understand and to maintain. Besides modularity, the framework was designed with the following main design goals in mind:

1. Reflect the physics:
    - A run consists of several sequential events. A single event here refers to an independent passage of one or multiple particles through the setup.
    - Detectors are treated as separate objects for particles to pass through.
    - All relevant information must be contained at the end of processing every single event (sequential events).
2. Ease of use (user-friendly):
    - Simple, intuitive configuration and execution ("does what you expect").
    - Clear and extensive logging and error reporting capabilities.
    - Implementing a new module should be feasible without knowing all details of the framework.
3. Flexibility:
    - Event loop runs sequence of modules, allowing for both simple and complex user configurations.
    - Possibility to run multiple different modules on different detectors.
    - Limit flexibility for the sake of simplicity and ease of use.

Allpix Squared has been designed following some ideas previously implemented in the AllPix [3, 4] project. Originally written as a Geant4 user application, AllPix has been successfully used for simulating a variety of different detector setups.

1.1 Scope of this Manual

This document is meant to be the primary User's Guide for Allpix Squared. It contains both an extensive description of the user interface and configuration possibilities, and a detailed introduction to the code base for potential developers. This manual is designed to:

- Guide new users through the installation;
- Introduce new users to the toolkit for the purpose of running their own simulations;
- Explain the structure of the core framework and the components it provides to the simulation modules;
- Provide detailed information about all modules and how to use and configure them;
- Describe the required steps for adding new detector models and implementing new simulation modules.

Within the scope of this document, only an overview of the framework can be provided and more detailed information on the code itself can be found in the Doxygen reference manual [5] available online. No programming experience is required from novice users, but knowledge of (modern) C++ will be useful in the later chapters and might contribute to the overall understanding of the mechanisms.

1.2 Support and Reporting Issues

As for most of the software used within the high-energy particle physics community, only limited support on best-effort basis for this software can be offered. The authors are, however, happy to receive feedback on potential improvements or problems arising. Reports on issues, questions concerning the software as well as the documentation and suggestions for improvements are much appreciated. These should preferably be brought up on the issues tracker of the project which can be found in the repository [6]. General support questions are best asked in the forum [7].

The FAQ in Chapter 13 is good place to start looking if a question arises. In particular Section 13.6 should be consulted before opening a bug report.

1.3 Contributing Code

Since Allpix Squared is a community project that benefits from active participation in the development and code contributions from users. Users and prospective developers are encouraged to discuss their needs either via the issue tracker of the repository [6], the forum [7] or the developer's mailing list to receive ideas and guidance on how to implement a specific feature. Getting in touch with other developers early in the development cycle avoids spending time on features which already exist or are currently under development by other users.

An introduction to the development of Allpix Squared and its different tools is provided in Chapter 10, including a "How to contribute" Section describing the steps necessary to get involved in the development. Chapter 11 details the tools provided with the repository which ease and facilitate contributions and ensure code quality.

1.4 Quick Start

This section serves as a swift introduction to Allpix Squared for users who prefer to start quickly and learn the details while simulating. The typical user should skip the next paragraphs and continue reading the following sections instead.

Allpix Squared provides a modular, flexible and user-friendly structure for the simulation of independent detectors in arbitrary configurations. The framework currently relies on the ROOT [2] and Boost.Random [8] libraries, which need to be installed and loaded before using Allpix Squared. For many use cases, installations of Geant4 [1] and Eigen3 [9] are required in addition.

The minimal, default installation can be obtained by executing the commands listed below.

```
git clone https://gitlab.cern.ch/allpix-squared/allpix-squared
cd allpix-squared
mkdir build && cd build/
cmake ..
make install
cd ..
```

The binary can then be executed with the provided example configuration file as follows:

```
bin/allpix -c examples/example.conf
```

Hereafter, the example configuration can be copied and adjusted to the needs of the user. This example contains a simple setup of two test detectors. It simulates the whole chain, starting from the passage of the beam, the deposition of charges in the detectors, the carrier propagation and the conversion of the collected charges to digitized pixel hits. All generated data is finally stored on disk in ROOT TTrees or other commonly used data formats for later analysis.

After this quick start it is highly recommended to proceed to the other chapters of this user manual. For quickly resolving common issues, the Frequently Asked Questions may be particularly useful.

2 Installation

This chapter aims to provide details and instructions on how to build and install Allpix Squared. An overview of possible build configurations is given. After installing and loading the required dependencies, there are various options to customize the installation of Allpix Squared. This chapter contains details on the standard installation process and information about custom build configurations.

Alternatively, Allpix Squared can installed without building via a Docker image (see Section 2.7) or via CVMFS (see Section 2.8).

2.1 Supported Operating Systems and Compilers

2.1.1 Operating Systems

Allpix Squared is designed to run without issues on either a recent Linux distribution or Mac OS X. Furthermore, the continuous integration of the project ensures correct building and functioning of the software framework on CentOS 7 (with GCC and LLVM), CentOS 8 (with GCC only), Ubuntu 20.04 LTS (Docker, GCC) and Mac OS Catalina 10.15 (AppleClang 12.0).

2.1.2 Compilers

Allpix Squared relies on functionality from the C++17 standard and therefore requires a C++17-compliant compiler. This comprises for example GCC 9+, LLVM/Clang 4.0+ and AppleClang 10.0+. A detailed list of supported compilers can be found at [10].

2.2 Prerequisites

If the framework is to be compiled and executed on CERN's LXPLUS service, all build dependencies can be loaded automatically from the CVMFS file system.

The core framework is compiled separately from the individual modules and Allpix Squared has therefore only two required external dependencies:

- ROOT 6 [2]: ROOT is used for histogramming as well as coordinate transformations. In addition, some modules implement I/O using ROOT libraries. The latest stable release of ROOT 6 is recommended and older versions, such as ROOT 5.x, are not supported. Please refer to [11] for instructions on how to install ROOT. ROOT has several components of which the `GenVector` package is required to run Allpix Squared. This package is included in the default build. ROOT needs

to be built using C++17, which is accomplished by supplying the CMake flag `-DCMAKE_CXX_STANDARD=17`.

- Boost.Random 1.64.0 or later [8]: Random number generator and distribution library of the Boost project, used in order to get cross-platform portable, STL-compatible random number distributions. While STL random number generators are portable and guarantee to deliver the same random number sequence given the same seed, random distributions are not, and their implementation is platform-dependent leading to different simulation results with the same seed. Since the implementation of some random distributions (most notably of `boost::normal_distribution`) has changed in the past, a recent version is required.

For some modules, additional dependencies exist. For details about the dependencies and their installation see Chapter 8. The following dependencies are needed to compile the standard installation:

- Geant4 [1]: Simulates the desired particles and their interactions with matter, depositing charges in the detectors with the help of the constructed geometry. See the instructions in [12] for details on how to install the software. All Geant4 data sets are required to run the modules successfully, and Geant4 must be built using C++17. For multithreading to be possible, this must also be enabled in the Geant4 installation. It is recommended to enable the Geant4 Qt extensions to allow visualization of the detector setup and the simulated particle tracks. A recommended set of CMake flags to build a Geant4 package suitable for usage with Allpix Squared are:

  ```
  -DGEANT4_INSTALL_DATA=ON
  -DGEANT4_USE_GDML=ON
  -DGEANT4_USE_QT=ON
  -DGEANT4_USE_XM=ON
  -DGEANT4_USE_OPENGL_X11=ON
  -DCMAKE_CXX_STANDARD=17
  -DGEANT4_BUILD_MULTITHREADED=ON
  -DGEANT4_BUILD_BUILTIN_BACKTRACE=OFF
  ```

- Eigen3 [9]: Vector package used to perform Runge-Kutta integration, used in some of the charge carrier propagation modules. Eigen is available in almost all Linux distributions through the package manager. Otherwise it can be easily installed, comprising a header-only library.

Extra flags need to be set for building an Allpix Squared installation without these dependencies. Details about these configuration options are given in the Section 2.5.

2.3 Downloading the Source Code

The latest version of Allpix Squared can be downloaded from the CERN Gitlab repository [13]. For production environments it is recommended to only download and use tagged software versions, as many of the available git branches are considered development versions and might exhibit unexpected behavior.

For developers, it is recommended to always use the latest available version from the git `master` branch. The software repository can be cloned as follows:

```
git clone https://gitlab.cern.ch/allpix-squared/allpix-squared
cd allpix-squared
```

2.4 Initializing the Dependencies

Before continuing with the build, the necessary setup scripts for ROOT and Geant4 (unless a build without Geant4 modules is attempted) should be executed. In a Bash terminal on a private Linux machine this means executing the following two commands from their respective installation directories (replacing `<root_install_dir>` with the local ROOT installation directory and likewise for Geant4):

```
source <root_install_dir>/bin/thisroot.sh
source <geant4_install_dir>/bin/geant4.sh
```

On the CERN LXPLUS service, a standard initialization script is available to load all dependencies from the CVMFS infrastructure. This script should be executed as follows (from the main repository directory):

```
source etc/scripts/setup_lxplus.sh
```

2.5 Configuration via CMake

Allpix Squared uses the CMake build system to configure, build and install the core framework as well as all modules. An out-of-source build is recommended: this means CMake should not be directly executed in the source folder. Instead, a build folder should be created, from which CMake should be run. For a standard build without any additional flags this implies executing:

```
mkdir build
cd build
cmake ..
```

CMake can be run with several extra arguments to change the type of installation. These options can be set with `-D<option>` (see the end of this section for an example). Currently the following options are supported:

- `CMAKE_INSTALL_PREFIX`: The directory to use as a prefix for installing the binaries, libraries and data. Defaults to the source directory (where the folders `bin/` and `lib/` are added).
- `CMAKE_BUILD_TYPE`: Type of build to install, defaults to `RelWithDebInfo` (compiles with optimizations and debug symbols). Other possible options are `Debug` (for compiling with no optimizations, but with debug symbols and extended tracing using the Clang Address Sanitizer library) and `Release` (for compiling with full optimizations and no debug symbols).

- `MODEL_DIRECTORY`: Directory to install the internal models to. Defaults to not installing if the `CMAKE_INSTALL_PREFIX` is set to the directory containing the sources (the default). Otherwise the default value is equal to the directory `<CMAKE_INSTALL_PREFIX>/share/allpix/`. The install directory is automatically added to the model search path used by the geometry model parsers to find all of the detector models.
- `BUILD_TOOLS`: Enable or disable the compilation of additional tools such as the mesh converter. Defaults to `ON`.
- `BUILD_<ModuleName>`: If the specific module should be installed or not. Defaults to `ON` for most modules, however some modules with large additional dependencies such as LCIO [14] are disabled by default. This set of parameters allows to configure the build for minimal requirements.
- `BUILD_ALL_MODULES`: Build all included modules, defaulting to `OFF`. This overwrites any selection using the parameters described above.

An example of a custom debug build, without the `GeometryBuilderGeant4` module and with installation to a custom directory is shown below:

```
mkdir build
cd build
cmake -DCMAKE_INSTALL_PREFIX=../install/ \
      -DCMAKE_BUILD_TYPE=DEBUG \
      -DBUILD_GeometryBuilderGeant4=OFF ..
```

2.6 Compilation and Installation

Compiling the framework is now a single command in the build folder created earlier (replacing `<number_of_cores>` with the number of cores to use for compilation):

```
make -j<number_of_cores>
```

The compiled (non-installed) version of the executable can be found at `src/exec/allpix` in the folder. Running Allpix Squared directly without installing can be useful for developers. It is not recommended for normal users, because the correct library and model paths are only fully configured during installation.

To install the library to the selected installation location (defaulting to the source directory of the repository) requires the following command:

```
make install
```

The binary is now available as `bin/allpix` in the installation directory. The example configuration files are not installed as they should only be used as a starting point for your own configuration. They can however be used to check if the installation was successful. Running the allpix binary with the example configuration using `bin/allpix -c examples/example.conf` should pass the test without problems if a standard installation is used.

2.7 Docker Images

Docker images are provided for the framework to allow anyone to run simulations without the need of installing Allpix Squared on their system. The only required program is the Docker executable, all other dependencies are provided within the Docker images. In order to exchange configuration files and output data between the host system and the Docker container, a folder from the host system should be mounted to the container's data path `/data`, which also acts as the Docker `WORKDIR` location.

The following command creates a container from the latest Docker image in the project registry and start an interactive shell session with the `allpix` executable already in the `$PATH`. Here, the current host system path is mounted to the `/data` directory of the container.

```
docker run --interactive --tty                                    \
           --volume "$(pwd)":/data                                \
           --name=allpix-squared                                  \
           gitlab-registry.cern.ch/allpix-squared/allpix-squared \
           bash
```

Alternatively it is also possible to directly start the simulation instead of an interactive shell, e.g. using the following command:

```
docker run --tty --rm                                             \
           --volume "$(pwd)":/data                                \
           --name=allpix-squared                                  \
           gitlab-registry.cern.ch/allpix-squared/allpix-squared \
           "allpix -c my_simulation.conf"
```

where a simulation described in the configuration `my_simulation.conf` is directly executed and the container terminated and deleted after completing the simulation. This closely resembles the behavior of running Allpix Squared natively on the host system. Of course, any additional command line arguments known to the `allpix` executable described in Section 3.5 can be appended.

For tagged versions, the tag name should be appended to the image name, e.g. `gitlab-registry.cern.ch/allpix-squared/allpix-squared:v2.2.2`, and a full list of available Docker containers is provided via the project's container registry [15]. A short description of how Docker images for this project are built can be found in Section 11.5.

2.8 Releases on CVMFS

For each release a binary tarball is created, which is published to CERN's VM file system (CVMFS) [16]. They can be used to run Allpix Squared without building it beforehand. Compared to the Docker images mentioned in Section 2.7, which can run on any operating system, binary releases are tied to a specific operating system.

Binaries for Allpix Squared are currently provided for CentOS 7 (both in a GCC and LLVM variant), CentOS 8 (GCC) and MacOS. Details on the deployment process are given in Setcion 11.4.

To use Allpix Squared from CVMFS, run:

```
source /cvmfs/clicdp.cern.ch/software/allpix-squared/<version>/<system-
↪ specifier>/setup.sh
```

Where `<version>` should be replaced with the desired Allpix Squared version (e.g. `2.4.0`) and `<system-specifier>` with the specifier for the system CVMFS is running on (e.g. `x86_64-centos7-gcc11-opt`).

To verify if Allpix Squared is working, you can run `allpix --version`.

3 Getting Started

This Getting Started guide is written with a default installation in mind, meaning that some parts may not apply if a custom installation was used. When the `allpix` binary is used, this refers to the executable installed in `bin/allpix` in the installation path. It is worth noting that before running any Allpix Squared simulation, ROOT and (in most cases) Geant4 should be initialized. Refer to Section 2.4 for instructions on how to load these libraries.

3.1 Configuration Files

The framework is configured with simple human-readable configuration files. The configuration format is described in detail in Section 4.3. It consists of several section headers within `[` and `]` brackets, and a section without header at the start. Each of these sections contains a set of key/value pairs separated by the `=` character. Comments are indicated using the hash symbol (`#`).

The framework has the following three required layers of configuration files:

- The **main** configuration: The most important configuration file and the file that is passed directly to the binary. Contains both the global framework configuration and the list of modules to instantiate together with their configuration. An example can be found in the repository at `examples/example.conf`. More details and a more thorough example are found in Section 3.2, several advanced simulation chain configurations are presented in Chapter 9.

- The **geometry** configuration: It is passed to the framework to determine the detector setup and passive materials. Describes the detector setup, containing the position, orientation and model type of all detectors. Optionally, passive materials can be added to this configuration. Examples are available in the repository at `examples / example_detector.conf` or `examples/example_detector_passive.conf`. Introduced in Section 3.3.

- The detector **model** configuration: Contains the parameters describing a particular type of detector. Several models are already provided by the framework, but new types of detectors can easily be added. See `models/test.conf` in the repository for an example. Please refer to Section 10.5 for more details about adding new models.

3.1.1 Parsing Types and Units

The Allpix Squared framework supports the use of a variety of types for all configuration values. The module specifies how the value type should be interpreted. An error will be raised if either the key is not specified in the configuration file, the conversion to the desired type is not possible, or if the given value is outside the domain of possible options. Please refer to the Chapter 8 for the list of module parameters and their types. Parsing the value roughly follows common-sense (more details can be found in Section 4.3). A few special rules do apply:

- If the value is a **string**, it may be enclosed by a single pair of double quotation marks (`"`), which are stripped before passing the value to the modules. If the string is not enclosed by quotation marks, all whitespace before and after the value is erased. If the value is an array of strings, the value is split at every whitespace or comma (`,`) that is not enclosed in quotation marks.

- If the value is a **boolean**, either numerical (`0`, `1`) or textual (`false`, `true`) representations are accepted.

- If the value is a **relative path**, that path will be made absolute by adding the absolute path of the directory that contains the configuration file where the key is defined.

- If the value is an **arithmetic** type, it may have a suffix indicating the unit. The list of base units is given in the table below.

Quantity	Default unit	Auxiliary units
Unity	1	-
Length	mm (millimeter)	nm (nanometer), um (micrometer), cm (centimeter), dm (decimeter), m (meter), km (kilometer)
Time	ns (nanosecond)	ps (picosecond), us (microsecond), ms (millisecond), s (second)
Energy	MeV (mega-electronvolt)	eV (electronvolt), keV (kiloelectronvolt), GeV (gigaelectronvolt)
Temperature	K (kelvin)	-
Charge	e (elementary charge)	ke (kiloelectrons), fC (femtocoulomb), C (coulomb)
Voltage	MV (megavolt)	V (volt), kV (kilovolt)
Magnetic field strength	kT (kilotesla)	T (tesla), mT (millitesla)
Angle	rad (radian)	deg (degree), mrad (milliradian)
Radiation fluence	Neq (1-MeV neutron-equivalent)	-

Warning: If no units are specified, values will always be interpreted in the base units of the framework. In some cases this can lead to unexpected results. E.g. specifying a bias voltage as `bias_voltage = 50` results in an applied voltage of 50 MV. Therefore it is strongly recommended to always specify units in the configuration files.

The internal base units of the framework are not chosen for user convenience but for maximum precision of the calculations and in order to avoid the necessity of conversions in the code.

Combinations of base units can be specified by using the multiplication sign `*` and the division sign `/` that are parsed in linear order (thus $\frac{Vm}{s^2}$ should be specified as `V*m/s/s`). The framework assumes the default units if the unit is not explicitly specified. It is recommended to always specify the unit explicitly for all parameters that are not dimensionless as well as for angles.

Examples of specifying key/values pairs of various types are given below:

```
# All whitespace at the front and back is removed
first_string =   string_without_quotation
# All whitespace within the quotation marks is preserved
second_string = "  string with quotation marks  "
# Keys are split on whitespace and commas
string_array = "first element" "second element","third element"
# Elements of matrices with more than one dimension are separated
# using square brackets
string_matrix_3x3 = [["1","0","0"], ["0","cos","-sin"], ["0","sin",cos]]
# If the matrix is of dimension 1xN, the outer brackets have to be
# added explicitly
integer_matrix_1x3 = [[10, 11, 12]]
# Integer and floats can be specified in standard formats
int_value = 42
float_value = 123.456e9
# Units can be passed to arithmetic types
energy_value = 1.23MeV
time_value = 42ns
# Units are combined in linear order without grouping or implicit
 ↪  brackets
acceleration_value = 1.0m/s/s
# Thus the quantity below is the same as 1.0deg*kV*K/m/s
random_quantity = 1.0deg*kV/m/s*K
# Relative paths are expanded to absolute paths, e.g. the following
 ↪  value
# will become "/home/user/test" if the configuration file is located
# at "/home/user"
output_path = "test"
# Booleans can be represented in numerical or textual style
my_switch = true
my_other_switch = 0
```

Note: In some places, providing configuration variables with units is mandatory. In case the respective input should be interpreted as base units, or without units such as a weighting potential, the parameter can be provided as empty string, i.e. `observable_units = ""`.

3.2 Main Configuration

The main configuration consists of a set of sections specifying the modules used. All modules are executed in the *linear* order in which they are defined. There are a few section names which have a special meaning in the main configuration, namely the following:

- The **global** (framework) header sections: These are all zero-length section headers (including the one at the beginning of the file) and all sections marked with the header `[Allpix]` (case-insensitive). These are combined and accessed together as the global configuration, which contain all parameters of the framework itself (see Section 3.4 for details). All key-value pairs defined in this section are also inherited by all individual configurations as long the key is not defined in the module configuration itself.

- The **ignore** header sections: All sections with name `[Ignore]` (case-insensitive) are ignored. Key-value pairs defined in the section as well as the section itself are discarded by the parser. These section headers are useful for quickly enabling and disabling individual modules by replacing their actual name by an ignore section header.

All other section headers are used to instantiate modules of the respective name. Installed modules are loaded automatically. If problems arise please review the loading rules described in Section 4.4.

Modules can be specified multiple times in the configuration files, depending on their type and configuration. The type of the module determines how the module is instantiated:

- If the module is **unique**, it is instantiated only a single time irrespective of the number of detectors. These kinds of modules should only appear once in the whole configuration file unless different inputs and outputs are used, as explained in Section 4.7.

- If the module is **detector**-specific, it is instantiated once for every detector it is configured to run on. By default, an instantiation is created for all detectors defined in the detector configuration file (see Section 3.3, lowest priority) unless one or both of the following parameters are specified:

  - `name`: An array of detector names the module should be executed for. Replaces all global and type-specific modules of the same kind (highest priority).

- `type`: An array of detector types the module should be executed for. Instantiated after considering all detectors specified by the name parameter above. Replaces all global modules of the same kind (medium priority).

  Within the same module, the order of the individual instances in the configuration file is irrelevant.

A valid example configuration using the detector configuration above is:

```
# Key is part of the empty section and therefore the global
 ↪  configuration
string_value = "example1"
# The location of the detector configuration is a global parameter
detectors_file = "manual_detector.conf"
# The Allpix section is also considered global and merged with the above
[Allpix]
another_random_string = "example2"

# First run a unique module
[MyUniqueModule]
# This module takes no parameters
# [MyUniqueModule] cannot be instantiated another time

# Then run detector modules on different detectors
# First run a module on the detector of type Timepix
[MyDetectorModule]
type = "timepix"
int_value = 1
# Replace the module above for `dut` with a specialized version
# It does not inherit any parameters from earlier modules
[MyDetectorModule]
name = "dut"
int_value = 2
# Run the module on the remaining unspecified detector (`telescope1`)
[MyDetectorModule]
# int_value is not specified, so it uses the default value
```

3.3 Detector Configuration

The detector configuration consists of a set of sections describing the detectors in the setup. Each section starts with a header describing the name used to identify the detector; all names are required to be unique. Every detector has to contain all of the following parameters:

- A string referring to the `type` of the detector model. The model should exist in the search path as described in Section 5.2.
- The 3-dimensional `position` in the world frame in the order x, y, z. See Section 5.1 for details.

- The `orientation` specified as X-Y-Z extrinsic Euler angles. This means the detector is rotated first around the world's X-axis, then around the world's Y-axis and then around the world's Z-axis. Alternatively the orientation can be set as Z-Y-X or Z-X-Z extrinsic Euler angles, refer to section Section 5.1 for details.

In addition to these required parameters, the following parameters allow to randomly misalign the respective detector from its initial position. The values are interpreted as width of a normal distribution centered around zero. In order to reproduce misalignments, a fixed random seed for the framework core can be used as explained in Section 3.4. Misalignment can be introduced both for shifts along the three global axes and the three rotations angles with the following parameters:

- The parameter `alignment_precision_position` allows the specification of the alignment precision along the three global axes. Each value represents the Gaussian width with which the detector will be randomly misaligned along the corresponding axis.
- The parameter `alignment_precision_orientation` allows to specify the alignment precision in the three rotation angles defined by the `orientation` parameter. The misalignments are added to the individual angles before combining them into the final rotation as defined by the `orientation_mode` parameter.

The optional parameter `role` accepts the values `active` for detectors and `passive` for passive elements in the setup. If no value is given, `active` is taken as the default value.

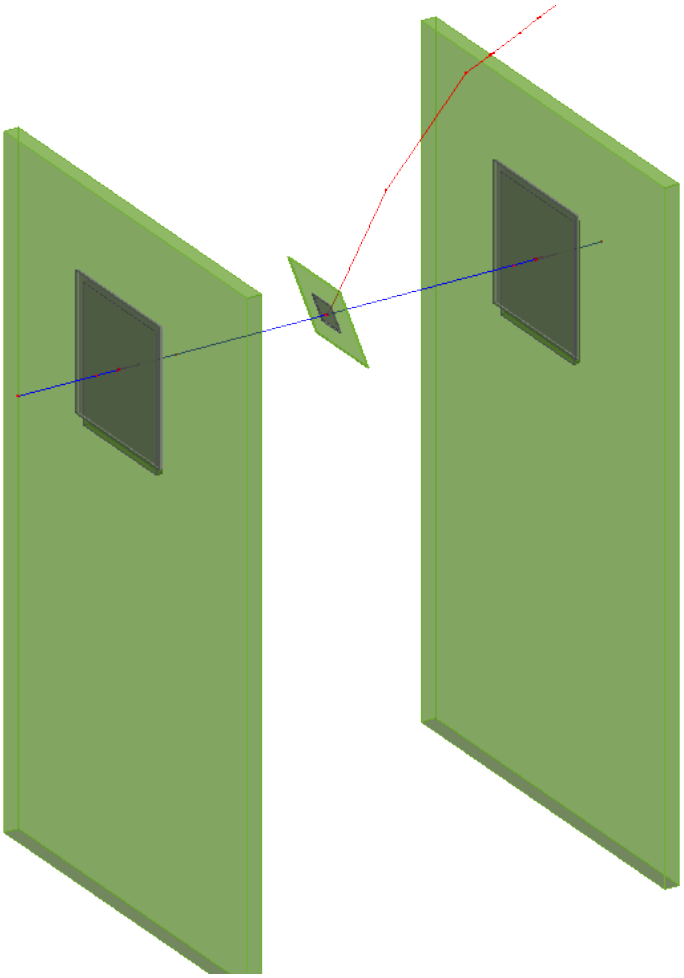
Furthermore it is possible to specify certain parameters of the detector, which is explained in more detail in Section 5.2. This allows to quickly adapt e.g. the sensor thickness of a certain detector without altering the actual detector model file.

An example configuration file describing a setup with one CLICpix2 detector and two Timepix [17] models is the following:

```
# Placement of first detector, named "telescope1"
[telescope1]
# Type to the detector is the "timepix" model
type = "timepix"
# Position the detector at the origin of the world frame
position = 0 0 0mm
# Default orientation: perpendicular to the incoming beam
orientation = 0 0 0

# Placement of the second detector, the "DUT (device under test)
[dut]
# Detector model is "clicpix2"
type = "clicpix2"
# Position is downstream of "telescope1":
position = 100um 100um 25mm
# Rotated by 20 degrees around the world x-axis
orientation = 20deg 0 0

# Third detector is downstream "telescope2"
[telescope2]
```

```
# Detector type again is "timepix"
type = "timepix"
# Placement 50 mm downstream of the first detector
position = 0 0 50mm
# Default orientation
orientation = 0 0 0
```

This configuration is used in the rest of this chapter for explaining concepts. A visualization of the setup is given below.



Visualization of a Pion passing through the telescope setup defined in the detector configuration file. A secondary particle is produced in the material of the detector in the center.

3.3.1 Passive material configuration

Descriptions of passive materials can be added to the detector setup via a set of sections, with a syntax similar to the detector configuration. Passive geometry entries are identified by the `role` parameter set to `passive`. Each section starts with a header describing the name used to identify the passive material; all names are required to be unique.

Every passive material has to contain all of the following parameters:

- The `position` and `orientation` of the material as described for the detector, see Section 3.3.
- A string referring to the `type` of the passive material. The model should be interpreted by the module constructing the passive material, for example the `GeometryBuilderGeant4` module.
- A string referring to the `material` of the passive material. The materials for the `GeometryBuilderGeant4` module are defined in the module documentation.
- A set of size parameters specific for the model that is chosen. All size parameters that describe the total length of something are placed such that half of this total length extends from each side of the given `position`. If a parameter describes the radius, this means the radius will extend from the `position` on both sides, making its total size two times the radius in the given direction. The size parameters for the specific models in the `GeometryBuilderGeant4` module are described in the module documentation.

In addition, an optional string referring to the `mother_volume`, which defines another passive material the volume will be placed in, can be specified.

Note: If a mother volume is chosen, the position defined in the configuration file will be relative to the center of the mother volume. An error will be given if the specified mother volume is too small for the specified size or position of this volume. Per default, the mother volume is the world frame.

Note: If the `mother_volume` is a hollow material, only the non-hollow part of the material is considered part of the material. Placing a passive volume in the hollow part requires a different `mother_volume`.

Similar to the detector configuration, the parameters `orientation_mode` (see Section 5.1), `alignment_precision_position` and `alignment_precision_orientation` (see Section 3.3) can be used optionally to define the rotation order and a possible misalignment of passive materials.

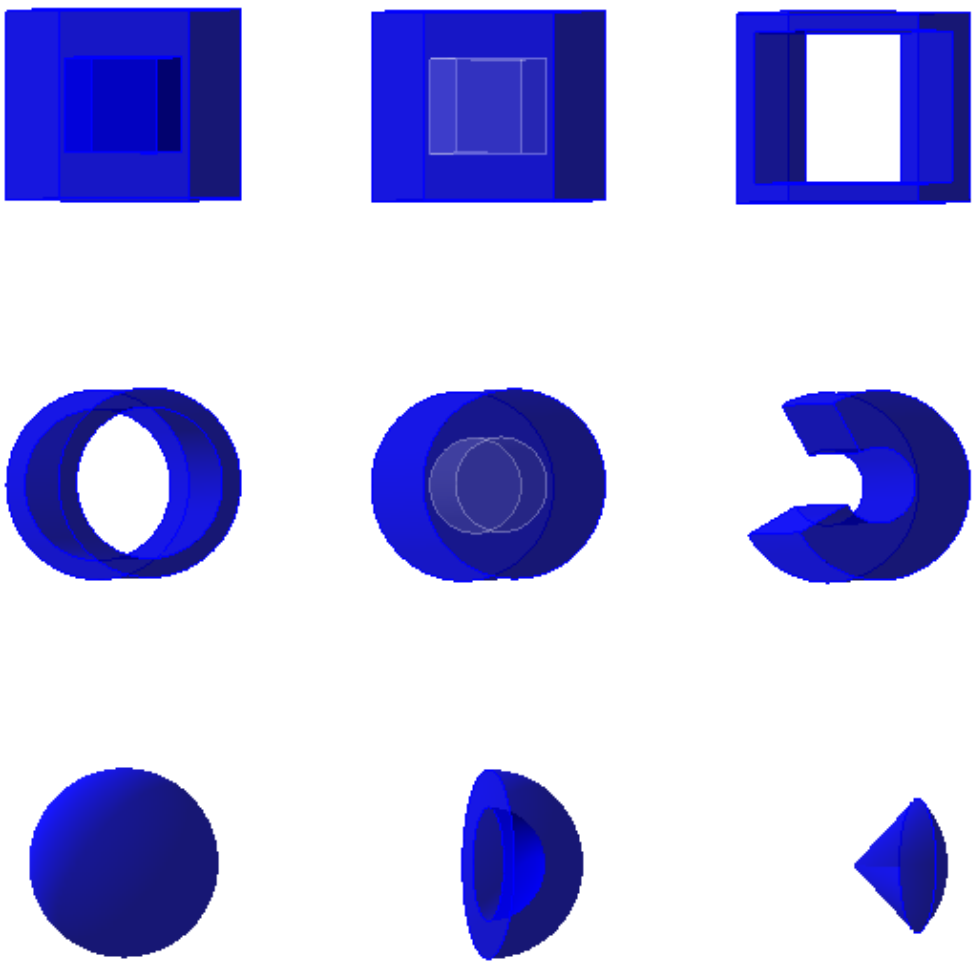
An example configuration file describing a set of passive materials with different configuration options is the following:

```
# Placement of a box made of lead
[box1]
type = "box"
size = 100mm 100mm 100mm
position = 200mm 200mm 0mm
orientation = 0 0deg 0deg
material = "lead"
role = "passive"
```

```
# Placement of a box made of lead
[box2]
type = "box"
size = 100mm 100mm 100mm
position = 0mm 200mm 0mm
orientation = 0 0deg 0deg
material = "lead"
role = "passive"

# Placement of a box made of lead, with a hollow opening
[box3]
type = "box"
size = 100mm 100mm 100mm
inner_size = 80mm 80mm 100mm
position = -200mm 200mm 0mm
orientation = 0 0deg 0deg
material = "lead"
role = "passive"

# Placement of a box made of aluminum, inside box1
[box4]
type = "box"
size = 50mm 50mm 50mm
position = 0mm 0mm -0mm
orientation = 0 0deg 0deg
material = "aluminum"
mother_volume = box1
role = "passive"

# Placement of a box made of the world material, inside box2
[box5]
type = "box"
size = 50mm 50mm 50mm
position = 0mm 0mm -0mm
orientation = 0 0deg 0deg
material = "world_material"
mother_volume = box2
role = "passive"

# Placement of a cylinder made of lead, with a hollow opening
[cylinder1]
type = "cylinder"
outer_radius = 50mm
inner_radius = 40mm
length = 100mm
position = 200mm 0mm 0mm
orientation = 0 0deg 0deg
material = "lead"
role = "passive"
```

```
# Placement of a cylinder made of lead
[cylinder2]
type = "cylinder"
outer_radius = 50mm
length = 100mm
position = 0mm 0mm 0mm
orientation = 0 0deg 0deg
material = "lead"
role = "passive"

# Placement of a cylinder made of lead, with a hollow opening, starting
 ↪  the building at an angle of 60deg and continue for 270deg
[cylinder3]
type = "cylinder"
outer_radius = 50mm
inner_radius = 20mm
length = 100mm
starting_angle = 60deg
arc_length = 270deg
position = -200mm 0mm 0mm
orientation = 0 0deg 0deg
material = "lead"
role = "passive"

# Placement of a cylinder made of the world material, inside cylinder2
[cylinder4]
type = "cylinder"
outer_radius = 25mm
length = 50mm
position = 0mm 0mm 0mm
orientation = 0 0deg 0deg
material = "world_material"
mother_volume = cylinder2
role = "passive"

# Placement of a sphere made of lead
[sphere1]
type = "sphere"
outer_radius = 50mm
position = 200mm -200mm 0mm
orientation = 0 0deg 0deg
material = "lead"
role = "passive"

# Placement of a sphere made of lead,  with a hollow opening, starting
 ↪  the building at a phi angle of 90deg and continue for 180deg.
[sphere2]
type = "sphere"
```

```
outer_radius = 50mm
inner_radius = 30mm
starting_angle_phi = 90deg
arc_length_phi = 180deg
position = 0mm -200mm 0mm
orientation = 0 0deg 0deg
material = "lead"
role = "passive"

# Placement of a sphere made of lead, starting the building at a theta
 ↪  angle of 0deg and continue for 45deg.
[sphere3]
type = "sphere"
outer_radius = 50mm
arc_length_theta = 45deg
position = -200mm -200mm 0mm
orientation = 0 -90deg 0deg
material = "lead"
role = "passive"
```



Visualization of the setup described in the geometry file.

3.4 Framework Parameters

The Allpix Squared framework provides a set of global parameters which control and alter its behavior:

- `detectors_file`: Location of the file describing the detector configuration (introduced in Section 3.3). The only *required* global parameter: the framework will fail to start if it is not specified.
- `number_of_events`: Determines the total number of events the framework should simulate. Defaults to one (simulating a single event).
- `skip_events`: A number of events (and therefore event seeds) to be skipped at start of the run. After skipping, the full `number_of_events` will be processed starting from the new event seed. Defaults to zero, i.e. starting with the first event seed.
- `root_file`: Location relative to the `output_directory` where the ROOT output data of all modules will be written to. The file extension `.root` will be appended if not present. Default value is `modules.root`. Directories within the ROOT file will be created automatically for all module instantiations.
- `log_level`: Specifies the lowest log level which should be reported. Possible values are `FATAL`, `STATUS`, `ERROR`, `WARNING`, `INFO`, `DEBUG`, `TRACE` and `PRNG` where all options are case-insensitive. Defaults to the `WARNING` level. More details and information about the log levels, including how to change them for a particular module, can be found in Section 3.8. Can be overwritten by the `-v` parameter on the command line (see Section 3.5).
- `log_format`: Determines the log message format to display. Possible options are `SHORT`, `DEFAULT` and `LONG`, where all options are case-insensitive. More information can be found in Section 3.8.
- `log_file`: File where the log output should be written to in addition to printing to the standard output (usually the terminal). Only writes to standard output if this option is not provided. Another (additional) location to write to can be specified on the command line using the `-l` parameter (see Section 3.5).
- `output_directory`: Directory to write all output files into. Subdirectories are created automatically for all module instantiations. This directory will also contain the `root_file` specified via the parameter described above. Defaults to the current working directory with the subdirectory `output/` attached.
- `purge_output_directory`: Decides whether the content of an already existing output directory is deleted before a new run starts. Defaults to `false`, i.e. files are kept but will be overwritten by new files created by the framework.
- `deny_overwrite`: Forces the framework to abort the run and throw an exception when attempting to overwrite an existing file. Defaults to `false`, i.e. files are overwritten when requested. This setting is inherited by all modules, but can be overwritten in the configuration section of each of the modules.

- `random_seed`: Seed for the global random seed generator used to initialize seeds for module instantiations. The 64-bit Mersenne Twister `mt19937_64` from the C++ Standard Library is used to generate seeds. A random seed from multiple entropy sources will be generated if the parameter is not specified. Can be used to reproduce an earlier simulation run.

- `random_seed_core`: Optional seed used for pseudo-random number generators in the core components of the framework. If not set explicitly, the value `random_seed + 1` is used.

- `library_directories`: Additional directories to search for module libraries, before searching the default paths. See Section 4.4 for more information.

- `model_paths`: Additional files or directories from which detector models should be read besides the standard search locations.

- `performance_plots`: Enable the creation of performance plots showing the processing time required per event both for individual modules and the full module stack. Defaults to `false`.

- `multithreading`: Enable multithreading for the framework. Defaults to `true`. More information about multithreading can be found in Section 4.3.

- `workers`: Specify the number of workers to use in total, should be strictly larger than zero. Only used if `multithreading` is set to `true`. Defaults to the number of native threads available on the system minus one, if this can be determined, otherwise one thread is used.

- `buffer_per_worker`: Specify the buffer depth available per worker for buffered modules to cache partially processed events until execution in the correct order can be guaranteed (see Section 4.10). Defaults to `256`.

3.5 The allpix Executable

The `allpix` executable functions as the interface between the user and the framework. It is primarily used to provide the main configuration file, but also allows to add and overwrite options from the main configuration file. This is both useful for quick testing as well as for batch processing of simulations.

The executable handles the following arguments:

- `-c <file>`: Specifies the configuration file to be used for the simulation, relative to the current directory. This is the only *required* argument, the simulation will fail to start if this argument is not given.

- `-l <file>`: Specify an additional location to forward log output to, besides standard output and the location specified in the framework parameters explained in Section 3.4.

- `-v <level>`: Sets the global log verbosity level, overwriting the value specified in the configuration file described in Section 3.4. Possible values are `FATAL`, `STATUS`, `ERROR`, `WARNING`, `INFO` and `DEBUG`, `TRACE` and `PRNG` where all options are case-insensitive. The module specific logging level introduced in Section 3.8 is not overwritten.
- `-j <workers>`: Enables multithreaded event processing with the given number of worker threads. This is equivalent to passing the framework parameters `-o multithreading=true -o workers=<workers>` to the executable.
- `--version`: Prints the version and build time of the executable and terminates the program.
- `-o <option>`: Passes extra framework or module options which are added and overwritten in the main configuration file. This argument may be specified multiple times, to add multiple options. Options are specified as key/value pairs in the same syntax as used in the configuration files (refer to Section 4.2 for more details), but the key is extended to include a reference to a configuration section or instantiation in shorthand notation. There are three types of keys that can be specified:
  - Keys to set **framework parameters**: These have to be provided in exactly the same way as they would be in the main configuration file (a section does not need to be specified). An example to overwrite the standard output directory would be `allpix -c <file> -o output_directory="run123456"`.
  - Keys for **module configurations**: These are specified by adding a dot (`.`) between the module and the actual key as it would be given in the configuration file (thus `module.key`). An example to overwrite the deposited particle to a positron would be `allpix -c <file> -o DepositionGeant4.particle_type="e+"`.
  - Keys to specify values for a particular **module instantiation**: The identifier of the instantiation and the name of the actual key are split by a dot (`.`), in the same way as for keys for module configurations (thus `identifier.key`). The unique identifier for a module can contains one or more colons (`:`) to distinguish between various instantiations of the same module. The exact name of an identifier depends on the name of the detector and the optional input and output name. Those identifiers can be extracted from the logging section headers. An example to change the temperature of propagation for a particular instantiation for a detector named `dut` could be `allpix -c <file> -o GenericPropagation:dut.temperature=273K`.

  Note that only the single argument directly following the `-o` is interpreted as the option. If there is whitespace in the key/value pair this should be properly enclosed in quotation marks to ensure the argument is parsed correctly.
- `-g <option>`: Passes extra detector options which are added and overwritten in the detector configuration file. This argument can be specified multiple times, to add multiple options. The options are parsed in the same way as described above for module options, but only one type of key can be specified to overwrite an option for a single detector. These are specified by adding a dot (`.`) between the detector and the actual key as it would be given in the detector configuration file (thus `detector.key`). This method also works for customizing detector models

as described in Section 5.2. An example to overwrite the sensor thickness for a particular detector named `detector1` to `50um` would be `allpix -c <file> -g detector1.sensor_thickness=50um`.

No interaction with the framework is possible during the simulation. Signals can however be send using keyboard shortcuts to terminate the simulation, either gracefully or with force. The executable understand the following signals:

- SIGINT (`CTRL+C`): Request a graceful shutdown of the simulation. This means the currently processed events are finished, while events placed on the buffer as well as all additionally requested events from the configuration file are ignored. After finishing the current events, the finalization stage is run for every module to ensure that the simulation terminates properly. This signal can be useful when too many events are specified and the simulation takes too long to finish entirely, but the output generated so far should still be kept.
- SIGTERM: Same as SIGINT, request a graceful shutdown of the simulation. This signal is emitted e.g. by the `kill` command or by cluster computing schedulers to ask for a termination of the job.
- SIGQUIT (`CTRL+\`): Forcefully terminates the simulation. It is not recommended to use this signal as it will normally lead to the loss of all generated data. This signal should only be used when graceful termination is for any reason not possible.

3.6 Setting up the Simulation Chain

In the following, the framework parameters are used to set up a fully functional simulation. Module parameters are shortly introduced when they are first used. For more details about the module parameters, the respective module documentation in Chapter 8 should be consulted. A typical simulation in Allpix Squared will contain the following components:

- The **geometry builder**, responsible for creating the external Geant4 geometry from the internal geometry. In this document, *internal geometry* refers to the detector parameters used by Allpix Squared for coordinate transformations and conversions throughout the simulation, while *external geometry* refers to the constructed Geant4 geometry used for charge carrier deposition (and possibly visualization).
- The **deposition** module that simulates the particle beam creating charge carriers in the detectors using the provided physics list (containing a description of the simulated interactions) and the geometry created above.
- A **propagation** module that propagates the charges through the sensor.
- A **transfer** module that transfers the charges from the sensor electrodes and assigns them to a pixel of the readout electronics.
- A **digitizer** module which converts the charges in the pixel to a detector hit, simulating the front-end electronics response.
- An **output** module, saving the data of the simulation. The Allpix Squared standard file format is a ROOT `TTree`, which is described in detail in Section 4.7.

In this example, charge carriers will be deposited in the three sensors defined in the detector configuration file from Section 3.3. All charge carriers deposited in the different sensors will be propagated and digitized. Finally, monitoring histograms for the device under test (DUT) will be recorded in the framework's main ROOT file and all simulated objects, including the entry and exit positions of the simulated particles (Monte Carlo truth), will be stored in a ROOT file using the Allpix Squared format. An example configuration file implementing this would look like:

```
# Global configuration
[Allpix]
# Simulate a total of 5 events
number_of_events = 5
# Use the short logging format
log_format = "SHORT"
# Location of the detector configuration
detectors_file = "manual_detector.conf"

# Read and instantiate the detectors and construct the Geant4 geometry
[GeometryBuilderGeant4]

# Initialize physics list and particle source
[DepositionGeant4]
# Use a Geant4 physics lists with EMPhysicsStandard_option3 enabled
physics_list = FTFP_BERT_LIV
# Use a charged pion as particle
particle_type = "pi+"
# Set the energy of the particle
source_energy = 120GeV
# Origin of the beam
source_position = 0 0 -12mm
# The direction of the beam
beam_direction = 0 0 1
# Use a single particle in a single event
number_of_particles = 1

# Propagate the charge carriers through the sensor
[GenericPropagation]
# Set the temperature of the sensor
temperature = 293K
# Propagate multiple charges at once
charge_per_step = 50

# Transfer the propagated charges to the pixels
[SimpleTransfer]
max_depth_distance = 5um

# Digitize the propagated charges
[DefaultDigitizer]
# Noise added by the readout electronics
electronics_noise = 110e
```

```
# Threshold for a hit to be detected
threshold = 600e
# Threshold dispersion
threshold_smearing = 30e
# Noise added by the digitisation
qdc_smearing = 100e

# Save histograms to the ROOT output file
[DetectorHistogrammer]
# Save histograms for the "dut" detector only
name = "dut"

# Store all simulated objects to a ROOT file with TTrees
[ROOTObjectWriter]
# File name of the output file
file_name = "allpix-squared-output"
# Ignore initially deposited charges and propagated carriers:
exclude = DepositedCharge, PropagatedCharge
```

This configuration is available in the repository [13] at `etc/manual.conf`. The detector configuration file can be found at `etc/manual_detector.conf`.

The simulation is started by passing the path of the main configuration file to the `allpix` executable as follows:

```
allpix -c etc/manual.conf
```

The detector histograms such as the hit map are stored in the ROOT file `output/modules.root` in the TDirectory `DetectorHistogrammer/`.

If problems occur when exercising this example, it should be made sure that an up-to-date and properly installed version of Allpix Squared is used (see the installation instructions in Chapter 2). If modules or models fail to load, more information about potential issues with the library loading can be found in the detailed framework description in Section 4.4.

3.7 Extending the Simulation Chain

In the following, a few basic modules will be discussed which may be of use during a first simulation.

3.7.1 Visualization

Displaying the geometry and the particle tracks helps both in checking and interpreting the results of a simulation. Visualization is fully supported through Geant4, supporting all the options provided by Geant4 [18]. Using the Qt viewer with OpenGL driver is the recommended option as long as the installed version of Geant4 is built with Qt support enabled.

To add the visualization, the `VisualizationGeant4` section should be added at the end of the configuration file. An example configuration with some useful parameters is given below:

```
[VisualizationGeant4]
# Use the Qt gui
mode = "gui"

# Set opacity of the detector models (in percent)
opacity = 0.4
# Set viewing style (alternative is 'wireframe')
view_style = "surface"

# Color trajectories by charge of the particle
trajectories_color_mode = "charge"
trajectories_color_positive = "blue"
trajectories_color_neutral = "green"
trajectories_color_negative = "red"
```

If Qt is not available, a VRML viewer can be used as an alternative, however it is recommended to reinstall Geant4 with the Qt viewer included as it offers the best visualization capabilities. The following steps are necessary in order to use a VRML viewer:

- A VRML viewer should be installed on the operating system. Good options are FreeWRL or OpenVRML.
- Subsequently, two environmental parameters have to be exported to the shell environment to inform Geant4 about the configuration: `G4VRMLFILE_VIEWER` should point to the location of the viewer executable and should `G4VRMLFILE_MAX_FILE_NUM` typically be set to 1 to prevent too many files from being created.
- Finally, the configuration section of the visualization module should be altered as follows:

  ```
  [VisualizationGeant4]
  # Do not start the Qt gui
  mode = "none"
  # Use the VRML driver
  driver = "VRML2FILE"
  ```

More information about all possible configuration parameters can be found in the `VisualizationGeant4` documentation.

3.7.2 Electric Fields

By default, detectors do not have an electric field associated with them, and no bias voltage is applied. A field can be added to each detector using the `ElectricFieldReader` module.

The section below calculates a linear electric field for every point in active sensor volume based on the depletion voltage of the sensor and the applied bias voltage. The sensor is

always depleted from the implant side. The direction of the electric field depends on the sign of the bias voltage as described in the `ElectricFieldReader` documentation.

```
# Add an electric field
[ElectricFieldReader]
# Set the field type to `linear`
model = "linear"
# Applied bias voltage to calculate the electric field from
bias_voltage = -50V
# Depletion voltage at which the given sensor is fully depleted
depletion_voltage = -10V
```

Allpix Squared also provides the possibility to utilize a full electrostatic TCAD simulation for the description of the electric field. In order to speed up the lookup of the electric field values at different positions in the sensor, the adaptive TCAD mesh has to be interpolated and transformed into a regular grid with configurable feature size before use. Allpix Squared comes with a converter tool which reads TCAD DF-ISE files from the sensor simulation, interpolates the field, and writes this out in an appropriate format. A more detailed description of the tool can be found in Section 14.2. An example electric field can be found in the repository [13] at `etc/example_electric_field.init`. A detailed description of supported field geometries and their mapping onto the sensor plane is provided in Section 4.5.

Electric fields can be attached to a specific detector using the standard syntax for detector binding. A possible configuration would be:

```
[ElectricFieldReader]
# Bind the electric field to the detector named `dut`
name = "dut"
# Specify that the model is provided as meshed electric field map format,
 ↪  e.g. converted from TCAD
model = "mesh"
# Name of the file containing the electric field
file_name = "example_electric_field.init"
```

3.7.3 Magnetic Fields

For simulating the detector response in the presence of a magnetic field with Allpix Squared, a constant, global magnetic field can be defined. By default, it is turned off. A field can be added to the whole setup using the unique module `MagneticFieldReader`, passing the field vector as parameter:

```
# Add a magnetic field
[MagneticFieldReader]
# Constant magnetic field (currently this is the default value)
model = "constant"
# Magnetic field vector
magnetic_field = 0mT 3.8T 0T
```

The global magnetic field is used by the interface to Geant4 and therefore exposes charged primary particles to the Lorentz force, and as a property of each detector present, enabling a Lorentz drift of the charge carriers in the active sensors, if supported by the used propagation modules. See the Chapter 8 for more information on the available propagation modules.

Currently, only constant magnetic fields can be applied. For all parameters, refer to the `MagneticFieldReader` documentation.

3.8 Logging and Verbosity Levels

Allpix Squared is designed to identify mistakes and implementation errors as early as possible and to provide the user with clear indications about the problem. The amount of feedback can be controlled using different log levels which are inclusive, i.e. lower levels also include messages from all higher levels. The global log level can be set using the global parameter `log_level`. The log level can be overridden for a specific module by adding the `log_level` parameter to the respective configuration section. The following log levels are supported:

- **FATAL**: Indicates a fatal error that will lead to direct termination of the application. Typically only emitted in the main executable after catching exceptions as they are the preferred way of fatal error handling (as discussed in Section 4.9). An example of a fatal error is an invalid configuration parameter.
- **STATUS**: Important information about the status of the simulation. Is only used for messages which have to be logged in every run such as the global seed for pseudo-random number generators and the current progress of the run.
- **ERROR**: Severe error that should not occur during a normal well-configured simulation run. Frequently leads to a fatal error and can be used to provide extra information that may help in finding the problem (for example used to indicate the reason a dynamic library cannot be loaded).
- **WARNING**: Indicate conditions that should not occur normally and possibly lead to unexpected results. The framework will however continue without problems after a warning. A warning is for example issued to indicate that an output message is not used and that a module may therefore perform unnecessary work.
- **INFO**: Information messages about the physics process of the simulation. Contains summaries of the simulation details for every event and for the overall simulation. Should typically produce maximum one line of output per event and module.
- **DEBUG**: In-depth details about the progress of the simulation and all physics details of the simulation. Produces large volumes of output per event, and should therefore only be used for debugging the physics simulation of the modules.
- **TRACE**: Messages to trace what the framework or a module is currently doing. Unlike the DEBUG level, it does not contain any direct information about the physics of the simulation but rather indicates which part of the module or framework is currently running. Mostly used for software debugging or determining performance bottlenecks in the simulations.

- **PRNG**: This level enables printing of every single pseudo-random number requested from any generator used in the framework. This can be useful in order to investigate random number distribution among threads and events.

Warning: It is not recommended to set the `log_level` higher than **WARNING** in a typical simulation as important messages may be missed. Setting too low logging levels should also be avoided since printing many log messages will significantly slow down the simulation.

The logging system supports several formats for displaying the log messages. The following formats are supported via the global parameter `log_format` or the individual module parameter with the same name:

- **SHORT**: Displays the data in a short form. Includes only the first character of the log level followed by the configuration section header and the message.
- **DEFAULT**: The default format. Displays system time, log level, section header and the message itself.
- **LONG**: Detailed logging format. Displays all of the above but also indicates source code file and line where the log message was produced. This can help in debugging modules.

More details about the logging system and the procedure for reporting errors in the code can be found in Section 4.8 and Section 4.9.

3.9 Storing Output Data

Storing the simulation output to persistent storage is of primary importance for subsequent reprocessing and analysis. Allpix Squared primarily uses ROOT for storing output data, given that it is a standard tool in High-Energy Physics and allows objects to be written directly to disk. The `ROOTObjectWriter` automatically saves all objects created in a `TTree` [19]. It stores separate trees for all object types and creates branches for every unique message name: a combination of the detector, the module and the message output name as described in Section 4.7. For each event, values are added to the leaves of the branches containing the data of the objects. This allows for easy histogramming of the acquired data over the total run using standard ROOT utilities.

Relations between objects within a single event are internally stored as ROOT TRefs [20], allowing retrieval of related objects as long as these are loaded in memory. An exception will be thrown when trying to access an object which is not in memory. Refer to Section 7.2 for more information about object history.

In order to save all objects of the simulation, a `ROOTObjectWriter` module has to be added with a `file_name` parameter to specify the file location of the created ROOT file in the global output directory. The file extension `.root` will be appended if not present. The default file name is `data`, i.e. the file `data.root` is created in the output directory. To replicate the default behaviour the following configuration can be used:

```
# The object writer listens to all output data
[ROOTObjectWriter]
# specify the output file (default file name is used if omitted)
file_name = "data"
```

The generated output file can be analyzed using ROOT macros. A simple macro for converting the results to a tree with standard branches for comparison is described in Section 14.3.

It is also possible to read object data back in, in order to dispatch them as messages to further modules. This feature is intended to allow splitting the execution of parts of the simulation into independent steps, which can be repeated multiple times. The tree data can be read using a `ROOTObjectReader` module, which automatically dispatches all objects to the correct module instances. An example configuration for using this module is:

```
# The object reader dispatches all objects in the tree
[ROOTObjectReader]
# path to the output data file, absolute or relative to the
 ↪  configuration file
file_name = "../output/data.root"
```

The Allpix Squared framework comes with a few more output modules which allow data storage in different formats, such as the `LCIOWriter` for the LCIO persistency event data model [14], the `RCEWriter` for the native RCE file format [21], or the `CorryvreckanWriter` for the Corryvreckan reconstruction framework data format. Consult Chapter 8 for all output modules.

4 Structure of the Framework

This chapter details the technical implementation of the Allpix Squared framework and is mostly intended to provide insight into the gearbox to potential developers and interested users.

4.1 Main Components

The framework consists of the following four main components that together form Allpix Squared:

1. **Core**: The core contains the internal logic to initialize the modules, provide the geometry, facilitate module communication and run the event sequence. The core keeps its dependencies to a minimum (it only relies on ROOT) and remains independent from the other components as far as possible. It is the main component discussed in this section.

2. **Modules**: A module is a set of methods which is executed as part of the simulation chain. Modules are built as separate libraries and loaded dynamically on demand by the core. The available modules and their parameters are discussed in detail in Chapter 8.

3. **Objects**: Objects form the data passed between modules using the message framework provided by the core. Modules can listen and bind to messages with objects they wish to receive. Messages are identified by the object type they are carrying, but can also be renamed to allow the direction of data to specific modules, facilitating more sophisticated simulation setups. Messages are intended to be read-only and a copy of the data should be made if a module wishes to change the data. All objects are compiled into a separate library which is automatically linked to every module. More information about the messaging system and the supported objects can be found in Section 4.6.

4. **Physics**: In many cases, several modules depend on the same underlying physics models. These models are separated from the modules themselves. The implemented physics models are described in Chapter 6.

5. **Tools**: Allpix Squared provides a set of header-only 'tools' and a shared library that allow access to common logic shared by various modules. Examples are the Runge-Kutta solver [22] implemented using the Eigen3 library and the set of template specializations for ROOT and Geant4 configurations. More information about the tools can be found in Chapter 14. This set of tools is different from the set of core utilities the framework itself provides, which is part of the core and explained in Section 4.8.

Finally, Allpix Squared provides an executable which instantiates the core of the framework, receives and distributes the configuration object and runs the simulation chain.

The chapter is structured as follows. Section 4.2 provides an overview of the architectural design of the core and describes its interaction with the rest of the Allpix Squared framework. The different subcomponents such as configuration, modules and messages are discussed in thereafter. The chapter closes with a description of the available framework tools in Section 4.8. Some C++ code will be provided in the text, but readers not interested may skip the technical details.

4.2 Architecture of the Core

The core is constructed as a light-weight framework which provides various subsystems to the modules. It contains the part of the software responsible for instantiating and running the modules from the supplied configuration file, and is structured around five subsystems, of which four are centered around a manager and the fifth contains a set of general utilities. The systems provided are:

1. **Configuration**: The configuration subsystem provides a configuration object from which data can be retrieved or stored, together with a TOML-like [23] parser to read configuration files. It also contains the Allpix Squared configuration manager which provides access to the main configuration file and its sections. It is used by the module manager system to find the required instantiations and access the global configuration. More information is given in Section 4.3.

2. **Module**: The module subsystem contains the base class of all Allpix Squared modules as well as the manager responsible for loading and executing the modules (using the configuration system). This component is discussed in more detail in Section 4.4.

3. **Geometry**: The geometry subsystem supplies helpers for the simulation geometry. The manager instantiates all detectors from the detector configuration file. A detector object contains the position and orientation linked to an instantiation of a particular detector model, itself containing all parameters describing the geometry of the detector. More details about geometry and detector models is provided in Chapter 5.

4. **Messenger**: The messenger is responsible for sending objects from one module to another. The messenger object is passed to every module and can be used to bind to messages to listen for. Messages with objects are also dispatched through the messenger as described in Section 4.6.

5. **Utilities**: The framework provides a set of utilities for logging, file and directory access, and unit conversion. An explanation on how to use of these utilities can be found in Section 4.8. A set of C++ exceptions is also provided in the utilities, which are inherited and extended by the other components. Proper use of exceptions, together with logging information and reporting errors, makes the framework easier to use and debug. A few notes about the use and structure of exceptions are provided in Section 4.9.

4.3 Configuration and Parameters

Individual modules as well as the framework itself are configured through configuration files, which all follow the same format. Explanations on how to use the various configuration files together with several examples have been provided in Section 3.1.

4.3.1 File format

Throughout the framework, a simplified version of TOML [23] is used as standard format for configuration files. The format is defined as follows:

1. All whitespace at the beginning or end of a line are stripped by the parser. In the rest of this format specification the *line* refers to the line with this whitespace stripped.
2. Empty lines are ignored.
3. Every non-empty line should start with either `#`, `[` or an alphanumeric character. Every other character should lead to an immediate parse error.
4. If the line starts with a hash character (`#`), it is interpreted as comment and all other content on the same line is ignored.
5. If the line starts with an open square bracket (`[`), it indicates a section header (also known as configuration header). The line should contain a string with alphanumeric characters and underscores, indicating the header name, followed by a closing square bracket (`]`), to end the header. After any number of ignored whitespace characters there could be a `#` character. If this is the case, the rest of the line is handled as specified in point 3. Otherwise there should not be any other character (except the whitespace) on the line. Any line that does not comply to these specifications should lead to an immediate parse error. Multiple section headers with the same name are allowed. All key-value pairs following this section header are part of this section until a new section header is started.
6. If the line starts with an alphanumeric character, the line should indicate a key-value pair. The beginning of the line should contain a string of alphabetic characters, numbers, dots (`.`), colons (`:`) and underscores (`_`), but it may only start with an alphanumeric character. This string indicates the key. After an optional number of ignored whitespace, the key should be followed by an equality sign (`=`). Any text between the `=` and the first `#` character not enclosed within a pair of single or double quotes (`'` or `"`) is known as the non-stripped string. Any character after the `#` is handled as specified in point 3. If the line does not contain any non-enclosed `#` character, the value ends at the end of the line instead. The 'value' of the key-value pair is the non-stripped string with all whitespace in front and at the end stripped. The value may not be empty. Any line that does not comply to these specifications should lead to an immediate parse error.
7. The value may consist of multiple nested dimensions which are grouped by pairs of square brackets (`[` and `]`). The number of square brackets should be properly balanced, otherwise an error is raised. Square brackets which should not be used for grouping should be enclosed in quotation marks. Every dimension is split at every whitespace sequence and comma character (`,`) not enclosed in quotation marks.

Implicit square brackets are added to the begin and end of the value, if these are not explicitly added. A few situations require explicit addition of outer brackets such as matrices with only one column element, i.e. with dimension 1xN.

8. The sections of the value which are interpreted as separate entities are named elements. For a single value the element is on the zeroth dimension, for arrays on the first dimension and for matrices on the second dimension. Elements can be forced by using quotation marks, either single or double quotes (' or "). The number of both types of quotation marks should be properly balanced, otherwise an error is raised. The conversion to the elements to the actual type is performed when accessing the value.

9. All key-value pairs defined before the first section header are part of a zero-length empty section header.

4.3.2 Accessing parameters

Values are accessed via the configuration object. In the following example, the key is a string called `key`, the object is named `config` and the type `TYPE` is a valid C++ type the value should represent. The values can be accessed via the following methods:

```
// Returns true if the key exists and false otherwise
config.has("key")
// Returns the number of keys found from the provided initializer list:
config.count({"key1", "key2", "key3"});
// Returns the value in the given type, throws an exception if not
 ↪  existing or a conversion to TYPE is not possible
config.get<TYPE>("key")
// Returns the value in the given type or the provided default value if
 ↪  it does not exist
config.get<TYPE>("key", default_value)
// Returns an array of elements of the given type
config.getArray<TYPE>("key")
// Returns a matrix: an array of arrays of elements of the given type
config.getMatrix<TYPE>("key")
// Returns an absolute (canonical if it should exist) path to a file
config.getPath("key", true /* check if path exists */)
// Return an array of absolute paths
config.getPathArray("key", false /* do not check if paths exists */)
// Returns the value as literal text including possible quotation marks
config.getText("key")
// Set the value of key to the default value if the key is not defined
config.setDefault("key", default_value)
// Set the value of the key to the default array if key is not defined
config.setDefaultArray<TYPE>("key", vector_of_default_values)
// Create an alias named new_key for the already existing old_key or
 ↪  throws an exception if the old_key does not exist
config.setAlias("new_key", "old_key")
```

Conversions to the requested type are using the `from_string` and `to_string` methods provided by the string utility library described in Section 4.8. These conversions largely follow standard parsing, with one important exception. If (and only if) the value is retrieved as a C/C++ string and the string is fully enclosed by a pair of " characters, these are stripped before returning the value. Strings can thus also be provided with or without quotation marks.

Warning: It should be noted that a conversion from string to the requested type is a comparatively heavy operation. For performance-critical sections of the code, one should consider fetching the configuration value once and caching it in a local variable.

4.4 Modules and the Module Manager

Allpix Squared is a modular framework and one of the core ideas is to partition functionality in independent modules which can be inserted or removed as required. These modules are located in the subdirectory `src/modules/` of the repository, with the name of the directory the unique name of the module. The suggested naming scheme is CamelCase, thus an example module name would be `GenericPropagation`. There are two different kind of modules which can be defined:

- **Unique**: Modules for which a single instance runs, irrespective of the number of detectors.
- **Detector**: Modules which are concerned with only a single detector at a time. These are then replicated for all required detectors.

The type of module determines the constructor used, the internal unique name and the supported configuration parameters. More details about the instantiation logic for the different types of modules are given in the next section.

4.4.1 Module instantiation

Modules are dynamically loaded and instantiated by the Module Manager. They are constructed, initialized, executed and finalized in the linear order in which they are defined in the configuration file; for this reason the configuration file should follow the order of the real process. For each section in the main configuration file (see Section 4.3 for more details), a corresponding library is searched for which contains the module (the exception being the global framework section). Module libraries are always named following the scheme `libAllpixModule<ModuleName>`, reflecting the `ModuleName` configured via CMake. The module search order is as follows:

1. Modules already loaded before from an earlier section header
2. All directories in the global configuration parameter `library_directories` in the provided order, if this parameter exists.

3. The internal library paths of the executable, that should automatically point to the libraries that are built and installed together with the executable. These library paths are stored in `RPATH` on Linux, see the next point for more information.

4. The other standard locations to search for libraries depending on the operating system. Details about the procedure Linux follows can be found in [24].

If the loading of the module library is successful, the module is checked to determine if it is a unique or detector module. As a single module may be called multiple times in the configuration, with overlapping requirements (such as a module which runs on all detectors of a given type, followed by the same module but with different parameters for one specific detector, also of this type) the Module Manager must establish which instantiations to keep and which to discard. The instantiation logic determines a unique name and priority, where a lower number indicates a higher priority, for every instantiation. The name and priority for the instantiation are determined differently for the two types of modules:

- **Unique**: Combination of the name of the module and the `input` and `output` parameter (both defaulting to an empty string). The priority is always zero.

- **Detector**: Combination of the name of the module, the `input` and `output` parameter (both defaulting to an empty string) and the name of detector this module is executed for. If the name of the detector is specified directly by the `name` parameter, the priority is *high*. If the detector is only matched by the `type` parameter, the priority is *medium*. If the `name` and `type` are both unspecified and the module is instantiated for all detectors, the priority is *low*.

In the end, only a single instance for every unique name is allowed. If there are multiple instantiations with the same unique name, the instantiation with the highest priority is kept. If multiple instantiations with the same unique name and the same priority exist, an exception is raised.

4.4.2 Multithreading: Parallel execution of events

The framework supports running several events in parallel via its multithreading feature. By default, this feature is disabled for new modules. If supported by all modules in the simulation, multithreading is enabled by default, but can be disabled by the user as described in Section 3.4. When enabled this feature can provide a significant speed improvement, depending on the simulation chain.

The framework allows to parallelize the execution of the same module for multiple events, if these would otherwise be executed directly after each other in a linear order. Thus, events are added to a work queue and then distributed to a set of worker threads as specified in the configuration or determined from system parameters.

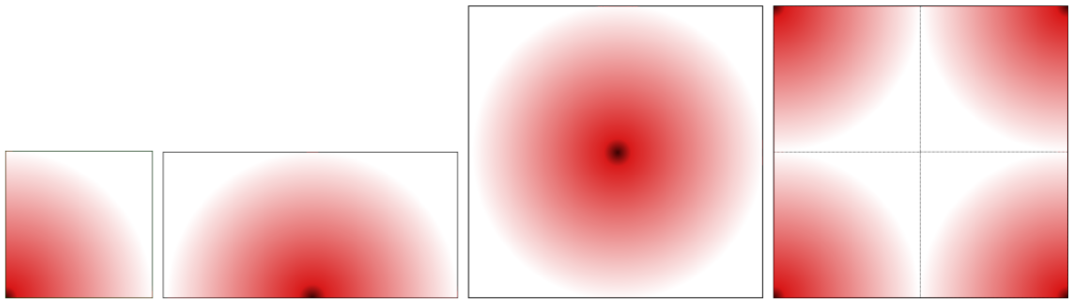
Detailed description of how the framework implements the multithreading feature can be found in Section 4.10 and an overview of important considerations when writing a new module capable of multithreading is provided in Section 10.4.

4.5 Field Maps

Allpix Squared allows to load externally generated field maps for various quantities such as the electric field or the doping profile of the sensor. These maps have to be provided as regularly spaced meshes in one of the supported field file formats. A conversion and interpolation tool to translate adaptive-mesh fields from TCAD applications to the format required by Allpix Squared is provided together with the framework and is described in Section 14.2.

This section of the manual provides an overview of the different field types and possibilities of mapping field of single pixels or fractions thereof to full sensor simulations in Allpix Squared.

4.5.1 Mapping of Fields to the Sensor Plane

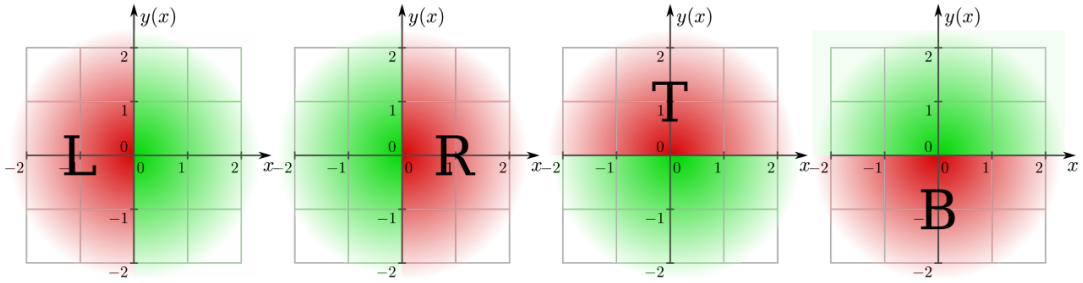


Examples for pixel geometries in field maps. The dark spot represents the pixel center, the red extend the electric field. Pixel boundaries are indicated with a dotted line where applicable.

Fields are always expected to be provided as rectangular maps, irrespective of the actual pixel shape. Maps are loaded once and assigned on a per-pixel basis. Depending on the symmetries of the pixel unit cell and the pixel grid, different geometries are supported as indicated in the figure above. The field for a quarter of the pixel plane, for half planes (see figures below) as well as for full planes (see figure above). The size of the field is not limited to a single pixel cell, however, for some quantities such as the electric field only the volume within the pixel cell is used and periodic boundary conditions are assumed and expected. Larger fields are for example useful for the weighting potential, where also potential differences to neighboring pixels are of interest.

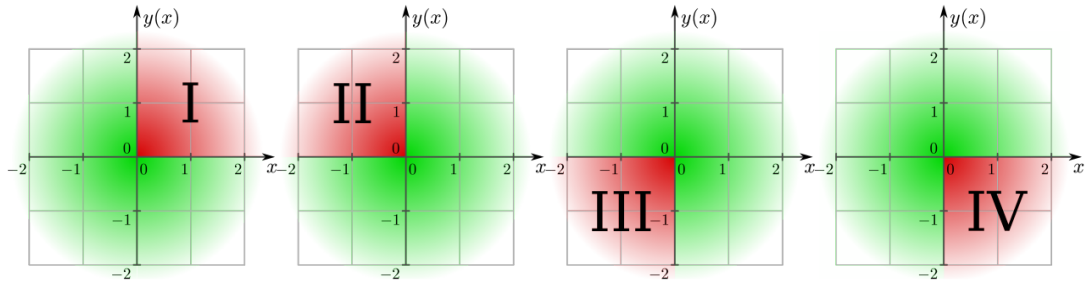
A special case is the field presented in the right panel of the figure above. Here, the field is not centered at the pixel unit cell center, but at the corner of four adjacent rectangular pixels.

Not all mapping geometries might be available for all types of fields used in Allpix Squared as will be detailed below.



(a) Left Half (b) Right Half (c) Top Half (d) Bottom Half

Location and orientation of the field map with respect to the pixel center when providing a half of the pixel plane. Here, $(0, 0)$ *denotes the pixel center, the red field portion is read from the field map and the green ones are replicated through mirroring.*



(a) Quadrant I (b) Quadrant II (c) Quadrant III (d) Quadrant IV

Location and orientation of the field map with respect to the pixel center when providing one quadrant in the pixel plane. Here, $(0, 0)$ *denotes the pixel center, the red field portion is read from the field map and the green ones are replicated through mirroring.*

The parameter `field_mapping` of the respective module defines how the map read from the mesh file should be interpreted and applied to the sensor, and the following possibilities are available:

- `SENSOR`: The map is read from the file and applied periodically to the full sensor, starting with the lower-left corner of the first pixel, i.e. at index 0,0. The field is then flipped at its edges to the right and upwards and the procedure is repeated until the other sensor edge is reached. This mode allows to apply fields that span several pixel to e.g. simulate even-odd differences in double columns, but only works well for regular, Cartesian pixel grids.

- `PIXEL_FULL`: The map is interpreted as field spanning the full Euclidean angle and aligned on the center of the pixel unit cell. No transformation is performed, but field values are obtained from the map with respect to the pixel center.

- `PIXEL_FULL_INVERSE`: The map is interpreted as full field, but with inverse alignment on the pixel corners as shown above. Consequently, the field value lookup from the four quadrants take into account their offset.

- `PIXEL_HALF_LEFT`: The map represents the left Euclidean half-plane, aligned at the y axis through the center of the pixel unit cell. Field values in the other half-plane are obtained by mirroring at the y axis as indicated in the figure above.

- `PIXEL_HALF_RIGHT`: The map represents the right Euclidean half-plane, aligned at the y axis through the center of the pixel unit cell. Field values in the other half-plane are obtained by mirroring at the y axis as indicated in the figure above.
- `PIXEL_HALF_TOP`: The map represents the top Euclidean half-plane, aligned at the x axis through the center of the pixel unit cell. Field values in the other half-plane are obtained by mirroring at the x axis as indicated in the figure above.
- `PIXEL_HALF_BOTTOM`: The map represents the bottom Euclidean half-plane, aligned at the x axis through the center of the pixel unit cell. Field values in the other half-plane are obtained by mirroring at the x axis as indicated in the figure above.
- `PIXEL_QUADRANT_I`: The map represents the quadrant of the plane where both vector components have a positive sign. Field values in the other three quadrants are obtained by mirroring at the axes of the coordinate system as shown in the figure above.
- `PIXEL_QUADRANT_II`: The map represents the quadrant of the plane where the vector component x has a negative and y a positive sign. Field values in the other three quadrants are obtained by mirroring at the axes of the coordinate system as shown in the figure above.
- `PIXEL_QUADRANT_III`: The map represents the quadrant of the plane where both vector components have a negative sign. Field values in the other three quadrants are obtained by mirroring at the axes of the coordinate system as shown in the figure above.
- `PIXEL_QUADRANT_IV`: The map represents the quadrant of the plane where the vector component x has a positive and y a negative sign. Field values in the other three quadrants are obtained by mirroring at the axes of the coordinate system as shown in the figure above.

All axes mentioned here are Cartesian axes aligning with the local coordinate system of the sensor, described in Section 5.2, and passing through the center of the pixel unit cell regarded. It should be noted that some of these mappings are equivalent to rotating or mirroring the field before loading it in Allpix Squared, and are only provided for convenience.

In addition to these mappings, the field maps can be shifted and stretched using the `field_offset` and `field_scale` parameters of the respective module. The values of these parameters are always interpreted as fractions of the field map size that has been loaded. This means for example, that an offset of `field_offset = 0.5, 0.5` applied to a field map with a size of `100um x 50um` will shift the respective field by `50um` along `x` and `25um` along `y`.

4.5.2 Weighting Potential Maps & Induction

Induced currents in Allpix Squared are calculated following the Shockley-Ramo theorem [25,26]. The induced current of a moving charge carrier requires the knowledge of the weighting potential in addition to the electric field of the sensor. The weighting potential for a given sensor geometry can be calculated analytically or by means of a finite-element

simulation by setting the electrode of the pixel under consideration to unit potential, and all other electrodes to ground [27].

The Shockley-Ramo theorem then states that the charge Q_n^{ind} induced by the motion of a charge carrier is equivalent to the difference in weighting potential between the previous location $\vec{x}_0$ and its current position $\vec{x}_1$, viz.

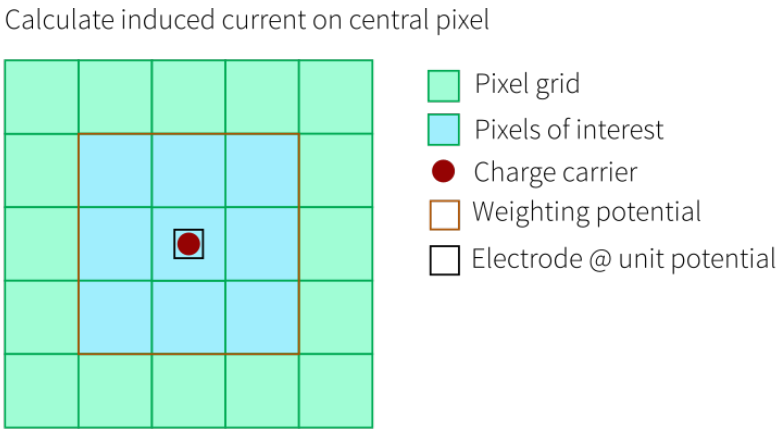
$$Q_n^{ind} = \int_{t_0}^{t_1} I_n^{ind} \mathrm{d}t = q\left[\phi(\vec{x}_1) - \phi(\vec{x}_0)\right],$$

assuming discrete time steps. Here, q is the charge of the carrier, $\phi(\vec{x})$ the weighting potential at position $\vec{x}$ and I_n^{ind} the induced current in the particular time step. A detailed description of the procedure is provided in [28] along with examples of application.

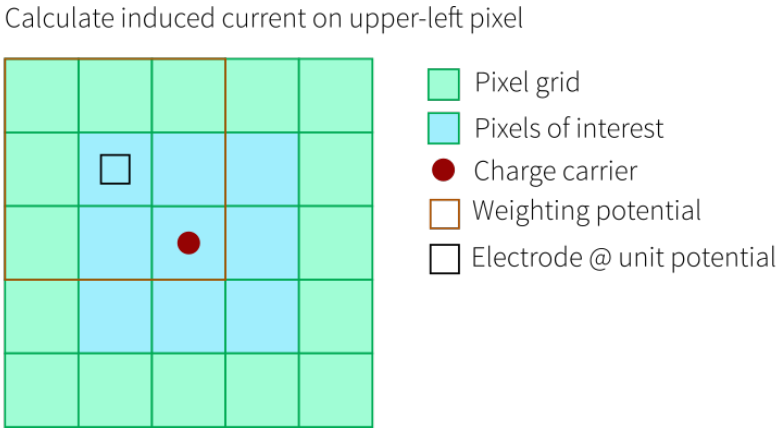
Since this procedure requires a realignment of the weighting potential for every pixel or electrode in question, the `SENSOR` mapping geometry is not a viable option. The weighting potential map needs to be centered around the electrode on unit potential.

The following drawings indicate how the induced current calculations are performed in Allpix Squared. Here, the pixels in the region of interest for which the induced current is calculated are shown in blue. The charge carrier position is indicated by the red dot and the weighting potential is displayed in orange, with its electrode at unit potential as small black square and its full extent indicated by the orange line.

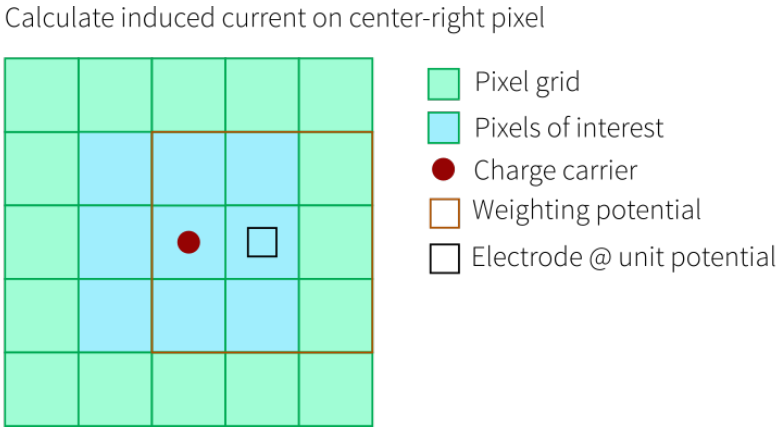
The weighting potential is centered with its readout electrode on unit potential on the pixel of interest for which the induced current by the charge carrier movement is to be calculated. For the subsequent pixel of interest, the position of the weighting potential is adjusted accordingly.



Calculation of the induced current in the pixel under which the charge carrier is moving. The weighting potential is therefore centered on this pixel. The weighting potential difference is calculated from the two carrier positions in the center of the 3x3 pixel map.

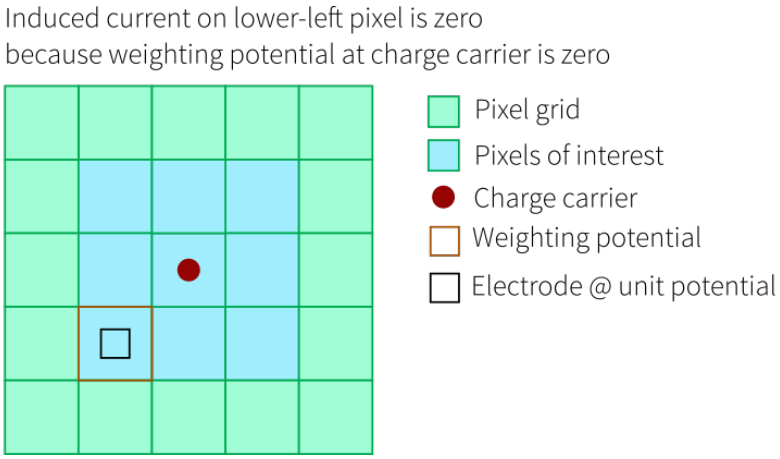


Calculation of the induced current in a pixel neighboring the one under which the charge carrier is moving. The weighting potential is shifted accordingly to be centered on the neighbor pixel in question. The weighting potential difference is calculated from the two carrier positions in the lower-right pixel of the 3x3 pixel map.



Calculation of the induced current in a pixel neighboring the one under which the charge carrier is moving. The weighting potential is shifted accordingly to be centered on the neighbor pixel in question. The weighting potential difference is calculated from the two carrier positions in the center-left pixel of the 3x3 pixel map.

For the special case of a strongly confined weighting potential at the collection electrode, it suffices to consider the potential of a single pixel cell. In this case, the induced current in all neighboring pixels is zero since they reside outside the defined weighting potential.



The induced current in the lower-left pixel neighboring the one under which the charge carrier moves is zero, since the weighting potential has a size of only 1x1 pixels and the potential at the position of the charge carrier with respect to the pixel in question os by definition zero.

4.6 Passing Objects using Messages

Communication between modules is performed by the exchange of messages. Messages are templated instantiations of the `Message` class carrying a vector of objects. The list of objects available in the Allpix Squared objects library is given in Section 7.1. The messaging system has a dispatching mechanism to send messages and a receiving part that fetches incoming messages. Messages are always received by modules in the order they have been dispatched by preceding modules.

The dispatching module can specify an optional name for the messages, but modules should normally not specify this name directly. If the name is not given (or equal to `-`) the `output` parameter of the module is used to determine the name of the message, defaulting to an empty string. Dispatching messages to their receivers is then performed following these rules:

1. The receiving module will *only* receive a message if it has the same type as the message dispatched (thus carrying the same objects). If the receiver is however listening to the `BaseMessage` type which does not specify the type of objects it is carrying, it will instead receive all dispatched messages.
2. The receiving module will *only* receive messages with the exact name it is listening for. The module uses the `input` parameter to determine which message names it should listen for; if the `input` parameter is equal to `*` the module will listen to all messages. Each module by default listens to messages with no name specified (thus receiving the messages of dispatching modules without output name specified).
3. If the receiving module is a detector module, it will *only* receive messages bound to that specific detector *or* messages that are not bound to any detector.

An example of how to dispatch a message containing an array of `Object` types bound to a detector named `dut` is provided below. As usual, the message is dispatched at the end of the `run()` function of the module.

```
void run(Event* event) {
    std::vector<Object> data;
    // ..fill the data vector with objects ...

    // The message is dispatched only for the module's detector, stored
 ↪  in "detector_"
    auto message = std::make_shared<Message<Object>>(data, detector_);

    // Send the message using the Messenger object for the given event
    messenger->dispatchMessage(this, message, event);
}
```

4.6.1 Methods to process messages

The message system has multiple methods to process received messages. The first two are the most common methods and the third should be avoided in almost every instance.

1. Bind a **single message** to the input of this module. This should usually be the preferred method, where a module expects only a single message to arrive per event containing the list of all relevant objects. The following example binds to a message containing an array of objects and is placed in the constructor of a detector-type `TestModule`:

   ```
   TestModule(Configuration&, Messenger* messenger,
    ↪  std::shared_ptr<Detector>) {
       // Subscribe to a single message, with no special messenger
    ↪  flags
       messenger->bindSingle<ExampleMessage>(this, MsgFlags::NONE);
   }
   ```

2. Bind a **set of messages** to the input of the module. This method should be used if the module can (and expects to) receive the same message multiple times (possibly because it wants to receive the same type of message for all detectors). An example to bind multiple messages containing an array of objects in the constructor of a unique-type `TestModule` would be:

   ```
   TestModule(Configuration&, Messenger* messenger, GeometryManager*
    ↪  geo_manager) {
       // Subscribe to multiple messages, with no special messenger
    ↪  flags
       messenger->bindMulti<Message<Object>>(this, MsgFlags::NONE);
   }
   ```

3. Listen to a particular message type and execute a **filter function** as soon as an object is received. This can be used for more advanced strategies of retrieving messages, but the other methods should be preferred whenever possible. The listening module should *not* do any heavy work in the filtering function as this is supposed to take place in the module `run` method instead. The filter function should return a boolean, indicating whether the message is wanted or not. Using a filter function can lead to unexpected behavior because the function is executed during the run method of the dispatching module. This means that logging is performed at the level of the dispatching module and that the filter method can be accessed from multiple threads if the dispatching module is parallelized. Listening to a message containing an array of objects in a detector-specific `TestModule` could be performed as follows:

   ```
   TestModule(Configuration&, Messenger* messenger,
    ↪  std::shared_ptr<Detector>) {
       messenger->registerFilter(this,
                                 /* Pointer to the filter method */
                                 &TestModule::filter,
                                 /* No special message flags */
                                 MsgFlags::NONE);
   ```

```
    }
    bool filter(std::shared_ptr<Message<Object>> message) const {
        // Decide if the message is wanted ...
    }
```

It should be noted that the `registerFilter` function by default adds the `IGNORE_NAME` message flag to receive all available messages if called without the message flag parameter:

```
messenger->registerFilter(this, &TestModule::filter);
```

This means that a possibly set `input` parameter of the respective module has no effect. If this behavior is undesired, the filter should be registered explicitly stating the desired message flags or `MsgFlags::NONE`. The available message flags are described in detail in the following section.

4.6.2 Message flags

Flags can be added to the bind and listening methods which enable a particular behavior of the framework.

- `REQUIRED`: Specifies that this message is required during the event processing. If this particular message is not received before it is time to execute the module's run function, the execution of the method is automatically skipped by the framework for the current event. This can be used to ignore modules which cannot perform any action without received messages, for example charge carrier propagation without any deposited charge carriers.
- `ALLOW_OVERWRITE`: By default an exception is automatically raised if a single bound message is overwritten (thus receiving it multiple times instead of once). This flag prevents this behavior. It can only be used for variables bound to a single message.
- `IGNORE_NAME`: If this flag is specified, the name of the dispatched message is not considered. Thus, the `input` parameter is ignored and forced to the value `*`.
- `UNNAMED_ONLY`: If this flag is specified, the module will only receive messages without explicit name. The `input` parameter is ignored and forced to the value `?` and all named messages are discarded. It should be noted that `IGNORE_NAME` takes precedence over this parameter.

4.6.3 Persistency

As objects may contain information relating to other objects, in particular for storing their corresponding Monte Carlo history (see Section 7.2), objects are by default persistent until the end of each event. All messages are stored as shared pointers and are released at the end of each event. If no other copies of the shared message pointer are created, then these will be subsequently deleted, including the objects stored therein. Where a module requires access to data from a previous event (such as to simulate the effects of pile-up etc.), local copies of the data objects must be created. Note that at the point of creating copies the corresponding history will be lost.

4.7 Redirect Module Inputs and Outputs

In the Allpix Squared framework, modules exchange messages typically based on their input and output message types and the detector type. It is, however, possible to specify a name for the incoming and outgoing messages for every module in the simulation. Modules will then only receive messages dispatched with the name provided and send named messages to other modules listening for messages with that specific name. This enables running the same module several times for the same detector, e.g. to test different parameter settings.

The message output name of a module can be changed by setting the `output` parameter of the module to a unique value. The output of this module is then not sent to modules without a configured input, because by default modules listens only to data without a name. The `input` parameter of a particular receiving module should therefore be set to match the value of the `output` parameter. In addition, it is permitted to set the `input` parameter to the special value `*` to indicate that the module should listen to all messages irrespective of their name.

An example of a configuration with two different settings for the digitization module is shown below:

```
# Digitize the propagated charges with low noise levels
[DefaultDigitizer]
# Specify an output identifier
output = "low_noise"
# Low amount of noise added by the electronics
electronics_noise = 100e
# Default values are used for the other parameters

# Digitize the propagated charges with high noise levels
[DefaultDigitizer]
# Specify an output identifier
output = "high_noise"
# High amount of noise added by the electronics
electronics_noise = 500e
# Default values are used for the other parameters

# Save histogram for 'low_noise' digitized charges
[DetectorHistogrammer]
# Specify input identifier
input = "low_noise"

# Save histogram for 'high_noise' digitized charges
[DetectorHistogrammer]
# Specify input identifier
input = "high_noise"
```

4.8 Logging and other Utilities

The Allpix Squared framework provides a set of utilities which improve the usability of the framework and extend the functionality provided by the C++ Standard Template Library (STL). The former includes a flexible and easy-to-use logging system and an easy-to-use framework for units that supports converting arbitrary combinations of units to common base units which can be used transparently throughout the framework. The latter comprise tools which provide functionality the C++17 standard does not contain. These utilities are used internally in the framework and are only shortly discussed.

Logging system

The logging system is built to handle input/output in the same way as `std::cin` and `std::cout` do. This approach is both flexible and easy to read. The system is globally configured, thus only one logger instance exists. The following commands are available for sending messages to the logging system at a level of `LEVEL`:

- `LOG(LEVEL)`: Sends a message with severity level `LOG(LEVEL)` to the logging system. Example:

  ```
  LOG(LEVEL) << "this is an example message with an integer and a
  ↪  double " << 1 << 2.0;
  ```

  A new line and carriage return is added at the end of every log message. Multi-line log messages can be used by adding new line commands to the stream. The logging system will automatically align every new line under the previous message and will leave the header space empty on new lines.

- `LOG_ONCE(LEVEL)`: Same as `LOG()`, but will only log this message once over the full run, even if the logging function is called multiple times. Example:

  ```
  LOG_ONCE(INFO) << "This message will appear once only, even if
  ↪  present in every event...";
  ```

  This can be used to log warnings or messages e.g. from the `run()` function of a module without flooding the log output with the same message for every event. The message is preceded by the information that further messages will be suppressed.

- `LOG_N(LEVEL, NUMBER)`: Same as `LOG_ONCE()` but allows to specify the number of times the message will be logged via the additional parameter `NUMBER`. Example:

  ```
  LOG_N(INFO, 10) << "This message will appear maximally 10 times
  ↪  throughout the run.";
  ```

  The last message is preceded by the information that further messages will be suppressed.

- `LOG_PROGRESS(LEVEL, IDENTIFIER)`: This function allows to update the message to be updated on the same line for simple progressbar-like functionality. Example:

  ```
  LOG_PROGRESS(STATUS, "EVENT_LOOP") << "Running event " << n << " of
  ↪  " << number_of_events;
  ```

Here, the `IDENTIFIER` is a unique string identifying this output stream in order not to mix different progress reports.

If the output is a terminal screen the logging output will be coloured to make it easier to identify warnings and error messages. This is disabled automatically for all non-terminal outputs.

More details about the logging levels and formats can be found in Section 3.8.

Unit system

Correctly handling units and conversions is of paramount importance. Having a separate C++ type for every unit would however be too cumbersome for a lot of operations, therefore units are stored in standard C++ floating point types in a default unit which all code in the framework should use for calculations. In configuration files, as well as for logging, it is however useful to provide quantities in different units.

The unit system allows adding, retrieving, converting and displaying units. It is a global system transparently used throughout the framework. Examples of using the unit system are given below:

```
// Define the standard length unit and an auxiliary unit
Units::add("mm", 1);
Units::add("m", 1e3);
// Define the standard time unit
Units::add("ns", 1);
// Get the units given in m/ns in the defined framework unit (mm/ns)
Units::get(1, "m/ns");
// Get the framework unit (mm/ns) in m/ns
Units::convert(1, "m/ns");
// Return the unit in the best type (lowest number larger than one) as
 ↪  string.
// The input is in default units 2000mm/ns and the 'best' output is
 ↪  2m/ns (string)
Units::display(2e3, {"mm/ns", "m/ns"});
```

A description of the use of units in config files within Allpix Squared was presented in Section 3.1.

Internal utilities

STL only provides string conversions for standard types using `std::stringstream` and `std::to_string`, which do not allow parsing strings encapsulated in pairs of double quote (") characters nor integrating different units. Furthermore it does not provide wide flexibility to add custom conversions for other external types in either way.

The framework's `to_string` and `from_string` methods provided by its **string utilities** do allow for these flexible conversions, and are extensively used in the configuration system. Conversions of numeric types with a unit attached are automatically resolved

using the unit system discussed above. The string utilities also include trim and split strings functions missing in the STL.

Furthermore, the Allpix Squared tool system contains extensions to allow automatic conversions for ROOT and Geant4 types as explained in Section 14.1.

To be able to provide cross-platform reproducibility of simulations, Allpix Squared uses random number distributions from the Boost.Random library. Their implementation is fixed and therefore does not depend on the standard library of the respective platform. For ease of use, the most common ones are exported to the `allpix::` namespace.

The pseudo-random number generator used for event seeds and random number generation within modules is the `std::mt19937_64`, a 64-bit Mersenne Twister algorithm. In order to allow for debugging of the random number distribution in a multithreaded environment, Allpix Squared provides the `allpix::RandomNumberGenerator` wrapper around the STL object, which allows to print every random number drawn from the generator to the logging facilities when setting the log level to `PRNG`.

4.9 Error Reporting and Exceptions

Allpix Squared generally follows the principle of throwing exceptions in all cases where something is definitely wrong. Exceptions are also thrown to signal errors in the user configuration. It does not attempt to circumvent problems or correct configuration mistakes, and the use of error return codes is to be discouraged. The asset of this method is that errors cannot easily be ignored and the code is more predictable in general.

For warnings and information messages, the logging system should be used extensively. This helps both in following the progress of the simulation and in debugging problems. Care should however be taken to limit the amount of messages in levels higher than `DEBUG` or `TRACE`. More details about the logging levels and their usage can be found in Section 3.8.

The base exceptions in Allpix Squared are available via the utilities. The most important exception base classes are the following:

- `ConfigurationError`: All errors related to incorrect user configuration. This could indicate a non-existing configuration file, a missing key or an invalid parameter value.
- `RuntimeError`: All other errors arising at run-time. Could be related to incorrect configuration if messages are not correctly passed or non-existing detectors are specified. Could also be raised if errors arise while loading a library or executing a module.
- `LogicError`: Problems related to modules which do not properly follow the specifications, for example if a detector module fails to pass the detector to the constructor. These methods should never be raised for correctly implemented modules and should therefore not be of any concern for the end users. Reporting this type of error can help developers during the development of new modules.

There are only four exceptions that are supposed to be used in specific modules, outside of the core framework. These exceptions should be used to indicate errors that modules cannot handle themselves:

- `InvalidValueError`: Derived from configuration exceptions. Signals any problem with the value of a configuration parameter not related to parsing or conversion to the required type. Can for example be used for parameters where the possible valid values are limited, like the set of logging levels, or for paths that do not exist. An example is shown below:

```
void run(Event* event) {
    // Fetch a key from the configuration
    std::string value = config.get("key");

    // Check if it is a 'valid' value
    if(value != 'A' && value != "B") {
        // Raise an error if it the value is not valid
        //   provide the configuration object, key and an
 ↪  explanation
        throw InvalidValueError(config, "key", "A and B are the only
 ↪  allowed values");
    }
}
```

- `InvalidCombinationError`: Derived from configuration exceptions. Signals any problem with a combination of configuration parameters used. This could be used if several optional but mutually exclusive parameters are present in a module, and it should be ensured that only one is specified at the time. The exceptions accepts the list of keys as initializer list. An example is shown below:

```
void run(Event* event) {
    // Check if we have mutually exclusive options defined:
    if(config.count({"exclusive_opt_a", "exclusive_opt_b"}) > 1) {
        // Raise an error if the combination of keys is not valid
        //   provide the configuration object, keys and an
 ↪  explanation
        throw InvalidCombinationError(config, {"exclusive_opt_a",
 ↪  "exclusive_opt_b"},
                "Options A and B are mutually exclusive, specify only
 ↪  one.");
    }
}
```

- `ModuleError`: Derived from module exceptions. Should be used to indicate any runtime error in a module not directly caused by an invalid configuration value, for example that it is not possible to write an output file. A reason should be given to indicate what the source of problem is.

- `EndOfRunException`: Derived from module exceptions. Should be used to request the end of event processing in the current run, e.g. if a module reading in data from a file reached the end of its input data.

4.10 Multithreading

Allpix Squared supports multithreading by running events in parallel. The module manager creates a thread pool with the configured number of workers or determines them from system parameters if not specified. Each event is represented by an instance of the `Event` class which encapsulates the data used during this event. The configured number of events are then submitted to the thread pool and executed by the thread pool's workers.

The thread pool features two independent queues. A FIFO-like unsorted queue for events to be processed, and a second, priority-ordered queue for buffered events. The former is constantly filled with new events to be processed by the main thread, while the latter is used to temporarily buffer events which wait to be picked up in the correct sequence by a `SequentialModule`.

By default modules are assumed to not operate in a thread-safe way and therefore cannot participate in multithreaded processing of events. Therefore each module must explicitly enable multithreading in its constructor in order to signal its multithreading capabilities to Allpix Squared. To support multithreading, the module `run()` method should be re-entrant and any shared member variables should be protected. If multithreading is enabled in the run configuration, the module manager will check if all the loaded modules support multithreading. In case one or more modules do not support multithreading, a warning is printed and the feature is disabled. Modules can inform themselves about the decision via the `multithreadingEnabled()` method.

Seed Distribution

A stable seed distribution to modules and core components of Allpix Squared is guaranteed in order to be able to provide reproducibility of simulation results from the same inputs even when the number of workers is different. Each event is seeded upon its creation by the main thread from a central event seed generator, in increasing sequence of event numbers. The event provides access to a random engine that can be used by each module in the `run()` method.

To avoid the memory overhead of maintaining random engine objects equal to the number of events, the storage of the engines is made static and thread-local, and is only provided to the event for temporary usage. This way ensures that the framework maintains the minimum number of such heavy objects equal to the number of workers used. When a worker starts to execute a new event, it seeds its local random engine first and passes it to the event object.

Using Messenger in Parallel

The `Messenger` handles communication in different events concurrently. It supports dispatching and fetching messages via the `LocalMessenger`. Each event has its own local messenger which stores all messages that was produced in this event. The `Messenger` owns the global message subscription information and internally forwards the module's requests to dispatch or fetch messages to the local messenger of the event in a thread-safe manner.

Running Events in order using SequentialModule

The `SequentialModule` class is made available for modules that require processing of events in the correct order without disabling multithreading. Inheriting from this class will allow the module to transparently check if the given event is in the correct sequence and decide whether to execute it immediately or to request buffering in the prioritized buffer queue if the thread pool if it is out of order.

Using the `SequentialModule` is suitable for I/O modules which read or write to the file system and do not allow random read or write access to events. This enables output modules to produce the same output file for the same simulation inputs without sacrificing the benefits of using multithreading for other modules.

Since random number generators are thread-local and shared between events processed on the same thread, their state is stored internally when being written into the buffer and restored before processing. This ensures that the sequence of pseudo-random numbers is exactly the same regardless of whether the event was buffered or directly processed.

Geant4 Modules

The usage of the Geant4 library in Allpix Squared has some constraints because the Geant4 multithreaded run manager expects to handle parallelization internally which violates the Allpix Squared design. Furthermore, Geant4 does not guarantee results reproducibility between its multithreaded and sequential run managers. Modules that would like to use the Geant4 library shall not use the run managers provided by Geant4. Instead, they must use the custom run managers provided by Allpix Squared as described in Section 14.1.

Object History, TRefs and PointerWrappers

Allpix Squared uses ROOTs `TRef` objects to store the persistent links of the simulation object history. These references act similar to C pointers and allow accessing the referenced object directly without additional bookkeeping or lookup of indices. Furthermore they persist when being written to a ROOT file and read back to memory. ROOT implements this via a central lookup table that keeps track of the referenced objects and their location in memory as described in the ROOT documentation.

This approach comes with some drawbacks, especially in multithreaded environments. Most importantly the lookup table is a global object, which means mutexes are required for accessing it. Multiple threads generating or using `TRef` references will have to share this mutex and will consequently be subject to significant waiting for lock release. Furthermore generating more and more `TRef` relations over the course of a simulation will increase the size of the central reference table. This table is initialized with a fixed size, and once the number of `TRef` objects outgrows this pre-allocated space, new memory has to be acquired, leading to a reallocation of memory for the entire new size of the table. With potentially millions of entries, this quickly becomes a computationally expensive operation, slowing down the simulation significantly.

Allpix Squared solves these limitations by wrapping the `TRef` objects into a class called `PointerWrapper`. It contains both a direct, but transitional C pointer and a `TRef` to the

referenced object. The latter, however, is only generated when required, i.e. if the object holding the `PointerWrapper` as well as referenced object are going to be written to file. This is achieved by first going through all relevant objects, marking them for storage:

```
for(auto& object : objects) {
    object.markForStorage();
}
```

Now, the required history references can be identified and `TRef` objects are generated *only* for relations between two objects that are both marked for storage:

```
for(auto& object : objects) {
    object.petrifyHistory();
}
```

Objects can now be written to file and will contain the persistent reference to the related object.

This approach solves the above problems. File writing has to be performed single-threaded anyway, so generating `TRef` objects on the same thread does not lead to additional locking of the central reference table mutex in root. In addition, `TRef` entries are only generated and stored in the table for objects that require it - all references to objects not to be stored will be `nullptr` in either case since the target object is not available anymore when reading in the data. Since now the generation of `TRef` objects and access to the reference table is performed by a single thread and one single event at a time, it is also possible to reset the ROOT-internal object ID of `TRef` references after the event has been processed. The subsequent event will reuse the same IDs again, preventing a continuous growth of the reference table and related memory re-allocation issues.

As a consequence, when reading objects back from file in a mutlithreaded environment, the `TRef` has to be converted back to a C memory pointer in the reading thread, both to prevent mixing of re-used `TRef` object IDs from different events and to avoid locking access to the central reference table when looking up the memory location from there. This is performed similarly to the generation of history relations, and here only relations to valid TRefs are loaded, other relations will hold a `nullptr`:

```
for(auto& object : objects) {
    object.loadHistory();
}
```

For single-threaded applications such as ROOT analysis macros, this step is not necessary and the reference will be lazy-loaded when accessed, i.e. the `TRef` reference will be converted to a direct raw pointer only when actually used. Since events are processed sequentially and memory is freed between events, no mixing of IDs occurs.

5 Geometry and Detectors

This chapter introduces the coordinate systems used throughout the Allpix Squared framework and details the different implemented detector models as well as additional features such as support layers.

5.1 Simulation Geometry

Simulations are frequently performed for a set of different detectors (such as a beam telescope and a device under test). All of these individual detectors together form what Allpix Squared defines as the geometry. Each detector has a set of properties attached to it:

- A unique detector `name` to refer to the detector in the configuration.
- The `position` in the world frame. This is the position of the geometric center of the sensitive device (sensor) given in world coordinates as X, Y and Z s defined in Section 5.1.1 (note that any additional components like the chip and possible support layers are ignored when determining the geometric center).
- An `orientation_mode` that determines the way that the orientation is applied. This can be either `xyz`, `zyx` or `zxz`, **where xyz is used as default if the parameter is not specified**. Three angles are expected as input, which should always be provided in the order in which they are applied.
  - The `xyz` option uses extrinsic Euler angles to apply a rotation around the global X axis, followed by a rotation around the global Y axis and finally a rotation around the global Z axis.
  - The `zyx` option uses the **extrinsic Z-Y-X convention** for Euler angles, also known as Pitch-Roll-Yaw or 321 convention. The rotation is represented by three angles describing first a rotation of an angle ϕ (yaw) about the Z axis, followed by a rotation of an angle θ (pitch) about the initial Y axis, followed by a third rotation of an angle ψ (roll) about the initial X axis.
  - The `zxz` uses the **extrinsic Z-X-Z convention** for Euler angles instead. This option is also known as the 3-1-3 or the "x-convention" and the most widely used definition of Euler angles [29].

Note: It is highly recommended to always explicitly state the orientation mode instead of relying on the default configuration.

- The `orientation` to specify the Euler angles in logical order (e.g. first X, then Y, then Z for the `xyz` method), interpreted using the method above (or with the `xyz` method if the `orientation_mode` is not specified). An example for three Euler angles would be

  ```
  orientation_mode = "zyx"
  orientation = 45deg 10deg 12deg
  ```

  which describes the rotation of 45° around the Z axis, followed by a 10° rotation around the initial Y axis, and finally] a rotation of 12° around the initial X axis.

  > **Note**: All supported rotations are extrinsic active rotations, i.e. the vector itself is rotated, not the coordinate system. All angles in configuration files should be specified in the order they will be applied.

- A `type` parameter describing the detector model, for example `timepix` or `mimosa26` . These models define the geometry and parameters of the detector. Multiple detectors can share the same model, several of which are shipped ready-to-use with the framework.
- An `alignment_precision_position` optional parameter to specify the alignment precision along the three global axes as described in Section 3.3.
- An optional parameter `alignment_precision_orientation` for the alignment precision in the three rotation angles as described in Section 3.3.
- An optional **electric or magnetic field** in the sensitive device. These fields can be added to a detector by special modules as demonstrated in Section 3.7.

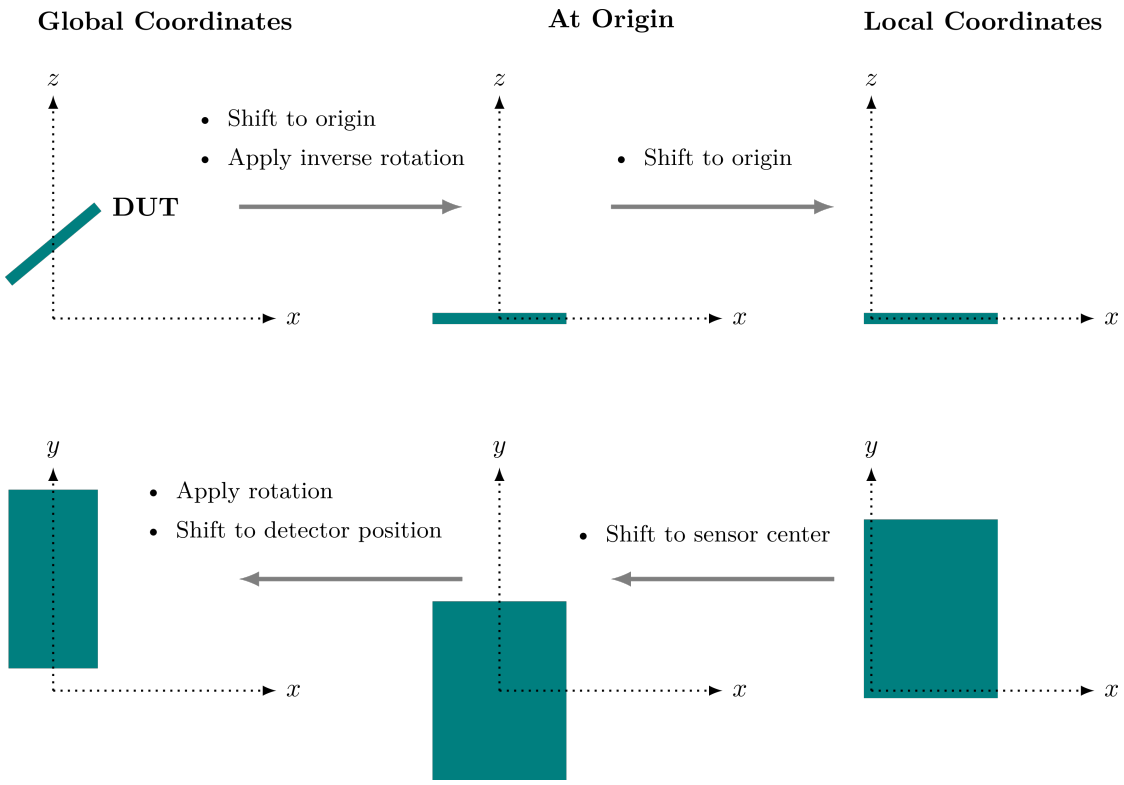
The detector configuration is provided in the detector configuration file as explained in Section 3.3.

5.1.1 Coordinate systems

Local coordinate systems for each detector and a global frame ofreference for the full setup are defined. The global coordinate system is chosen as a right-handed Cartesian system, and the rotations of individual devices are performed around the geometric center of their sensor.

Local coordinate systems for the detectors are also right-handed Cartesian systems, with the x- and y-axes defining the sensor plane. The origin of this coordinate system is the center of the lower left pixel in the grid, i.e. the pixel with indices (0,0). This simplifies calculations in the local coordinate system as all positions can either be stated in absolute numbers or in fractions of the pixel pitch.

A sketch of the actual coordinate transformations performed, including the order of transformations, is given below. The global coordinate system used for tracking of particles through detetector setup is shown on the left side, while the local coordinate system used to describe the individual sensors is located at the right.



Coordinate transformations from global to local and revers. The first row shows the detector positions in the respective coordinate systems in top view, the second row in side view.

The global reference for time measurements is the beginning of the event, i.e. the start of the particle tracking through the setup. The local time reference is the time of entry of the *first* primary particle of the event into the sensor. This means that secondary particles created within the sensor inherit the local time reference from their parent particles in order to have a uniform time reference in the sensor. It should be noted that Monte Carlo particles that start the local time frame on different detectors do not necessarily have to belong to the same particle track.

5.1.2 Changing and accessing the geometry

The geometry is needed at an early stage because it determines the number of detector module instantiations as explained in Section 4.4. The procedure of finding and loading the appropriate detector models is explained in more detail in the next section.

The geometry is directly added from the detector configuration file described in Section 3.3. The geometry manager parses this file on construction, and the detector models are loaded and linked later during geometry closing as described above. It is also possible to add additional models and detectors directly using `addModel` and `addDetector` (before the geometry is closed). Furthermore it is possible to add additional points which should be part of the world geometry using `addPoint`. This can for example be used to add the beam source to the world geometry.

The detectors and models can be accessed by name and type through the geometry manager using `getDetector` and `getModel`, respectively. All detectors can be fetched at once using the `getDetectors` method. If the module is a detector-specific module its

related detector can be accessed through the `getDetector` method of the module base class instead (returns a null pointer for unique modules) as follows:

```
void run(Event* event) {
    // Returns the linked detector
    std::shared_ptr<Detector> detector = this->getDetector();
}
```

5.2 Detector Models

Different types of detector models are available and distributed together with the framework: these models use the configuration format introduced in Section 4.3 and can be found in the `models` directory of the repository. Every model extends from the `DetectorModel` base class, which defines the minimum required parameters of a detector model within the framework. The coordinates place the detector in the global coordinate system, with the reference point taken as the geometric center of the active matrix. This is defined by the number of pixels in the sensor in both the x- and y-direction, and together with the pitch of the individual pixels the total size of the pixel matrix is determined. Outside the active matrix, the sensor can feature excess material in all directions in the x-y-plane. A detector of base class type does not feature a separate readout chip, thus only the thickness of an additional, inactive silicon layer can be specified. Derived models allow for separate readout chips, optionally connected with bump bonds.

The base detector model can be extended to provide different sensor geometries, and new assembly types can be added for more complex detector assembly setups. Currently, two assembly types are implemented, `MonolithicAssembly`, which describes a monolithic detector with all electronics directly implemented in the same wafer as the sensor, and the `HybridAssembly`, which in addition to the features described above also includes a separate readout chip with configurable size and bump bonds between the sensor and readout chip.

5.2.1 Detector model parameters

Models are defined in configuration files which are used to instantiate the actual model classes; these files contain various types of parameters, some of which are required for all models while others are optional or only supported by certain model types. For more details on how to add and use a new detector model, Section 10.5 should be consulted.

The set of base parameters supported by every model is provided below. These parameters should be given at the top of the file before the start of any sub-sections.

- `geometry`: A required parameter describing the geometry of the model. At the moment either `pixel` or `radial_strip`. This value determines some of the supported parameters as discussed later.
- `type`: A required parameter describing the type of the detector assembly. At the moment either `monolithic` or `hybrid`. This value determines some of the supported parameters as discussed later.

- number_of_pixels: The number of pixels in the 2D pixel matrix. Determines the base size of the sensor together with the pixel_size parameter below.
- pixel_size: The pitch of a single pixel in the pixel matrix. Provided as 2D parameter in the x-y-plane. This parameter is required for all models.
- sensor_material: Semiconductor material of the sensor. This can be any of the sensor materials supported by Allpix Squared, currently SILICON, GALLIUM_ARSENIDE, GERMANIUM, CADMIUM_TELLURIDE, CADMIUM_ZINC_TELLURIDE, DIAMOND and SILICON_CARBIDE. Defaults to SILICON if not specified.
- sensor_thickness: Thickness of the active area of the detector model containing the individual pixels. This parameter is required for all models.
- sensor_excess_<direction>: With <direction> either top, bottom, right or left, where the top, bottom, right and left direction are the positive y-axis, the negative y-axis, the positive x-axis and the negative x-axis, respectively. Specifies the extra material added to the sensor outside the active pixel matrix in the given direction.
- sensor_excess: Fallback for the excess width of the sensor in all four directions (top, bottom, right and left). Used if the specialized parameters described below are not given. Defaults to zero, thus having a sensor size equal to the number of pixels times the pixel pitch.
- chip_thickness: Thickness of the readout chip, placed next to the sensor.

The base parameters described above are the only set of parameters supported by the **monolithic** assembly. For this assembly, the chip_thickness parameter represents the first few micrometers of sensor material which contain the chip circuitry and are shielded from the bias voltage and thus do not contribute to the signal formation.

The **hybrid** assembly adds bump bonds between the chip and sensor while automatically making sure the chip and support layers are shifted appropriately. Furthermore, it allows the user to specify the chip dimensions independently from the sensor size, as the readout chip is treated as a separate entity. The additional parameters for the **hybrid** assembly are the following:

- chip_excess_<direction>: With direction either top, bottom, right or left. The chip excess in the specific direction, similar to the sensor_excess_<direction> parameter described above.
- chip_excess: Fallback for the excess width of the chip, defaults to zero and thus to a chip size equal to the dimensions of the pixel matrix. See the sensor_excess parameter above.
- bump_height: Height of the bump bonds (the separation distance between the chip and the sensor).
- bump_sphere_radius: The individual bump bonds are simulated as union solids of a sphere and a cylinder. This parameter sets the radius of the sphere to use.
- bump_cylinder_radius: The radius of the cylinder part of the bump. The height of the cylinder is determined by bump_height the parameter.

- `bump_offset`: A 2D offset of the grid of bumps. The individual bumps are by default positioned at the center of each single pixel in the grid.

5.2.2 Specializing detector models

A detector model contains default values for all parameters. Some parameters like the sensor thickness can however vary between different detectors of the same model. To allow for easy adjustment of these parameters, models can be specialized in the detector configuration file introduced in Section 3.3. All model parameters, except the type parameter and the support layers, can be changed by adding a parameter with the same key and the updated value to the detector configuration. The framework will then automatically create a copy of this model with the requested change.

Note: Before re-implementing models, it should be checked if the desired change can be achieved using the detector model specialization. For most cases this provides a quick and flexible way to adapt detectors to different needs and setups (for example, detectors with different sensor thicknesses).

5.2.3 Search order for models

To support different detector models and storage locations, the framework searches different paths for model files in the following order:

1. If defined, the paths provided in the global `model_paths` parameter are searched first. Files are read and parsed directly. If the path is a directory, all files in the directory are added (without recursing into subdirectories).
2. The location where the models are installed to (refer to the description of the `MODEL_DIRECTORY` variable in Section 2.5).
3. The standard data paths on the system as given by the environmental variable `XDG_ DATA_DIRS` with `Allpix/models` appended. The variable defaults to `/usr/local/share/` (thus effectively `/usr/local/share/Allpix/models`) followed by `/usr/share/` (effectively `/usr/share/Allpix/models`).
4. The path of the main configuration file.

5.2.4 Implants

Multiple implants per pixel cell can be simulated in Allpix Squared. Here, implants are any volume in the sensor in which charge carriers do not propagated, such as collection diodes, ohmic volumes or columns, as well as trenches filled with different materials e.g. for alpha conversion.

When charge carriers reach an implant, their propagation is stopped. Depending on the type of implant, they might be either discarded by the transfer module for back-side

implants, or taken into account when forming the front-end electronics input signal for front-side implants.

Each implant should be defined in its own section headed with the name `[implant]`. By default, no implants are added. Implants allow for the following parameters:

- `type`: Type of the implant. This parameter can be set to either `frontside` for an implant from the sensor front side, collecting charge carriers, or to `backside` for an implant connected to the ohmic contact at the sensor back side.
- `shape`: Shape of the implant, supported shapes are `rectangle` and `ellipse`. Defaults to `rectangle`.
- `size`: The size of the implant as 3D vector with size in x and y as well as the implant depth. Depending on the implant shape, the x and y values are either interpreted as the side lengths of the rectangle or the major and minor axes of the ellipse.
- `orientation`: Rotation of the implant around its z axis. Defaults to 0 degrees, i.e. the implant axes are aligned with the local coordinate system of the sensor.
- `offset`: 2D values in the x-y plane, defining the offset of the implant from the center of the pixel cell. This parameter is optional and defaults to $0, 0$, i.e. a position at the pixel center be default.

5.2.5 Support Layers

In addition to the active layer, multiple layers of support material can be added to the detector description. It is possible to place support layers at arbitrary positions relative to the sensor, while the default position is behind the readout chip or inactive sensor layer. The defined support materials will always be positioned relative to the corresponding detector. The support material can be chosen either from a set of predefined materials, including PCB and Kapton, or any material available via the Geant4 material database.

Every support layer should be defined in its own section headed with the name `[support ]`. By default, no support layers are added. Support layers allow for the following parameters.

- `size`: Size of the support in 2D (the thickness is given separately below). This parameter is required for all support layers.
- `thickness`: Thickness of the support layers. This parameter is required for all support layers.
- `location`: Location of the support layer. Either `sensor` to attach it to the sensor (opposite to the readout chip/inactive sensor layer), `chip` to add the support layer behind the chip/inactive layer or `absolute` to specify the offset in the z-direction manually. Defaults to `chip` if not specified. If the parameter is equal to `sensor` or `chip`, the support layers are stacked in the respective direction when multiple layers of support are specified.

- `offset`: If the `location` parameter is equal to `sensor` or `chip`, an optional 2D offset can be specified using this parameter, the offset in the z-direction is then automatically determined. These support layers are by default centered around the middle of the pixel matrix (the rotation center of the model). If the `location` is set to `absolute`, the offset is a required parameter and should be provided as a 3D vector with respect to the center of the model (thus the center of the active sensor). Care should be taken to ensure that these support layers and the rest of the model do not overlap.

- `hole_size`: Adds an optional cut-out hole to the support with the 2D size provided. The hole always cuts through the full support thickness. No hole will be added if this parameter is not present.

- `hole_type`: Type of hole to be punched into the support layer. Currently supported are `rectangle` and `cylinder`. Defaults to `rectangle`.

- `hole_offset`: If present, the hole is by default placed at the center of the support layer. A 2D offset with respect to its default position can be specified using this parameter.

- `material`: Material of the support. Allpix Squared does not provide a set of materials to choose from; it is up to the modules using this parameter to implement the materials such that they can use it. Chapter 8 provides details about the materials supported by the geometry builder modules (for example in the `GeometryBuilderGeant4 module documentation`).

5.2.6 Accessing specific detector models within the framework

Some modules are written to act on only a particular type of detector model. In order to ensure that a specific detector model has been used, the model should be downcast: the downcast returns a null pointer if the class is not of the appropriate type. An example for fetching a `HybridPixelDetectorModel` would thus be:

```
// "detector" is a pointer to a Detector object
auto model = detector->getModel();
auto hybrid_model =
 ↪  std::dynamic_pointer_cast<HybridPixelDetectorModel>(model);
if(hybrid_model != nullptr) {
    // The model of this Detector is a HybridPixelDetectorModel
}
```

5.3 Sensor Geometries

Allpix Squared implements different sensor geometries such as rectangular, Cartesian pixel grids, hexagon patterns, or radial strip-like channels. This section details the different geometries and their respective coordinate system. Geometries are selected via the parameter `geometry` in the detector model file.

5.3.1 Rectangular Pixels on a Cartesian Grid

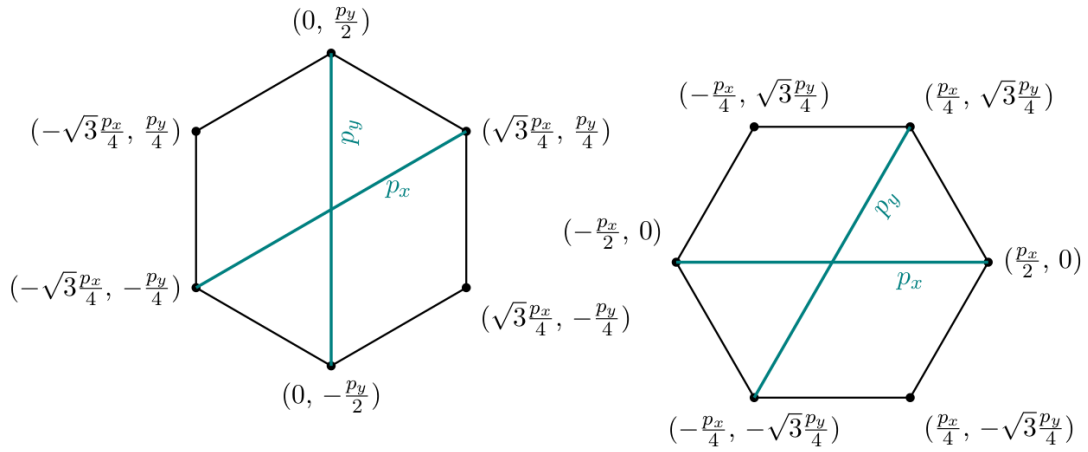
This geometry is the default assumed for any detector without the `geometry` keyword. The individual channels are rectangular pixels, the `pixel_size` parameter denotes the pitch in Cartesian `x` and `y` direction.

This geometry can be selected using `geometry = pixel`

5.3.2 Hexagonal Pixels

Hexagonal pixel grids in Allpix Squared use an axial coordinate system to describe the relative positions and indices of hexagons on the grid, following largely the definitions provided in [30]. Similar to the Cartesian coordinate system used for regular pixel layouts, the origin is the lower-left corner of the sensor, with the hexagon indices $(0, 0)$. Owing to the orientation of the grid axes, negative can occur in the top-left region of the sensor.

Two orientations of hexagons are supported, subsequently referred to as *pointy* with sides parallel to the y axis of the Cartesian coordinate system and corners at the top and bottom, and *flat* with sides parallel to the Cartesian x axis and corners to the left and right. The pitches p_x and p_y of the hexagon align with the axial coordinate system and are rotated differently with respect to the Cartesian system between the two variants. The orientation of the pitches as well as the resulting corner positions in Cartesian coordinates are shown in the figure below:



(a) Hexagon in *pointy* orientation

(b) Hexagon in *flat* orientation

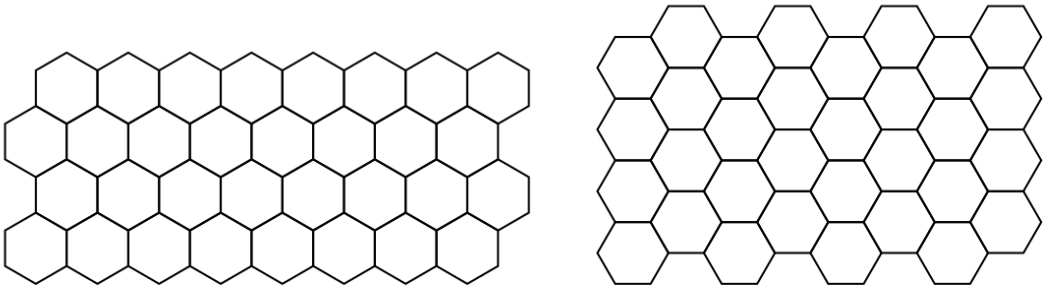
Definition of the pitches p_x and p_y, and corner positions for the pointy (left) and flat (right) hexagon orientation in Cartesian coordinates. The pitches align with the axes of the axial coordinate system of the hexagonal grid.

The additional parameters for the **hexagonal** model are as follows:

- `pixel_type`: The shape/orientation of the hexagonal pixels within the grid, either `hexagon_pointy` or `hexagon_flat`.

The number of pixels in a hexagonal grid are counted along the Cartesian axes, taking the offset pixels into account. For example, an 8-by-4 grid comprises 32 pixels both for

pointy and *flat* hexagon orientation, but results in different overall grid dimensions as demonstrated below:



(a) 8x4 grid with *pointy* hexagons (b) 8x4 grid with *flat* hexagons

Grid layouts for pointy (left) and flat (right) hexagons with a size of 8-by-4 pixels.

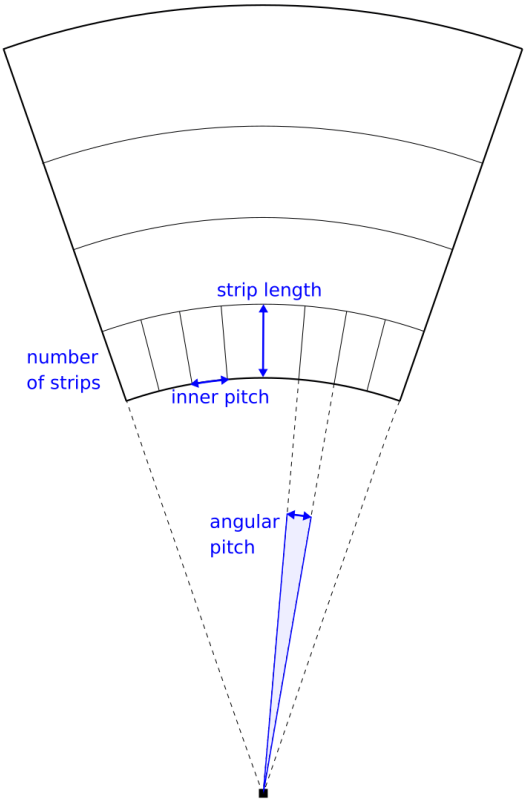
This geometry can be selected using `geometry = hexagon`

5.3.3 Radial Strips

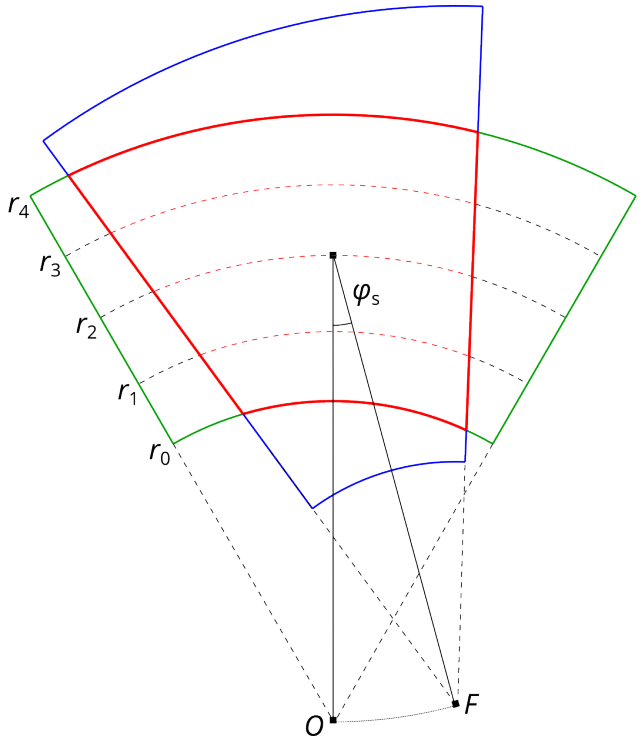
Radial strip detectors feature a trapezoidal shape with curved edges and radial geometry – the strips on such a sensor are arranged in a fan-like geometry, pointing to a common focal point. Shape, size and segmentation of a radial strip detector are defined using four parameters, each passed as an array with the number of elements equal to the number of strip rows: - `number_of_strips` - `angular_pitch` - `inner_pitch` - `strip_length`

Additionally, model parameters have to be set to `type = monolithic` and `geometry = radial_strip`. Due to the complexity of the geometry, this detector model currently doesn't allow the creation of passive support structures.

For radial strip detectors, the coordinate origin is placed in the center of concentric arcs, which form the strip row edges, to enable easier transformation to polar coordinates utilized by the detector model's member functions.



The optional parameter `stereo_angle` can be used to shift the strip focal point around the center of the sensor to create an asymmetrical sensor. By default, the stereo angle is disabled.



An examples of radial strip detector model implementation can be seen in `models/atlas_itk_r0` and further in the `examples/atlas_itk_petal` example.

6 Physics Models & Materials

Allpix Squared implements a variety of properties and models to describe the physics of semiconductor detectors. Models are implemented module-independently and can be selected via configuration parameters in the respective models, while sensor material properties serve as a default to module parameters and can be overwritten in the respective configuration section. This chapter serves as central reference for the different properties and models.

6.1 Sensor Material Properties

Allpix Squared supports the definition of a variety of semiconductor sensor materials. To simplify the setup of simulations with certain materials and to avoid inconsistent results, a set of default material properties is defined for each available material. These stored values serve as defaults to modules depending on one of these properties and may thus be overwritten using the corresponding configuration key in the respective section of the main configuration file.

The following parameters are currently provided by the framework:

- Charge creation energy
- Fano factor

The values for various materials are listed in the table below. It should be noted that for many of the following values a significant variation on measurements exist throughout literature, among others owed to a variation of material quality and composition and of vendors. The sources for the chosen default values are provided in the table.

Material	Energy $[eV]$	Fano factor	References
Silicon	3.64	0.115	[31, 32]
Germanium	2.97	0.112	[33]
Gallium Arsenide	4.2	0.14	[34]
Gallium Nitride	8.33	0.07	[35]
Cadmium Telluride	4.43	0.24	[36, 37]
Cadmium Zinc Telluride ($Cd_{0.8}Zn_{0.2}Te$)	4.6	0.14	[38, 39]
Diamond	13.1	0.382	[40]
Silicon Carbide (4H-SiC)	7.6	0.1	[41, 42]

It should be noted that material properties such as the density and composition of materials are defined only in case of constructing a Geant4 geometry via the

`GeometryBuilderGeant4` module, therefore these values are implemented within the respective module.

6.2 Charge Carrier Mobility

Allpix Squared provides different charge carrier mobility models, the best-suited model depends on the simulated device and other simulation parameters. Some models depend on the electric field strength to parametrize the mobility, others on the doping concentration of the device. The charge carrier mobility models are used by all propagation modules and comprise the following models:

6.2.1 Jacoboni-Canali Model

The Jacoboni-Canali model [43] is the most widely used parametrization of charge carrier mobility in Silicon as a function of the electric field E. It has originally been derived for $\langle 111 \rangle$ silicon lattice orientation, but is widely used also for the common $\langle 100 \rangle$ orientation. The mobility is parametrized as:

$$\mu(E) = \frac{v_m}{E_c} \frac{1}{\left(1 + (E/E_c)^\beta\right)^{1/\beta}}$$

where v_m, E_c, and β are phenomenological parameters, defined for electrons and holes respectively. The temperature dependence of these parameters is taken into account by scaling them with respect to a reference parameter value

$$A = A_{ref} \cdot T^\gamma$$

where A_{ref} is the reference parameter value, T the temperature in units of Kelvin, and γ the temperature scaling factor.

The parameter values implemented in Allpix Squared are taken from Table 5 of [43] as

$$\begin{aligned} v_{m,e} &= 1.53 \times 10^9 \,\mathrm{cm\,s^{-1}} \cdot T^{-0.87} \\ E_{c,e} &= 1.01 \,\mathrm{V\,cm^{-1}} \cdot T^{1.55} \\ \beta_e &= 2.57 \times 10^{-2} \cdot T^{0.66} \end{aligned}$$

$$\begin{aligned} v_{m,h} &= 1.62 \times 10^8 \,\mathrm{cm\,s^{-1}} \cdot T^{-0.52} \\ E_{c,h} &= 1.24 \,\mathrm{V\,cm^{-1}} \cdot T^{1.68} \\ \beta_h &= 0.46 \cdot T^{0.17} \end{aligned}$$

for electrons and holes, respectively.

This model can be selected in the configuration file via the parameter `mobility_model = "jacoboni"`.

6.2.2 Canali Model

The Canali model [44] differs from the Jacoboni-Canali model in the equation only by the value of v_m for electrons. The difference is most likely a typo in the Jacoboni reproduction of the parametrization, so this one can be considered the original parametrization derived from data. The altered value is taken from equation 2a in [44] and amounts to

$$v_{m,e} \;=\; 1.43 \times 10^9\,\mathrm{cm\,s^{-1}} \cdot T^{-0.87}.$$

A comparison with other models exhibits a better accordance of the electron mobility compared to the Jacoboni-Canali parameter value, especially at very high values of the electric field.

This model can be selected in the configuration file via the parameter `mobility_model = "canali"`.

6.2.3 Hamburg Model

The Hamburg model [45] presents an empirical parametrization of electron and hole mobility as a function of the electric field E based on measurements of drift velocities in high-ohmic silicon with $\langle 100 \rangle$ lattice orientation. The mobility is parametrized as

$$\mu_e^{-1}(E) = 1/\mu_{0,e} + E/v_{sat}$$

$$\begin{aligned} \mu_h^{-1}(E) &= 1/\mu_{0,h} && \text{for} \quad E < E_0 \\ &= 1/\mu_{0,h} + b \cdot (E - E_0) + c \cdot (E - E_0)^2 && \text{for} \quad E \geq E_0 \end{aligned}$$

as taken from equations 3 and 5 of [45].

The temperature dependence of the model parameters are calculated with respect to their reference values at a temperature of 300 Kelvin via equation 6 of [45] as:

$$A_i = A_i(T = 300\,\mathrm{K}) \cdot \left(\frac{T}{300\,\mathrm{K}}\right)^{\gamma_i}$$

The hole mobility parameter c is assumed to have no temperature dependence.

The parameter values implemented in Allpix Squared are taken from Table 4 of [45] as

$$\begin{aligned} \mu_{0,e} &= 1530\,\mathrm{cm^2\,V^{-1}\,s^{-1}} \cdot (T \,/\, 300\,\mathrm{K})^{-2.42} \\ v_{sat} &= 1.03 \times 10^7\,\mathrm{cm\,s^{-1}} \cdot (T \,/\, 300\,\mathrm{K})^{-0.226} \end{aligned}$$

$$\begin{aligned} \mu_{0,h} &= 464\,\mathrm{cm^2\,V^{-1}\,s^{-1}} \cdot (T \,/\, 300\,\mathrm{K})^{-2.20} \\ b &= 9.57 \times 10^{-8}\,\mathrm{cm\,s^{-1}} \cdot (T \,/\, 300\,\mathrm{K})^{-0.101} \\ c &= -3.31 \times 10^{-13}\,\mathrm{s\,V^{-1}} \\ E_0 &= 2640\,\mathrm{V\,cm^{-1}} \cdot (T \,/\, 300\,\mathrm{K})^{0.526} \end{aligned}$$

for electrons and holes, respectively.

This model can be selected in the configuration file via the parameter `mobility_model = "hamburg"`.

6.2.4 Hamburg High-Field Model

The Hamburg high-field model [45] takes the same form as the Hamburg model, but uses a different set of parameter values. The values are taken from Table 3 of [45] and are suitable for electric field strengths above $2.5\,\mathrm{kV\,cm^{-1}}$. Again, no temperature dependence is assumed on hole mobility parameter c, while all other parameters are scaled to temperatures different than 300 Kelvin using the equation from the Hamburg model.

The parameter values implemented in Allpix Squared are

$$\begin{aligned}
\mu_{0,e} &= 1430\,\mathrm{cm^2\,V^{-1}\,s^{-1}} \cdot (T \,/\, 300\,\mathrm{K})^{-1.99} \\
v_{sat} &= 1.05 \times 10^7\,\mathrm{cm\,s^{-1}} \cdot (T \,/\, 300\,\mathrm{K})^{-0.302}
\end{aligned}$$

$$\begin{aligned}
\mu_{0,h} &= 457\,\mathrm{cm^2\,V^{-1}\,s^{-1}} \cdot (T \,/\, 300\,\mathrm{K})^{-2.80} \\
b &= 9.57 \times 10^{-8}\,\mathrm{cm\,s^{-1}} \cdot (T \,/\, 300\,\mathrm{K})^{-0.155} \\
c &= -3.24 \times 10^{-13}\,\mathrm{s\,V^{-1}} \\
E_0 &= 2970\,\mathrm{V\,cm^{-1}} \cdot (T \,/\, 300\,\mathrm{K})^{0.563}
\end{aligned}$$

for electrons and holes, respectively.

This model can be selected in the configuration file via the parameter `mobility_model = "hamburg_highfield"`.

6.2.5 Masetti Model

The Masetti mobility model [46] parametrizes electron and hole mobility as a function of the total doping concentration D of the silicon material. This model requires a doping profile to be loaded for the detector in question, and an error will be returned if the doping profile is missing.

While this mobility model requires the *total doping concentration* $N_D + N_A$ as parameter, the doping profile used throughout Allpix Squared provides the *effective doping concentration* $N_D - N_A$ since this also encodes the majority charge carriers via its sign. However, in the parts of a silicon detector relevant for this simulation, i.e. the sensing volume, the difference between effective and total concentration is expected to be negligible. Therefore the doping concentration in this model is taken as the absolute value $N = |N_D - N_A|$.

The mobility is parametrized as

$$\mu_e(N) = \mu_{0,e} + \frac{\mu_{max,e} - \mu_{0,e}}{1 + (N/C_{r,e})^{\alpha_e}} - \frac{\mu_{1,e}}{1 + (C_{s,e}/N)^{\beta_e}}$$
$$\mu_h(N) = \mu_{0,h} \cdot e^{-P_c/N} + \frac{\mu_{max,h}}{1 + (N/C_{r,h})^{\alpha_h}} - \frac{\mu_{1,h}}{1 + (C_{s,h}/N)^{\beta_h}}$$

as taken from equations 1 (for electrons) and 4 (for holes) of [46].

Only the parameters μ_{max} for both electrons and holes are temperature dependent and are scaled according to the equation from the Hamburg model with parameters $\gamma_e = -2.5$ for electrons and $\gamma_e = -2.2$ for holes.

The parameter values implemented in Allpix Squared are taken from Table I of [46] for phosphorus and boron as

$$\begin{aligned}
\mu_{0,e} &= 68.5\,\mathrm{cm^2\,V^{-1}\,s^{-1}} \\
\mu_{max,e} &= 1414\,\mathrm{cm^2\,V^{-1}\,s^{-1}} \cdot (T \,/\, 300\,\mathrm{K})^{-2.5} \\
C_{r,e} &= 9.20 \times 10^{16}\,\mathrm{cm^{-3}} \\
\alpha_e &= 0.711 \\
\mu_{1,e} &= 56.1\,\mathrm{cm^2\,V^{-1}\,s^{-1}} \\
C_{s,e} &= 3.41 \times 10^{20}\,\mathrm{cm^{-3}} \\
\beta_e &= 1.98
\end{aligned}$$

$$\begin{aligned}
\mu_{0,h} &= 44.9\,\mathrm{cm^2\,V^{-1}\,s^{-1}} \\
\mu_{max,h} &= 470.5\,\mathrm{cm^2\,V^{-1}\,s^{-1}} \cdot (T \,/\, 300\,\mathrm{K})^{-2.2} \\
C_{r,h} &= 2.23 \times 10^{17}\,\mathrm{cm^{-3}} \\
\alpha_h &= 0.719 \\
\mu_{1,h} &= 29.0\,\mathrm{cm^2\,V^{-1}\,s^{-1}} \\
C_{s,h} &= 6.1 \times 10^{20}\,\mathrm{cm^{-3}} \\
\beta_h &= 2.0 \\
P_c &= 9.23 \times 10^{16}\,\mathrm{cm^{-3}}
\end{aligned}$$

for electrons and holes, respectively.

This model can be selected in the configuration file via the parameter `mobility_model = "masetti"`.

6.2.6 Arora Model

The Arora mobility model [47] parametrizes electron and hole mobility as a function of the total doping concentration of the silicon material. This model requires a doping profile to be loaded for the detector in question, and an error will be returned if the doping profile is missing. The same caveat to doping concentration information in Allpix Squared applies as described in the previous section.

The mobility is parametrized as

$$\mu_e(N) = \mu_{min,e} + \mu_{0,e} / \left(1 + (N/N_{ref,e})^{\alpha}\right)$$
$$\mu_h(N) = \mu_{min,h} + \mu_{0,h} / \left(1 + (N/N_{ref,h})^{\alpha}\right)$$

as taken from equations 8 (for electrons) and 13 (for holes) of [47].

The parameter values are provided at the reference temperature of 300 Kelvin and scaled to different temperatures according to the equation from the Hamburg model. The values implemented in Allpix Squared are taken from Table 1 and the formulas of [47] as

$$\begin{aligned}
\mu_{min,e} &= 88.0\,\mathrm{cm^2\,V^{-1}\,s^{-1}} \cdot (T \,/\, 300\,\mathrm{K})^{-0.57} \\
\mu_{0,e} &= 7.40 \times 10^8\,\mathrm{cm^2\,V^{-1}\,s^{-1}} \cdot T^{-2.33} \\
N_{ref,e} &= 1.26 \times 10^{17}\,\mathrm{cm^{-3}} \cdot (T \,/\, 300\,\mathrm{K})^{2.4}
\end{aligned}$$

$$\begin{aligned}
\mu_{min,h} &= 54.3\,\mathrm{cm^2\,V^{-1}\,s^{-1}} \cdot (T \,/\, 300\,\mathrm{K})^{-0.57} \\
\mu_{0,h} &= 1.36 \times 10^8\,\mathrm{cm^2\,V^{-1}\,s^{-1}} \cdot T^{-2.23} \\
N_{ref,h} &= 2.35 \times 10^{17}\,\mathrm{cm^{-3}} \cdot (T \,/\, 300\,\mathrm{K})^{2.4} \\
\alpha &= 0.88 \cdot (T \,/\, 300\,\mathrm{K})^{-0.146}
\end{aligned}$$

for electrons and holes, respectively.

This model can be selected in the configuration file via the parameter `mobility_model = "arora"`.

6.2.7 Extended Canali Model

This model extends the Jacoboni-Canali model described with other doping concentration dependent, low-field models such as the Masetti model. This technique is for example used in the Synopsys Sentaurus TCAD software.

The mobility is then parametrized using the two models as

$$\mu(E, N) = \frac{\mu_m(N)}{\left(1 + (\mu_m(N) \cdot E / v_m)^{\beta}\right)^{1/\beta}},$$

where $\mu_m(N)$ is the mobility from the Masetti model and v_m, β are the respective parameters from the Canali model.

This model can be selected in the configuration file via the parameter `mobility_model = "masetti_canali"`.

6.2.8 Ruch-Kino Model

The Ruch-Kino mobility model [48] parametrizes electron and hole mobility in GaAs sensor material. The model parameters implemented in Allpix Squared is taken from measurements [49].

The mobility is parametrized as

$$\begin{aligned} \mu_e(E) &= \mu_{0,e} && \text{for} \quad E < E_0 \\ &= \mu_{0,e} / \sqrt{1 + (E - E_0)^2 / E_c^2} && \text{for} \quad E \geq E_0 \end{aligned}$$

$$\mu_h(E) = \mu_{0,h}.$$

The values implemented in Allpix Squared are:

$$\begin{aligned} E_0 &= 3.1 \times 10^3 \,\mathrm{V\,cm^{-1}} \\ E_c &= 1.36 \times 10^3 \,\mathrm{V\,cm^{-1}} \\ \mu\_0, e &= 7.6 \times 10^3 \,\mathrm{cm^2\,V^{-1}\,s^{-1}} \\ \mu\_0, h &= 3.2 \times 10^2 \,\mathrm{cm^2\,V^{-1}\,s^{-1}} \end{aligned}$$

This model can be selected in the configuration file via the parameter `mobility_model = "ruch_kino"`.

6.2.9 Quay Model

The Quay mobility model describes the mobility of electron and holes in a large range of semiconductor materials. In the original publication [50], the saturation velocity is modeled via the relation

$$v_{sat}\,(T) = \frac{v_{sat,300}}{(1 - A) + A \cdot (T / 300\,K)},$$

with the saturation velocity at 300 Kelvin and the free parameter A.

In Allpix Squared the mobility is determined according to a model published in [51], as a function of the saturation velocity v_{sat}, the electrical field E and the critical field E_C:

$$\mu_e\,(E) = \frac{v_{sat}}{E_C \cdot \sqrt{1 + (E / E_C)^2}}.$$

The critical field in turn is defined as the saturation velocity divided by the mobility at zero field, where the zero-field mobility scales with temperature according to [51]:

$$E_C(T) = \frac{v_{sat}}{M\,T^{-\gamma}}.$$

The model has been implemented for silicon, germanium and gallium arsenide. Parameters for several other compound semiconductors are given in [50] and [52]. The parameters implemented in Allpix Squared and their references are listed in the table below.

Material	Parameter	Electrons	Holes	References
Silicon	$v_{sat,300}$ [$\mathrm{cm\,s^{-1}}$]	1.02×10^{7}	0.72×10^{7}	[50]
	A	0.74	0.37	[50]
	M [$\mathrm{cm^2\,K^{\gamma}\,V^{-1}\,s^{-1}}$]	1.43×10^{9}	1.35×10^{8}	[43]
	γ	2.42	2.2	[43]
Germanium	$v_{sat,300}$ [$\mathrm{cm\,s^{-1}}$]	0.7×10^{7}	0.63×10^{7}	[50]
	A	0.45	0.39	[50]
	M [$\mathrm{cm^2\,K^{\gamma}\,V^{-1}\,s^{-1}}$]	5.66×10^{7}	1.05×10^{9}	[51, 52]
	γ	1.68	2.33	[51, 52]
Gallium Arsenide	$v_{sat,300}$ [$\mathrm{cm\,s^{-1}}$]	0.72×10^{7}	0.9×10^{7}	[50]
	A	0.44	0.59	[50]
	M [$\mathrm{cm^2\,K^{\gamma}\,V^{-1}\,s^{-1}}$]	2.5×10^{6}	6.3×10^{7}	[52]
	γ	1.0	2.1	[52]

6.2.10 Levinshtein Mobility

The Levinshtein mobility model describes the mobility of electron and holes in Gallium Nitride. The publication [53] models the electron and hole mobilities as a function of doping concentration and temperature. The temperature dependent model follows the relation

$$\mu_e(T,N) = \frac{\mu_{max,e}}{\frac{1}{B_e(N)(T/T_0)^{\beta_e}} + \left(\frac{T}{T_0}\right)^{\alpha_e}}$$
$$\mu_h(T,N) = \frac{\mu_{max,h}}{\frac{1}{B_h(N)} + \left(\frac{T}{T_0}\right)^{\alpha_h}}$$
$$B_i(N) = \left[\frac{\mu_{min,i} + \mu_{max,i}\left(\frac{N_{ref,i}}{N}\right)^{\gamma_i}}{\mu_{max,i} - \mu_{min,i}}\right]\Bigg|_{T=T_0}$$

as taken from equations 6 and 7. The following parameters in use are taken from tables 1 and 2 in the reference publication:

$$
\begin{aligned}
\mu\_max, e &= 1000\,\mathrm{cm}^2\,\mathrm{V}^{-1}\,\mathrm{s}^{-1} \\
\mu\_min, e &= 55\,\mathrm{cm}^2\,\mathrm{V}^{-1}\,\mathrm{s}^{-1} \\
N_{ref,e} &= 2 \times 10^17 \\
\alpha_e &= 2.0 \\
\beta_e &= 0.7 \\
\gamma_e &= 1.0 \\
\mu\_max, h &= 170\,\mathrm{cm}^2\,\mathrm{V}^{-1}\,\mathrm{s}^{-1} \\
\mu\_min, h &= 3\,\mathrm{cm}^2\,\mathrm{V}^{-1}\,\mathrm{s}^{-1} \\
N_{ref,h} &= 3 \times 10^17 \\
\alpha_h &= 5.0 \\
\gamma_h &= 2.0
\end{aligned}
$$

This model can be selected in the configuration file via the parameter `mobility_model = "levinshtein"`.

6.2.11 Constant Mobility

Some simulations require constant charge carrier mobility values $\mu = \text{const}$. This can be simulated with this model. For more complex mobility dependencies the custom mobility model described next should be used.

This model can be selected in the configuration file via the parameter `mobility_model = "constant`.

It requires the additional configuration keys `mobility_electron` and `mobility_hole` to be present in the module configuration section, for example:

```
mobility_model = "constant"
mobility_electron = 1000cm*cm/V/s
mobility_hole = 50cm*cm/V/s
```

6.2.12 Custom Mobility Models

Allpix Squared provides the possibility to use fully custom mobility models. In order to use a custom model, the parameter `mobility_model = "custom"` needs to be set in the configuration file. Additionally, the following configuration keys have to be provided:

- `mobility_function_electrons`: The formula describing the electron mobility.
- `mobility_function_holes`: The formula describing the hole mobility.

The functions defined via these parameters can depend on the local electric field and the local doping concentration. In order to use the electric field magnitude in the formula, an `x` has to be placed at the respective position, for the doping concentration a `y` is used as placeholder.

Parameters of the functions can either be placed directly in the formulas in framework-internal units, or provided separately as arrays via the `mobility_parameters_electrons`

and `mobility_parameters_electrons`. Placeholders for parameters in the formula are denoted with squared brackets and a parameter number, for example `[0]` for the first parameter provided. Parameters specified separately from the formula can contain units which will be interpreted automatically.

Warning: Parameters directly placed in the mobility formula have to be supplied in framework-internal units since the function will be evaluated with both electric field strength and doping concentration in internal units. It is recommended to use the possibility of separately configuring the parameters and to make use of units to avoid conversion mistakes.

The following set of parameters re-implements the Jacoboni-Canali mobility model using a custom mobility model. The mobility is calculated at a fixed temperature of 293 Kelvin.

```
# Replicating the Jacoboni-Canali mobility model at T = 293K
mobility_model = "custom"

mobility_function_electrons = "[0]/[1]/pow(1.0+pow(x/[1],[2]),1.0/[2])"
mobility_parameters_electrons = 1.0927393e7cm/s, 6729.24V/cm, 1.0916

mobility_function_holes = "[0]/[1]/pow(1.0+pow(x/[1],[2]),1.0/[2])"
mobility_parameters_holes = 8.447804e6cm/s, 17288.57V/cm, 1.2081
```

Warning: It should be noted that the temperature passed via the module configuration is not evaluated for the custom mobility model, but the model parameters need to be manually adjusted to the required temperature.

The interpretation of the custom mobility functions is based on the `ROOT::TFormula` class [54] and supports all corresponding features, mathematical expressions and constants.

6.3 Charge Carrier Lifetime & Recombination

Allpix Squared provides the possibility to simulate finite lifetimes of charge carriers as a function of the local doping concentration via non-radiative recombination processes. While most of these models require the *total doping concentration* $N_D + N_A$ as parameter, the doping profile used throughout Allpix Squared provides the *effective doping concentration* $N_D - N_A$ since this also encodes the majority charge carriers via its sign - an information relevant to some of the models. However, in the parts of a silicon detector relevant for this simulation, i.e. the sensing volume, the difference between effective and total concentration is expected to be negligible. Therefore the two values are treated as equivalent throughout the lifetime models and the doping concentration is taken as the absolute value $N = |N_D - N_A|$.

Whether a charge carrier has recombined with the lattice is calculated for every step of the simulation using the relation

$$p < 1 - e^{-dt/\tau(N)}$$

where p is a recombination probability, drawn from a uniform distribution with $[0, 1]$, dt is the last time step of the charge carrier motion and τ the lifetime for the local doping concentration calculated by the models described in the following. If the above equation evaluates to *false*, the charge carrier still exists, if it evaluates to *true* it has been recombined with the lattice.

Finite charge carrier lifetime can be simulated by all propagation modules and comprise the following models:

6.3.1 Shockley-Read-Hall Recombination

This model describes the finite lifetime based on Shockley-Read-Hall or trap-assisted recombination of charge carriers with the lattice [55, 56]. The lifetime is calculated using the Shockley-Read-Hall relation as given by [57]:

$$\tau(N) = \frac{\tau_0}{1 + \frac{N}{N_{d0}}}$$

where τ_0 and N_{d0} are reference lifetime and doping concentration, for electrons and holes respectively. The parameter values implemented in Allpix Squared are taken from [57] and the Synopsys Sentaurus TCAD software manual as

$$\begin{aligned} \tau_{0,e} &= 1 \times 10^{-5}\,\mathrm{s} \\ N_{d0,e} &= 1 \times 10^{16}\,\mathrm{cm}^{-3} \end{aligned}$$

$$\begin{aligned} \tau_{0,h} &= 4.0 \times 10^{-4}\,\mathrm{s} \\ N_{d0,h} &= 7.1 \times 10^{15}\,\mathrm{cm}^{-3} \end{aligned}$$

for electrons and holes, respectively.

The temperature dependence of the Shockley-Read-Hall lifetime is scaled following the low-temperature approximation model presented [58] as:

$$\tau(N, T) = \tau(N) \cdot \left(\frac{300\,\mathrm{K}}{T}\right)^{3/2}$$

This model can be selected in the configuration file via the parameter `recombination_model = "srh"`.

6.3.2 Auger Recombination

At high doping levels exceeding $5 \times 10^{18}\,\mathrm{cm}^{-3}$ [57], the Auger recombination model becomes increasingly important. It assumes that the excess energy created by electron-hole recombinations is transferred to another electron (*e-e-h process*) or another hole (*e-h-h process*). The total recombination rate is then given by [59]:

$$R_{Auger} = C_n n^2 p + C_p n p^2$$

where C_n and C_p are the Auger coefficients. The first term corresponds to the e-e-h process and the second term to the e-h-h process. In highly-doped silicon, the Auger lifetime for minority charge carriers can be written as:

$$\tau(N) = \frac{1}{C_a \cdot N^2}$$

where $C_a = C_n + C_p$ is the ambipolar Auger coefficient, taken as $C_a = 3.8 \times 10^{-31}\,\mathrm{cm}^6\,\mathrm{s}^{-1}$ from [60].

This recombination mode applies to minority charge carriers only, majority charge carriers have an infinite life time under this model and the recombination equation will always evaluate to *true*.

This model can be selected in the configuration file via the parameter `recombination_model = "auger"`.

6.3.3 Combined SRH/Auger Recombination

This model combines the charge carrier recombination from the Shockley-Read-Hall and the Auger model by inversely summing the individual lifetimes calculated by the models via

$$\begin{aligned} \tau^{-1}(N) &= \tau_{srh}^{-1}(N) + \tau_a^{-1}(N) && \text{for minority charge carriers} \\ &= \tau_{srh}^{-1}(N) && \text{for majority charge carriers} \end{aligned}$$

where $\tau_{srh}(N)$ is the Shockley-Read-Hall and $\tau_a(N)$ the Auger lifetime. The latter is only taken into account for minority charge carriers.

This model can be selected in the configuration file via the parameter `recombination_model = "srh_auger"`.

6.3.4 Recombination with Constant Lifetimes

Some materials require constant lifetimes for charge carriers $\tau(N) = \tau$. This model requires the additional configuration keys `lifetime_electron` and `lifetime_hole` to be present in the module configuration section, for example:

```
# Constant lifetimes for electrons and holes in GaAs with Cr
 ↪  compensation:
recombination_model = "constant"
lifetime_electron = 30ns
lifetime_hole = 4.5ns
```

This model can be selected in the configuration file via the parameter `recombination_model = "constant"`.

6.3.5 Custom Recombination Models

Allpix Squared provides the possibility to use fully custom recombination models. In order to use a custom model, the parameter `recombination_model = "custom"` needs to be set in the configuration file. Additionally, the following configuration keys have to be provided:

- `lifetime_function_electrons`: The formula describing the electron lifetime.
- `lifetime_function_holes`: The formula describing the hole lifetime.

The functions defined via these parameters can depend on the local doping concentration. In order to use the doping concentration in the formula, an `x` has to be placed at the respective position.

Parameters of the functions can either be placed directly in the formulas in framework-internal units, or provided separately as arrays via the `lifetime_parameters_electrons` and `lifetime_parameters_electrons`. Placeholders for parameters in the formula are denoted with squared brackets and a parameter number, for example `[0]` for the first parameter provided. Parameters specified separately from the formula can contain units which will be interpreted automatically.

Warning: Parameters directly placed in the recombination formula have to be supplied in framework-internal units since the function will be evaluated with the doping concentration in internal units. It is recommended to use the possibility of separately configuring the parameters and to make use of units to avoid conversion mistakes.

The following set of parameters re-implements the Shockley-Read-Hall recombination model using a custom recombination model. The lifetimes are calculated at a fixed temperature of 293 Kelvin.

```
# Replicating the Shockley-Read-Hall model at T = 293K
recombination_model = "custom"

lifetime_function_electrons = "[0]/(1 + x / [1])"
lifetime_parameters_electrons = 1.036e-5s, 1e16/cm/cm/cm

lifetime_function_holes = "[0]/(1 + x / [1])"
lifetime_parameters_holes = 4.144e-4s, 7.1e15/cm/cm/cm
```

Warning: It should be noted that the temperature passed via the module configuration is not evaluated for the custom recombination model, but the model parameters need to be manually adjusted to the required temperature.

The interpretation of the custom recombination functions is based on the `ROOT::TFormula` class [54] and supports all corresponding features, mathematical expressions and constants.

6.4 Trapping and Detrapping of Charge Carriers

Allpix Squared provides the possibility to simulate the trapping and detrapping of charge carriers as a consequence of radiation induced lattice defects. Several models exist, that quantify the effective lifetime of electrons and holes, respectively, as a function of the fluence and, partially, the temperature. The fluence needs to be provided to the corresponding propagation module, and is always interpreted as 1-MeV neutron equivalent fluence [61].

The decision on whether a charge carrier has been trapped during a step during the propagation process is calculated similarly to the recombination processes, described in Section 6.3.

It should be noted that the trapping of charge carriers is only one of several effects induced by radiation damage. In Allpix Squared, these effects are treated independently, i.e. defining the fluence for a propagation module will not affect any other process than trapping.

have been extracted under certain annealing conditions. A dependency on annealing conditions has not been implemented here. Please refer to the corresponding reference publications for further details.

The trapping probability is calculated as an exponential decay as a function of the simulation timestep as

$$p_{e,h} = \left(1 - \exp^{1\frac{\delta t}{\tau_{e,h}}}\right)$$

where δt is the simulation timestep and $\tau e, h$ the effective lifetime of electrons and holes, respectively. At the same time, a total time spent in the trap is calculated if a detrapping model is selected. Here, the time until the charge carrier is de-trapped is calculated as

$$\delta t = -\tau_{e.h} \ln 1 - p$$

where p is a probability randomly chosen from a uniform distribution between 0 and 1.

6.4.1 Trapping Models

The following models for trapping of charge carriers can be selected:

Ljubljana

In the Ljubljana (sometimes referred to as *Kramberger*) model [62], the trapping time follows the relation

$$\tau^{-1}(T) = \beta(T)\Phi_{eq},$$

where the temperature scaling of β is given as

$$\beta(T) = \beta(T_0) \left(\frac{T}{T_0}\right)^{\kappa},$$

extracted at the reference temperature of $T_0 = -10\,°\mathrm{C}$.

The parameters used in Allpix Squared are

$$\begin{aligned} \beta_e(T_0) &= 5.6 \times 10^{-16}\,\mathrm{cm}^2\,\mathrm{ns}^{-1} \\ \kappa_e &= -0.86 \end{aligned}$$

$$\begin{aligned} \beta_h(T_0) &= 7.7 \times 10^{-16}\,\mathrm{cm}^2\,\mathrm{ns}^{-1} \\ \kappa_h &= -1.52 \end{aligned}$$

for electrons and holes, respectively.

While [62] quotes different values for β for irradiation with neutrons, pions and protons, the values for protons have been applied here.

The parameters arise from measurements of the were obtained evaluating current signals of irradiated sensors via light injection at fluences up to $\Phi_{eq} = 2 \times 10^{14}\ n_{eq}\,\mathrm{cm}^2$.

This model can be selected in the configuration file via the parameter `trapping_model = "ljubljana"`.

Dortmund

The Dortmund (sometimes referred to as *Krasel*) model [63], describes the effective trapping times as

$$\tau^{-1} = \gamma \Phi_{eq},$$

with the parameters

$$\gamma_e = 5.13 \times 10^{-16}\,\mathrm{cm}^2\,\mathrm{ns}^{-1}$$
$$\gamma_h = 5.04 \times 10^{-16}\,\mathrm{cm}^2\,\mathrm{ns}^{-1}$$

for electrons and holes, respectively.

The values have been extracted evaluating current signals of irradiated sensors via light injection at fluences up to $\Phi_{eq} = 8.9 \times 10^{14}\ n_{eq}\,\mathrm{cm}^2$, at a temperature of $T = 0\,°\mathrm{C}$. No temperature scaling is provided. Values for neutron and proton irradiation have been evaluated in [63], Allpix Squared makes use of the values for proton irradiation.

This model can be selected in the configuration file via the parameter `trapping_model = "dortmund"`.

CMS Tracker

This effective trapping model has been developed by the CMS Tracker Group. It follows the results of [64], with measurements at fluences of up to $\Phi_{eq} = 3 \times 10^{15}\ n_{eq}\,\mathrm{cm}^2$, at a temperature of $T = -20\,°\mathrm{C}$ and an irradiation with protons.

The interpolation of the results follows the relation

$$\tau^{-1} = \beta \Phi_{eq} + \tau_0^{-1}$$

with the parameters

$$\beta_e(T_0) = 1.71 \times 10^{-16}\,\mathrm{cm}^2\,\mathrm{ns}^{-1}$$
$$\tau_{0,e}^{-1} = -0.114\,\mathrm{ns}^{-1}$$

$$\beta_h(T_0) = 2.79 \times 10^{-16}\,\mathrm{cm}^2\,\mathrm{ns}^{-1}$$
$$\tau_{0,h}^{-1} = -0.093\,\mathrm{ns}^{-1}$$

for electrons and holes, respectively.

No temperature scaling is provided.

This model can be selected in the configuration file via the parameter `trapping_model = "cmstracker"`.

Mandic

The Mandić model [65] is an empirical model developed from measurements with high fluences ranging from $\Phi_{eq} = 5 \times 10^{15}\ n_{eq}\,\mathrm{cm}^2$ to $\Phi_{eq} = 1 \times 10^{17}\ n_{eq}\,\mathrm{cm}^2$ and describes the lifetime via

$$\tau = c\Phi_{eq}^{\kappa}$$

with the parameters

$$\begin{aligned} c_e &= 0.054\,\mathrm{ns\,cm}^{-2} \\ \kappa_e &= -0.62 \end{aligned}$$

$$\begin{aligned} c_h &= 0.0427\,\mathrm{ns\,cm}^{-2} \\ \kappa_h &= -0.62 \end{aligned}$$

for electrons and holes, respectively.

The parameters for electrons are taken from [65], for measurements at a temperature of $T = -20\,^{\circ}\mathrm{C}$, and the results extrapolated to $T = -30\,^{\circ}\mathrm{C}$. A scaling from electrons to holes was performed based on the default values in Weightfield2 [66].

This model can be selected in the configuration file via the parameter `trapping_model = "mandic"`.

Constant Trapping Model

For some situations or materials, a constant trapping probability is necessary. This can be achieved with the constant trapping model. Here, the lifetimes are constant and set from the values provided in the configuration file with the parameters `trapping_time_electron` and `trapping_time_hole`:

```
# Constant trapping times for electrons and holes:
trapping_model = "constant"
trapping_time_electron = 5ns
trapping_time_hole = 5ns
```

This model can be selected in the configuration file via the parameter `trapping_model = "constant"`.

Custom Trapping Model

Similarly to the mobility models described above, Allpix Squared provides the possibility to use fully custom trapping models. The model requires the following configuration keys:

- `trapping_function_electrons`: The formula describing the effective electron trapping time.

- `trapping_function_holes`: The formula describing the effective hole trapping time.

The functions defined via these parameters can depend on the local electric field. In order to use the electric field magnitude in the formula, an `x` has to be placed at the respective position.

Parameters of the functions can either be placed directly in the formulas in framework-internal units, or provided separately as arrays via the `trapping_parameters_electrons` and `trapping_parameters_holes`. Placeholders for parameters in the formula are denoted with squared brackets and a parameter number, for example `[0]` for the first parameter provided. Parameters specified separately from the formula can contain units which will be interpreted automatically.

Note: Both fluence and temperature are not inherently available in the custom trapping model, but need to be provided as additional parameters as described above.

The following configuration parameters replicate the Ljubljana model using a custom trapping model.

```
# Replicating the Ljubljana trapping model at a temperature of 293 K and
 ↪  a neutron equivalent fluence of 1e14 neq/cm^2
trapping_model = "custom"

trapping_function_electrons = "1/([0]*pow([1]/263,[2]))/[3]"
trapping_parameters_electrons = 5.6e-16cm*cm/ns, 293K, -0.86, 1e14/cm/cm

trapping_function_holes = "1/([0]*pow([1]/263,[2]))/[3]"
trapping_parameters_holes = 7.7e-16cm*cm/ns, 293K, -1.52, 1e14/cm/cm
```

Fixed, effective trapping times can be defined using this model similar to the following configuration example.

```
# Defining a fixed trapping time
trapping_model = "custom"

trapping_function_electrons = "[0]"
trapping_parameters_electrons = 5ns

trapping_function_holes = "[0]"
trapping_parameters_holes = 7ns
```

This model can be selected in the configuration file via the parameter `trapping_model = "custom"`.

6.4.2 Detrapping Models

The detrapping is configured via the `detrapping_model` parameter. Currently, only `detrapping_model = "none"` and `detrapping_model = "constant"` are supported.

The following models for trapping of charge carriers can be selected:

Constant Detrapping Model

A constant detrapping probability, with the detrapping time defined separately for electrons and holes, can be implemented via the constant detrapping model. This model requires the parameters `detrapping_time_electron` and `detrapping_time_hole` to be configured.

```
# Constant detrapping times for electrons and holes:
detrapping_model = "constant"
detrapping_time_electron = 10ns
detrapping_time_hole = 10ns
```

6.5 Impact Ionization

Allpix Squared implements charge multiplication via impact ionization models. These models are only used by propagation modules which perform a step-by-step simulation of the charge carrier motion.

The per-step gain g is calculated for all models as exponential of the model-dependent impact ionization coefficient α and the length of the step l performed in the respective electric field. If the electric field strength stays below a configurable threshold E_{thr}, unity gain is assumed:

$$g(E,T) = \begin{cases} e^{l\cdot\alpha(E,T)} & E > E_{\text{thr}} \\ 1.0 & E < E_{\text{thr}} \end{cases}$$

The impact ionization coefficient α is calculated depending on the selected impact ionization model. The models themselves are described below.

The number of additional charge carriers generated per step n is determined via a stochastic approach by applying the following equation dependent on a random number drawn from a uniform distribution $u(0,1)$

$$n = \frac{\ln(u)}{\ln(1-1/g)} = \frac{1}{\log_u(1-1/g)}$$

This distribution is applied e.g. in Garfield++[**garfieldpp**] and represents a microscopic simulation of Yule processes.

The number of secondary charge carriers generated from impact ionization is calculated for every individual charge carrier within a group of charge carriers and summed per

propagation step. Additional charge carriers are then added to the group (same-type carriers) or deposited (opposite-type) at the end of the corresponding step.

This algorithm results in a mean number of secondaries generated equal to

$$< n_{total} >= \exp\left(\int_{x_0}^{x_n} \alpha(x)dx\right)$$

for sufficiently low step sizes.

The following impact ionization models are available:

6.5.1 Massey Model

The Massey model [67] describes impact ionization as a function of the electric field E. The ionization coefficients are parametrized as

$$\alpha(E,T) = Ae^{-\frac{B(T)}{E}},$$

where A and $B(T)$ are phenomenological parameters, defined for electrons and holes respectively. While A is assumed to be temperature-independent, parameter B exhibits a temperature dependence and is defined as

$$B(T) = C + D \cdot T.$$

Original publication

The parameter values implemented in Allpix Squared are taken from Section 3 of [67] as:

$$\begin{aligned} A_e &= 4.43 \times 10^5\,/\mathrm{cm} \\ C_e &= 9.66 \times 10^5\,\mathrm{V/cm} \\ D_e &= 4.99 \times 10^2\,\mathrm{V/cm/K} \end{aligned}$$

$$\begin{aligned} A_h &= 1.13 \times 10^6\,/\mathrm{cm} \\ C_h &= 1.71 \times 10^6\,\mathrm{V/cm} \\ D_h &= 1.09 \times 10^3\,\mathrm{V/cm/K} \end{aligned}$$

for electrons and holes, respectively.

This model can be selected in the configuration file via the parameter `multiplication_model = "massey"`.

Optimized parameters

An optimized parametrization of the Massey model based on measurements with an infrared laser is implemented in Allpix Squared, based on Table 2 of [68] with the values:

$$\begin{aligned} A_e &= 1.186 \times 10^6\,/\mathrm{cm} \\ C_e &= 1.020 \times 10^6\,\mathrm{V/cm} \\ D_e &= 1.043 \times 10^3\,\mathrm{V/cm/K} \end{aligned}$$

$$\begin{aligned} A_h &= 2.250 \times 10^6\,/\mathrm{cm} \\ C_h &= 1.851 \times 10^6\,\mathrm{V/cm} \\ D_h &= 1.828 \times 10^3\,\mathrm{V/cm/K} \end{aligned}$$

for electrons and holes, respectively.

This model can be selected in the configuration file via the parameter `multiplication_model = "massey_optimized"`.

6.5.2 Van Overstraeten-De Man Model

The Van Overstraeten-De Man model [69] describes impact ionization using Chynoweth's law, given by

$$\alpha(E,T) = \gamma(T) \cdot a_\infty \cdot e^{-\frac{\gamma(T)\cdot b}{E}},$$

For holes, two sets of impact ionization parameters $p = \{a_\infty, b\}$ are used depending on the electric field:

$$p = \begin{cases} p_{\mathrm{low}} & E < E_0 \\ p_{\mathrm{high}} & E > E_0 \end{cases}$$

Temperature scaling of the ionization coefficient is performed via the $\gamma(T)$ parameter following the Synposys Sentaurus TCAD user manual as:

$$\gamma(T) = \tanh\left(\frac{\hbar\omega_{op}}{2k_\mathrm{B}\cdot T_0}\right) \cdot \tanh\left(\frac{\hbar\omega_{op}}{2k_\mathrm{B}\cdot T}\right)^{-1}$$

with $\hbar\omega_{op} = 0.063\,\mathrm{eV}$ and the Boltzmann constant $k_\mathrm{B} = 8.6173 \times 10^{-5}\,\mathrm{eV/K}$. The value of the reference temperature T_0 is not entirely clear as it is never stated explicitly, a value of $T_0 = 300\,\mathrm{K}$ is assumed.

Original publication

The other model parameter values implemented in Allpix Squared are taken from the abstract of [69] as:

$$\begin{aligned} E_0 &= 4.0 \times 10^5\,\mathrm{V/cm} \\ a_{\infty,e} &= 7.03 \times 10^5\,/\mathrm{cm} \\ b_e &= 1.231 \times 10^6\,\mathrm{V/cm} \end{aligned}$$

$$\begin{aligned} a_{\infty,h,\mathrm{low}} &= 1.582 \times 10^6\,/\mathrm{cm} \\ a_{\infty,h,\mathrm{high}} &= 6.71 \times 10^5\,/\mathrm{cm} \\ b_{h,\mathrm{low}} &= 2.036 \times 10^6\,\mathrm{V/cm} \\ b_{h,\mathrm{high}} &= 1.693 \times 10^6\,\mathrm{V/cm} \end{aligned}$$

This model can be selected in the configuration file via the parameter `multiplication_model = "overstraeten"`.

Optimized parameters

An optimized parametrization of the Van Overstraeten-De Man model based on measurements with an infrared laser is implemented in Allpix Squared, based on Table 3 of [68] with the following parameter values:

$$\begin{aligned} a_{\infty,e} &= 1.149 \times 10^6\,/\mathrm{cm} \\ b_e &= 1.325 \times 10^6\,\mathrm{V/cm} \end{aligned}$$

$$\begin{aligned} a_{\infty,h} &= 2.519 \times 10^6\,/\mathrm{cm} \\ b_h &= 2.428 \times 10^6\,\mathrm{V/cm} \end{aligned}$$

$$\hbar\omega_{op} = 0.0758\,\mathrm{eV}$$

In contrast to the original model, this publication uses a parametrization without differentiating between low and high field regions, hence only one parameter value is provided for each of $a_{\infty,h}$ and b_h.

This model can be selected in the configuration file via the parameter `multiplication_model = "overstraeten_optimized"`.

6.5.3 Okuto-Crowell Model

The Okuto-Crowell model [70] defines the impact ionization coefficient similarly to the above models but in addition features a linear dependence on the electric field strength E. The coefficient is given by:

$$\alpha(E,T) = a(T) \cdot E \cdot e^{-\left(\frac{b(T)}{E}\right)^2}.$$

The two parameters a, b are temperature dependent and scale with respect to the reference temperature $T_0 = 300\,\mathrm{K}$ as:

$$a(T) = a_{300}\left[1 + c\,(T - T_0)\right]$$
$$b(T) = a_{300}\left[1 + d\,(T - T_0)\right]$$

6.5.4 Original publication

The parameter values implemented in Allpix Squared are taken from Table 1 of [70], using the values for silicon, as:

$$\begin{aligned} a_{300,e} &= 0.426\,/\mathrm{V} \\ c_e &= 3.05 \times 10^{-4} \\ b_{300,e} &= 4.81 \times 10^5\,\mathrm{V/cm} \\ d_e &= 6.86 \times 10^{-4} \end{aligned}$$

$$\begin{aligned} a_{300,h} &= 0.243\,/\mathrm{V} \\ c_h &= 5.35 \times 10^{-4} \\ b_{300,h} &= 6.53 \times 10^5\,\mathrm{V/cm} \\ d_h &= 5.67 \times 10^{-4} \end{aligned}$$

This model can be selected in the configuration file via the parameter `multiplication_model = "okuto"`.

Optimized parameters

An optimized parametrization of the Okuto-Crowell model based on measurements with an infrared laser is implemented in Allpix Squared, based on Table 4 of [68] with the following parameter values:

$$\begin{aligned} a_{300,e} &= 0.289\,/\mathrm{V} \\ c_e &= 9.03 \times 10^{-4} \\ b_{300,e} &= 4.01 \times 10^5\,\mathrm{V/cm} \\ d_e &= 1.11 \times 10^{-3} \end{aligned}$$

$$\begin{aligned} a_{300,h} &= 0.202\,/\mathrm{V} \\ c_h &= -2.20 \times 10^{-3} \\ b_{300,h} &= 6.40 \times 10^5\,\mathrm{V/cm} \\ d_h &= 8.25 \times 10^{-4} \end{aligned}$$

This model can be selected in the configuration file via the parameter `multiplication_model = "okuto_optimized"`.

6.5.5 Bologna Model

The Bologna model [71] describes impact ionization for experimental data in an electric field range from 130 kV/cm to 230 kV/cm and temperatures up to 400 °C. The impact ionization coefficient takes a different form than the previous models and is given by

$$\alpha(E,T) = \frac{E}{a(T) + b(T)e^{d(T)/(E+c(T))}},$$

for both electrons and holes. The temperature-dependent parameters $a(T), b(T), c(T)$ and $d(T)$ are defined as:

$$\begin{aligned} a(T) &= a_0 + a_1 T^{a_2} \\ b(T) &= b_0 e^{b_1 T} \\ c(T) &= c_0 + c_1 T^{c_2} + c_3 T^2 \\ d(T) &= d_0 + d_1 T + d_2 T^2 \end{aligned}$$

The parameter values implemented in Allpix Squared are taken from Table 1 of [71] as:

$$
\begin{aligned}
a_{0,e} &= 4.3383\,\mathrm{V} \\
a_{1,e} &= -2.42 \times 10^{-12}\,\mathrm{V} \\
a_{2,e} &= 4.1233 \\
b_{0,e} &= 0.235\,\mathrm{V} \\
b_{1,e} &= 0 \\
c_{0,e} &= 1.6831 \times 10^{4}\,\mathrm{V/cm} \\
c_{1,e} &= 4.3796\,\mathrm{V/cm} \\
c_{2,e} &= 1 \\
c_{3,e} &= 0.13005\,\mathrm{V/cm} \\
d_{0,e} &= 1.2337 \times 10^{6}\,\mathrm{V/cm} \\
d_{1,e} &= 1.2039 \times 10^{3}\,\mathrm{V/cm} \\
d_{2,e} &= 0.56703\,\mathrm{V/cm}
\end{aligned}
$$

$$
\begin{aligned}
a_{0,h} &= 2.376\,\mathrm{V} \\
a_{1,h} &= 1.033 \times 10^{-2}\,\mathrm{V} \\
a_{2,h} &= 1 \\
b_{0,h} &= 0.17714\,\mathrm{V} \\
b_{1,h} &= -2.178 \times 10^{-3}\,\mathrm{/K} \\
c_{1,h} &= 0 \\
c_{1,h} &= 9.47 \times 10^{-3}\,\mathrm{V/cm} \\
c_{2,h} &= 2.4924 \\
c_{3,h} &= 0 \\
d_{0,h} &= 1.4043 \times 10^{6}\,\mathrm{V/cm} \\
d_{1,h} &= 2.9744 \times 10^{3}\,\mathrm{V/cm} \\
d_{2,h} &= 1.4829\,\mathrm{V/cm}
\end{aligned}
$$

This model can be selected in the configuration file via the parameter `multiplication_model = "bologna"`.

6.5.6 Custom Impact Ionization Models

Allpix Squared provides the possibility to use fully custom impact ionization models. In order to use a custom model, the parameter `multiplication_model = "custom"` needs to be set in the configuration file. Additionally, the following configuration keys have to be provided:

- `multiplication_function_electrons`: The formula describing the electron impact ionization gain.
- `multiplication_function_holes`: The formula describing the hole impact ionization gain.

The functions defined via these parameters can depend on the local electric field. In order to use the electric field magnitude in the formula, an `x` has to be placed at the respective position.

Parameters of the functions can either be placed directly in the formulas in framework-internal units, or provided separately as arrays via the `multiplication_parameters_electrons` and `multiplication_parameters_electrons`. Placeholders for parameters in the formula are denoted with squared brackets and a parameter number, for example `[0]` for the first parameter provided. Parameters specified separately from the formula can contain units which will be interpreted automatically.

Warning: Parameters directly placed in the impact ionization formula have to be supplied in framework-internal units since the function will be evaluated with both electric field strength and doping concentration in internal units. It is recommended to use the possibility of separately configuring the parameters and to make use of units to avoid conversion mistakes.

Warning: It should be noted that the temperature passed via the module configuration is not evaluated for the custom impact ionization model, but the model parameters need to be manually adjusted to the required temperature.

The interpretation of the custom impact ionization functions is based on the `ROOT::TFormula` class [54] and supports all corresponding features, mathematical expressions and constants.

7 Objects

Allpix Squared provides a set of objects which can be used to transfer data between modules and to store the simulation results to file. These objects can be read again from file and dispatched to a secondary simulation chain using the ROOTObjectReader and ROOTObjectWriter modules which dispatch them via the messaging system as explained in Section 4.6.

Objects stored to a ROOT file can be analyzed using C or Python scripts, the example scripts for both languages described in Section 14.3 are provided in the repository.

7.1 Object Types

The list of currently supported objects is given below. A `typedef` is added to every object in order to provide an alternative name for the message which is directly indicating the carried object.

For writing analysis scripts, a detailed description of the code interface for each object can be found in the Object Group of the Doxygen reference manual [5].

7.1.1 MCTrack

The MCTrack objects reflects the state of a particle's trajectory when it was created and when it terminates. Moreover, it allows to retrieve the hierarchy of secondary tracks. This can be done via the parent-child relations the MCTrack objects store, allowing retrieval of the primary track for a given track. Combining this information with MCParticles allows the Monte-Carlo trajectory to be fully reconstructed. In addition to these relational information, the MCTrack stores information on the initial and final point of the trajectory (in *global* coordinates), the initial and final timestamps in global coordinates of the event, the energies (total as well as kinetic only) at those points, the creation process type, name, and the volume it took place in. Furthermore, the particle's PDG id [72] is stored.

Main properties: - Global points where track came into and went out of existence (`getStartPoint()`, `getEndPoint()`) - Global time when the track had its first and last appearance (`getGlobalStartTime()`, `getGlobalEndTime()`) - Initial and final kinetic and total energy (`getKineticEnergyInitial()`, `getTotalEnergyInitial()`, `getKineticEnergyFinal()`, `getTotalEnergyFinal()`)

For more details refer to the code reference.

7.1.2 MCParticle

The Monte-Carlo truth information about the particle passage through the sensor. A start and end point are stored in the object: for events involving a single MCParticle passing through the sensor, the start and end points correspond to the entry and exit points. The exact handling of non-linear particle trajectories due to multiple scattering is up to module. In addition, it provides a member function to retrieve the reference point at the sensor center plane in local coordinates for convenience. The MCParticle also stores an identifier of the particle type, using the PDG particle codes [72], as well as the time it has first been observed in the respective sensor. The MCParticle additionally stores a parent MCParticle object, if available. The lack of a parent doesn't guarantee that this MCParticle originates from a primary particle, but only means that no parent on the given detector exists. Also, the MCParticle stores a reference to the MCTrack it is associated with.

MCParticles provide local and global coordinates in space for both the entry and the exit of the particle in the sensor volume, as well as local and global time information. The global spatial coordinates are calculated with respect to the global reference frame defined in Section 5.1, the global time is counted from the beginning of the event. Local spatial coordinates are determined by the respective detector, the local time measurement references the entry point of the *first* MCParticle of the event into the detector.

Main parameters: - Entry and exit points of the particle in the sensor in local and global coordinates (`getLocalStartPoint()`, `getGlobalStartPoint()`, `getLocalEndPoint()`, `getGlobalEndPoint()`) - The arrival time of the particle in the sensor in local and global coordinates (`getLocalTime()`, `getGlobalTime()`) - PDG id for this particle type (`getParticleID()`) - If the particle is a primary particle or has a parent particle (`isPrimary()`), and the parent particle, if any (`getParent()`) - The track the particle is related to, if any (`getTrack()`)

For more details refer to the code reference.

7.1.3 SensorCharge

This is a meta class for a set of charges within a sensor. The properties of this object are inherited by DepositedCharge and PropagatedCharge objects.

Main parameters: - The position of the set of charges in the sensor in local and global coordinates (`getLocalPosition()`, `getGlobalPosition()`) - The associated time of the set of charges in local and global coordinates (`getLocalTime()`, `getGlobalTime()`) - The sign of the charge carries (`getSign()`) and the amount of charges in the set (`getCharge()`) - The carrier type to check if the charge carriers are electrons or holes (`getType()`)

For more details refer to the code reference.

7.1.4 DepositedCharge

The set of charge carriers deposited by an ionizing particle crossing the active material of the sensor. The object stores the *local* position in the sensor together with the total number of deposited charges in elementary charge units. In addition, the time (in *ns* as

the internal framework unit) of the deposition after the start of the event and the type of carrier (electron or hole) is stored.

Main parameters: - Everything from SensorCharge - The MCParticle that created the deposited charge (`getMCParticle()`)

For more details refer to the code reference.

7.1.5 PropagatedCharge

The set of charge carriers propagated through the sensor due to drift and/or diffusion processes. The object should store the final *local* position of the propagated charges. This is either on the pixel implant (if the set of charge carriers are ready to be collected) or on any other position in the sensor if the set of charge carriers got trapped or was lost in another process. Timing information giving the total time to arrive at the final location, from the start of the event, can also be stored.

Main parameters: - Everything from SensorCharge - The associated DepositedCharge object (`getDepositedCharge()`) - The associated induced pulses, if any (`getPulses()`) - The carrier state of the charge carriers described below (`getState()`)

The following values for the carrier state are possible: - `CarrierState::UNKNOWN`: The final state of the charge carrier is unknown, for example because it might not have been provided by the used propagation algorithm - `CarrierState::MOTION`: The charge carrier was still in motion when the propagation routine finished, for example when the configured integration time was reached - `CarrierState::RECOMBINED`: The charge carrier has recombined with the silicon lattice at the given position - `CarrierState::TRAPPED`: The charge carrier has been trapped by a lattice defect at the given position - `CarrierState::HALTED`: The motion of the charge carrier has stopped, for example because it has reached an implant or the sensor surface

For more details refer to the code reference.

7.1.6 PixelCharge

The set of charge carriers collected at a single pixel. The pixel indices are stored in both the x and y direction, starting from zero for the first pixel. Only the total number of charges at the pixel is currently stored, the timing information of the individual charges can be retrieved from the related PropagatedCharge objects.

Main parameters: - The pixel corresponding to the charge (`getPixel()`) and its index (`getIndex()`) - The charge collected in the pixel (`getCharge()`, `getAbsoluteCharge()`) - The related propagates charges (`getPropagatedCharges()`) - The associated time of the charge in local and global coordinates (`getLocalTime()`, `getGlobalTime()`) - The recorded charge pulse, if any (`getPulse()`)

For more details refer to the code reference.

7.1.7 Pulse

The pulse object is a meta class mainly used to hold the time information of a charge pulse arriving at the collection implant, if such information is available in the simulation. A pulse object always has a fixed time binning chosen during the creation of the object. It inherits from std::vector.

Main parameters: - The total charge of the pulse (`getCharge()`) - The time binning of the pulse (`getBinning()`)

For more details refer to the code reference

7.1.8 PixelHit

The digitised pixel hits after processing the PixelCharge in the detector's front-end electronics. The object allows the storage of both the time and signal value. The time can be stored in an arbitrary unit used to timestamp the hits. The signal can hold different kinds of information depending on the type of the digitizer used. Examples of the signal information is the "true" information of a binary readout chip, the number of ADC counts or the ToT (time-over-threshold).

The exact type of the time and signal values depends on the digitizer module used, for which the module documentation is to be consulted.

Main parameters: - The pixel corresponding to the hit (`getPixel()`) and its index (`getIndex()`) - The related PixelCharge (`getPixelCharge()`) and PixelPulse, if any (`getPixelPulse()`) - The signal of the hit (`getSignal()`) - The time information of the hit in local and global coordinates (`getLocalTime()`, `getGlobalTime()`)

For more details refer to the code reference.

7.1.9 PixelPulse

If the detector's front-end electronics provide a digitized front-end pulse, this object can be used to store that information. It inherits from the Pulse object.

Main parameters: - Everything from Pulse - The pixel corresponding to the digitized pulse (`getPixel()`) and its index (`getIndex()`) - The corresponding pixel charge (`getPixelCharge()`)

For more details refer to the code reference.

7.2 Object History

Objects may carry information about the objects which were used to create them. For example, a `PropagatedCharge` could hold a link to the `DepositedCharge` object at which the propagation started. All objects created during a single simulation event are accessible until the end of the event; more information on object persistency within the framework can be found in Section 4.6.

Object history is implemented using the ROOT TRef class [20], which acts as a special reference. On construction, every object gets a unique identifier assigned, that can be stored in other linked objects. This identifier can be used to retrieve the history, even after the objects are written out to ROOT TTrees [19]. TRef objects are however not automatically fetched and can only be retrieved if their linked objects are available in memory, which has to be ensured explicitly. Outside the framework this means that the relevant tree containing the linked objects should be retrieved and loaded at the same entry as the object that request the history. Whenever the related object is not in memory (either because it is not available or not fetched) a `MissingReferenceException` will be thrown.

A MCTrack which originated from another MCTrack is linked via a reference to this track, this way the track hierarchy can be obtained. Every MCParticle is linked to the MCTrack it is associated with. A MCParticle can furthermore be linked to another MCParticle on the same detector. This will be the case if there are MCParticles from a primary (parent) and secondary (child) track on one detector. The corresponding child MCParticles will then carry a reference to the parent MCParticle.

8 Modules

This section describes all currently available modules in detail. This comprises a description of the physics implemented as well as a list of all configuration parameters along with their default values. For inquiries about a specific module or its documentation, the respective module maintainers should be contacted directly. The modules are listed in alphabetical order.

8.1 CapacitiveTransfer

8.1.1 Description

Similar to the SimpleTransferModule, this module combines individual sets of propagated charges together to a set of charges on the sensor pixels and thus prepares them for processing by the detector front-end electronics. In addition to the SimpleTransferModule, where the charge close to the implants is transferred only to the closest read-out pixel, this module also copies the propagated charge to the neighboring pixels, scaled by the respective cross-coupling (i.e. `cross_capacitance / nominal_capacitance`), in order to simulate the cross-coupling between neighboring pixels in Capacitively Coupled Pixel Detectors (CCPDs).

It is also possible to simulate assemblies with tilted chips, with non-uniform coupling over the pixel matrix, by providing the tilting angles between the chips, the nominal and minimum gaps between the pixel pads, the pixel coordinates where the chips are away from each other by the minimum gap provided and a root file containing ROOT::TGraph with coupling capacitances *vs* gap between pixel pads.

The coupling matrix (imported via the `coupling_matrix` or the `coupling_file` configuration keys) represents the pixels coupling with a nominal gap between the chips, while the ROOT file imported with the configuration key `coupling_scan_file` contains the coupling between the pixels for several gaps.

The coupling matrices can be used to easily simulate the cross-coupling in CCPDs with the nominal, and constant, gap between chips over the pixel matrix. In such cases, the "central pixel" (center element of the coupling matrix) always receive 100% of the charge transferred while neighbor pixels, with lower coupling capacitance, gets a fraction of the charged transferred to the central pixel, normalized by the nominal capacitance (capacitance to central pixel). The coupling matrices always represents the coupling in fractions from 0 (no charge transferred) up to 1 (100% transfer).

If a coupling_scan_file is provided the gap between the chips will be calculated on each pixel with a hit and the charge transferred will be normalized by the capacitance value of the central pixel at the nominal gap. This model will reproduce the results with

the coupling matrices if `chip_angle = 0rad 0rad` (parallel chips) and `minimum_gap = nominal_gap`.

8.1.2 Dependencies

This module requires an installation of Eigen3.

8.1.3 Parameters

- `coupling_scan_file`: Root file containing a TGraph, for each pixel, with the capacitance simulated for each gap between the pixel pads. The TGraph objects in the root file should be named `Pixel_X` where `X` goes from 1 to 9.
- `chip_angle`: Tilt angle between chips. The first angle is the rotation along the columns axis, and second is along the row axis. It defaults to 0.0 radians (parallel chips).
- `tilt_center`: Pixel position for the nominal coupling/distance.
- `nominal_gap`: Nominal gap between chips.
- `minimum_gap`: Closest distance between chips.
- `cross_coupling`: Enables cross-coupling between pixels. Defaults to `true` (enabled).
- `coupling_file`: Path to the file containing the cross-coupling matrix. The file must contain the relative capacitance to the central pixel.
- `coupling_matrix`: Cross-coupling matrix with relative capacitances.
- `max_depth_distance`: Maximum distance in depth, i.e. normal to the sensor surface at the implant side, for a propagated charge to be taken into account. Defaults to 5um.
- `output_plots`: Saves the output plots for this module. Defaults to 1 (enabled).

The cross-coupling matrix, to be parsed via the matrix file or via the configuration file, must be organized in Row vs Col, such as:

```
cross_coupling_00    cross_coupling_01    cross_coupling_02
cross_coupling_10    cross_coupling_11    cross_coupling_12
cross_coupling_20    cross_coupling_21    cross_coupling_22
```

The matrix center element, `cross_coupling_11` in this example, is the coupling to the closest pixel and should be always 1. The matrix can have any size, although square 3x3 matrices are recommended as the coupling decreases significantly after the first neighbors and the simulation will scale with NxM, where N and M are the respective sizes of the matrix.

8.1.4 Usage

This module accepts only one coupling model (`coupling_scan_file`, coupling_file or `coupling_matrix`) at each time. If more then one option is provided, the simulation will not run.

```
[CapacitiveTransfer]
coupling_scan_file = "capacitance_vs_gap.root"
nominal_gap = 2um
minimum_gap = 8um
chip_angle = -0.000524rad 0.000350rad
tilt_center = 80 336
cross_coupling = true
max_depth_distance = 5um
```

or

```
[CapacitiveTransfer]
max_depth_distance = 5um
coupling_file = "capacitance_matrix.txt"
```

or

```
[CapacitiveTransfer]
max_depth_distance = 5um
coupling_matrix = [[0.1, 0.3, 0.1], [0.2, 1, 0.2], [0.1, 0.3, 1.1]]
```

8.2 CorryvreckanWriter

8.2.1 Description

Takes all digitised pixel hits and converts them into Corryvreckan pixel format. These are then written to an output file in the expected format to be read in by the reconstruction software. Will optionally write out the MC Truth information, storing the MC particle class from Corryvreckan. It is noted that the time resolution is hard-coded as `5ns` for all detectors due to time structure of written out events: events of length `5ns`, with a gap of `10ns` in between events.

This module writes output compatible with Corryvreckan 1.0 and later.

8.2.2 Parameters

- `file_name` : Output filename (file extension `.root` will be appended if not present). Defaults to `corryvreckanOutput.root`
- `geometry_file` : Name of the output geometry file in the Corryvreckan format. Defaults to `corryvreckanGeometry.conf`
- `reference`: Name of the detector used as reference in the reconstruction.
- `dut`: List of detector names to be treated as device under test in the reconstruction. Defaults to an empty list.
- `output_mctruth` : Flag to write out MCParticle information for each hit. Defaults to `true`.

- `global_timing`: Flag to select global timing information to be written to the Corryvreckan file. By default, local information is written, i.e. only the local time information from the pixel hit or MCParticle in question. If enabled, the timestamp is set as the event time plus the global time information of the object with respect to the event begin. Defaults to `false`.

8.2.3 Usage

Typical usage is:

```
[CorryvreckanWriter]
file_name = corryvreckan
output_mctruth = true
reference = "telescope_plane0"
```

8.3 CSADigitizer

8.3.1 Description

Digitization module which translates the collected charges into a digitized signal, emulating a charge sensitive amplifier with Krummenacher feedback. For this purpose, a transfer function for a CSA with Krummenacher feedback is taken from [73]:

$$H(s) = \frac{R_f}{(1 + \tau_f s) * (1 + \tau_r s)},$$

with fall time constant

$$\tau_f = R_f C_f$$

and rise time constant

$$\tau_r = \frac{C_{det} * C_{out}}{g_m * C_f}$$

The impulse response function of this transfer function is convoluted with the charge pulse. This module can be steered by either providing all contributions to the transfer function as parameters within the `csa` model, or using a simplified parametrization providing rise time and feedback time. In the latter case, the parameters are used to derive the contributions to the transfer function (see e.g. [74] for calculation of transconductance).

Alternatively a custom impulse response function can be provided by using the `custom` model.

Noise can be applied to the individual bins of the output pulse, drawn from a normal distribution.

The values stored in `PixelHit` depend on the Time-of-Arrival (ToA) and Time-over-Threshold (ToT) settings. If a ToA clock is defined, then `local_time` will be stored in

ToA clock cycles, else in time units. If a ToT clock is defined, then `signal` will be the amount of ToT cycles the pulse is above the threshold, else it will be the integral of the amplified pulse.

Since the input pulse may have different polarity, it is important to set the threshold accordingly to a positive or negative value, otherwise it may not trigger at all. If this behavior is not desired, the `ignore_polarity` parameter can be set to compare only the absolute values of the input and the threshold value.

8.3.2 Parameters

- `model`: Choice between different CSA models. Currently implemented are two parametrizations of the circuit from [73], `simple` and `csa`, and the `custom` model for a custom impulse response.
- `integration_time`: The length of time the amplifier output is registered. Defaults to 500 ns.
- `sigma_noise`: Standard deviation of the Gaussian-distributed noise added to the output signal. Defaults to 0.1 mV.
- `threshold`: Threshold for TOT/TOA logic, for considering the output signal as a hit. Defaults to 10mV.
- `ignore_polarity`: Select whether polarity of the threshold is ignored, i.e. the absolute values are compared, or if polarity is taken into account. Defaults to `false`.
- `clock_bin_toa`: Duration of a clock cycle for the time-of-arrival (ToA) clock. If set, the output timestamp is delivered in units of ToA clock cycles, otherwise in nanoseconds.
- `clock_bin_tot`: Duration of a clock cycle for the time-over-threshold (ToT) clock. If set, the output charge is delivered as time over threshold in units of ToT clock cycles, otherwise the pulse integral is stored instead.

Parameters for the simplified model

- `feedback_capacitance`: The feedback capacity to the amplifier circuit. Defaults to 5e-15 F.
- `rise_time_constant`: Rise time constant of CSA output. Defaults to 1 ns.
- `feedback_time_constant`: Feedback time constant of CSA output. Defaults to 10 ns.

Parameters for the CSA model

- `feedback_capacitance`: The feedback capacity to the amplifier circuit. Defaults to 5e-15 F.
- `krummenacher_current`: The feedback current setting of the CSA. Defaults to 20 nA.
- `detector_capacitance`: The detector capacitance. Defaults to 100 e-15 F.
- `amp_output_capacitance`: The capacitance at the amplifier output. Defaults to 20 e-15 F.

- `transconductance`: The transconductance of the CSA feedback circuit. Defaults to 50e-6 C/s/V.
- `temperature`: Defaults to 293.15K.

Parameters for the custom model

- `response_function`: A 1-dimensional `ROOT::TFormula` [54] expression for the impulse response function.
- `response_parameters`: Array of the parameters in the response function. The number of parameters given need to match up with the number of parameters in the formula.

Plotting parameters

- `output_plots`: Enables simple output histograms to be generated from the data in every step (slows down simulation considerably). Disabled by default.
- `output_plots_scale`: Set the x-axis scale of the output histograms, defaults to 30ke.
- `output_plots_bins`: Set the number of bins for the output histograms, defaults to 100.
- `output_pulsegraphs`: Determines if pulse graphs should be generated for every event. This creates several graphs per event, depending on how many pixels see a signal, and can slow down the simulation. It is not recommended to enable this option for runs with more than a couple of events. Disabled by default.

8.3.3 Usage

Example how to use the `csa` model in this module:

```
[CSADigitizer]
model = "csa"
feedback_capacitance = 10e-15C/V
detector_capacitance = 100e-15C/V
krummenacher_current = 25e-9C/s
amp_output_capacitance = 15e-15C/V
transconductance = 50e-6C/s/V
temperature = 298
integration_time = 0.5e-6s
threshold = 10e-3V
sigma_noise = 0.1e-3V
```

Example for the `simple` model:

```
[CSADigitizer]
model = "simple"
feedback_capacitance = 5e-15C/V
rise_time_constant = 1e-9s
feedback_time_constant = 10e-9 s
```

```
integration_time = 0.5e-6s
threshold = 10e-3V
clock_bin_toa = 1.5625ns
clock_bin_tot = 25.0ns
```

Example for the `custom` model:

```
[CSADigitizer]
model = "custom"
response_function = "TMath::Max([0]*(1.-TMath::Exp(-x/[1]))-[2]*x,0.)"
response_parameters = [2.6e14V/C, 9.1e1ns, 4.2e19V/C/s]
integration_time = 10us
threshold = 60mV
clock_bin_toa = 8ns
clock_bin_tot = 8ns
```

8.4 DatabaseWriter

8.4.1 Description

This module enables writing the simulation output into a postgreSQL database. This is useful when fast I/O between applications is needed (e.g. real time visualization and/or analysis). By default, all object types (MCTrack, MCParticle, DepositedCharge, PropagatedCharge, PixelCharge, PixelHit) are written. However, it should be kept in mind that PropagatedCharge and DepositedCharge data will slow down the simulation significantly and will lead to a large database. Unless really required for the analysis of the simulation, it is recommended to exclude these objects. This can be accomplished by using the `include` and `exclude` parameters in the configuration file. In order to use this module, one is required to install PostgreSQL and generate a database using the `create-db.sql` script in `/etc/scripts`. On Linux, this can be done as

```
sudo -u postgres psql
postgres: CREATE DATABASE mydb;
postgres: \q
sudo -u postgres psql mydb
postgres: \i etc/scripts/create-db.sql
```

This generates a database with the following structure:

```
 Schema |                    Name                     |   Type   |  Owner
--------+---------------------------------------------+----------+----------
 public | depositedcharge                             | table    | postgres
 public | depositedcharge_depositedcharge_nr_seq      | sequence | postgres
 public | event                                       | table    | postgres
 public | event_event_nr_seq                          | sequence | postgres
 public | mcparticle                                  | table    | postgres
 public | mcparticle_mcparticle_nr_seq                | sequence | postgres
 public | mctrack                                     | table    | postgres
 public | mctrack_mctrack_nr_seq                      | sequence | postgres
```

```
 public | pixelcharge                                | table    | postgres
 public | pixelcharge_pixelcharge_nr_seq             | sequence | postgres
 public | pixelhit                                   | table    | postgres
 public | pixelhit_pixelhit_nr_seq                   | sequence | postgres
 public | propagatedcharge                           | table    | postgres
 public | propagatedcharge_propagatedcharge_nr_seq | sequence | postgres
 public | run                                        | table    | postgres
 public | run_run_nr_seq                             | sequence | postgres
(16 rows)
```

Host, username and password are required to write into the database. A new user/password pair can be created and relevant privileges to edit the database can be created via

```
sudo -u postgres createuser myuser
sudo -u postgres psql mydb
postgres: CREATE USER myuser WITH ENCRYPTED PASSWORD 'mypass';
postgres: GRANT ALL PRIVILEGES ON DATABASE mydb TO myuser;
postgres: GRANT SELECT, INSERT, UPDATE, DELETE ON ALL TABLES IN SCHEMA
 ↪  public TO myuser;
postgres: GRANT SELECT, USAGE ON ALL SEQUENCES IN SCHEMA public TO
 ↪  myuser;
```

In case of an authentication failure error being issues, the password of the user can be changed using

```
sudo -u postgres psql -c "ALTER USER myuser PASSWORD 'mypass';"
```

The database is structured so that the data are referenced according to the sequence

```
MCTrack -> MCParticle -> DepositedCharge -> PropagatedCharge ->
 ↪  PixelCharge -> PixelHit
```

This allows for the full reconstruction of the MC truth when retrieving information out of the database. When one of the objects is excluded, the corresponding reference is obviously lost and the chain is broken. The only exception to this chain rule is the direct reference MCParticle -> PixelHit. By default, each module always refers to the run and event numbers. As an example, the following is the table corresponding to the PixelHit objects for a single run of four events:

```
mydb: SELECT * FROM pixelhit;
 pixelhit_nr | run_nr | event_nr | mcparticle_nr | pixelcharge_nr |
  ↪  detector  | x | y | signal  | hittime
-------------+--------+----------+---------------+----------------+-----------+---+---
           1 |      1 |        1 |             2 |              2 |
            ↪  detector1 | 2 | 2 | 46447.9 |       0
           2 |      1 |        1 |             2 |              2 |
            ↪  detector2 | 2 | 2 | 34847.5 |       0
           3 |      1 |        2 |             4 |              4 |
            ↪  detector1 | 2 | 2 | 27788.1 |       0
           4 |      1 |        2 |             4 |              4 |
            ↪  detector2 | 2 | 2 | 38011.6 |       0
```

8.4.2 Parameters

- `host`: Host address on which the database server runs, can be an IP address or host name. Mandatory parameter.
- `port`: Port the database server listens on. Mandatory parameter.
- `database_name`: Name of the database to store data in. The database needs to exist and has to be created before starting the simulation. Mandatory parameter.
- `user`: User name of the SQL user with access rights to the relevant database. mandatory parameter.
- `password`: Password of the user account with database write access. Mandatory parameter.
- `run_id`: Arbitrary run identifier assigned to this simulation in the database. This parameter is a string and defaults to `none`.
- `include`: Array of object names (without `allpix::` prefix) to write to the ROOT trees, all other object names are ignored (cannot be used together simultaneously with the *exclude* parameter).
- `exclude`: Array of object names (without `allpix::` prefix) that are not written to the ROOT trees (cannot be used together simultaneously with the *include* parameter).
- `global_timing`: Flag to select global timing information to be written to the database. By default, local information is written, i.e. only the local time information from the pixel hit in question. If enabled, the timestamp is set as the global time information of the object with respect to the event begin. Defaults to `false`.
- `require_sequence`: Boolean flag to select whether events have to be written in sequential order or can be stored in the order of processing. Defaults to `false`, writing events immediately. If strict adherence to the order of events is required, finished events are buffered until they can be written to the database. Since in this case database access happens single-threaded, this might impact the performance of the simulation.

8.4.3 Usage

To write objects excluding `PropagatedCharge` and `DepositedCharge` to a PostgreSQL database running on `localhost` with user `myuser`, the following configuration can be placed at the end of the main configuration:

```
[DatabaseWriter]
exclude = PropagatedCharge, DepositedCharge
host = "localhost"
port = 5432
database_name = "mydb"
user = "myuser"
password = "mypass"
run_id = "myRun"
```

Optionally the password can also be provided via the command line only, using `allpix -c config.conf -o DatabaseWriter.password="mypass"`.

8.5 DefaultDigitizer

8.5.1 Description

Simple digitization module which translates the collected charges into a digitized signal proportional to the input charge. It simulates noise contributions from the readout electronics as Gaussian noise and allows for a configurable threshold. Furthermore, the linear response of an QDC as well as a TDC with configurable resolution can be simulated. For maximum simplicity only the absolute of the charge is used and compared to a positive threshold.

In detail, the following steps are performed for every pixel charge:

- A Gaussian noise is added to the input charge value in order to simulate input noise to the preamplifier circuit.
- The preamplifier is simulated by applying a gain function to the input charge, or by multiplying the input charge with a defined gain factor.
- An optional simplistic front-end saturation can be simulated which replaces the measured pixel charge with a value drawn from a Gaussian distribution with the configured saturation mean and width if the charge measured is larger than the calculated saturation value. This follows the approach taken in [75]. The pixel charge is compared to the smeared saturation value in order to generate a smooth transition rather than an edge in the spectrum.
- A charge threshold is applied. Only if the threshold is surpassed, the pixel is accounted for - for all values below the threshold, the pixel charge is discarded. The actually applied threshold is smeared with a Gaussian distribution on an event-by-event basis allowing for simulating fluctuations of the threshold level. It should be noted that only positive threshold values are possible, and that this threshold will be compared to the absolute of the charge. This therefore both works for positive and negative inputs.
- A charge-to-digital converter (QDC) with configurable resolution, given in bit, can be simulated. For this, first an inaccuracy of the QDC is simulated using an additional Gaussian smearing which allows to take QDC noise into account. Then, the charge is converted into QDC units using the `qdc_slope` and `qdc_offset` parameters provided. Finally, the calculated value is clamped to be contained within the QDC resolution, over- and underflows are treated as saturation. The QDC implementation also allows to simulate ToT (time-over-threshold) devices by setting the `qdc_offset` parameter to the negative `threshold`. Then, the QDC only converts charge above threshold.
- A time-to-digital converter (TDC) with configurable resolution, given in bit, can be simulated if pulse information is available from the input data. If the necessary pulse information is available from the input data, e.g. by using the PulseTransfer module to generate PixelCharge objects, this module calculates the time-of-arrival (ToA) as the time when the integrated input charge crosses the threshold. Also here, the absolute of the integrated charge is compared to a positive threshold value to be independent of the signal polarity. First, the time from the start of the event until the first crossing of the charge threshold is calculated. It should be noted that this calculation does not take into account charge noise simulated in the QDC. The resulting ToA is smeared with a Gaussian distribution which allows to take TDC fluctuations into account. Then, the ToA is converted into TDC units

using the `tdc_slope` and `tdc_offset` parameters provided. Finally, the calculated value is clamped to be contained within the TDC resolution, over- and underflows are treated as saturation. If no time information is available from the input data, a local time stamp of 0 is stored. It should be noted that when using the TDC simulation, the local time stamp of the produced PixelHit object is provided in TDC bins rather than in nanoseconds of the framework-internal units. The global timestamp, however, is always provided in nanoseconds and independent of the TDC settings.

Gain Function

Apart from a linear gain configured via the `gain` parameter, this module also supports arbitrary gain/response functions, defined via the `gain_function` parameter, which depends on the input charge. In order to use the input charge in the formula, an `x` has to be placed at the respective position.

Parameters of the function can either be placed directly in the formula in framework-internal units, or provided separately as arrays via the `gain_parameters` parameter. Placeholders for parameters in the formula are denoted with squared brackets and a parameter number, for example `[0]` for the first parameter provided. Parameters specified separately from the formula can contain units which will be interpreted automatically - parameters directly placed in the mobility formula have to be supplied in framework-internal units since the function will be evaluated in internal units. It is recommended to use the possibility of separately configuring the parameters and to make use of units to avoid conversion mistakes.

As an example, the following configuration implements a surrogate response function, i.e.

$$f(q) = a \cdot q + b - \frac{c}{q - t}$$

```
gain_function = "[0]*x + [1] - [2] / (x - [3])"
gain_parameters = 1.09, 5.8e, 130.5e*e, 20.2e
```

Output Plots

With the `output_plots` parameter activated, the module produces histograms of the charge distribution at the different stages of the simulation, i.e. before processing, with electronics noise, after threshold selection, and with ADC smearing applied. A 2D-histogram of the actual pixel charge in electrons and the converted charge in QDC units is provided if QDC simulation is enabled by setting `qdc_resolution` to a value different from zero. In addition, the distribution of the actually applied threshold is provided as histogram.

8.5.2 Parameters

- `electronics_noise` : Standard deviation of the Gaussian noise in the electronics (before amplification and application of the threshold). Defaults to 110 electrons.
- `gain` : Gain factor the input charge is multiplied with, defaults to 1.0 (no gain) if no gain function is supplied. `gain` and `gain_function` are mutually exclusive.
- `gain_function` : Formula describing the gain as a function of the input charge. `gain` and `gain_function` are mutually exclusive.
- `gain_parameters` : Parameters of the gain formula. This parameter needs to be provided as array of values, physical units are supported for each parameter individually.
- `saturation`: Enable front-end saturation simulation. Defaults to `false`.
- `saturation_mean`: Mean of the simulated front-end saturation charge, defaults to `190ke`. Only used if `saturation` is `true`.
- `saturation_width`: Width of the Gaussian distribution used to calculate the new charge value of the simulated front-end saturation, defaults to `20ke`. Only used if `saturation` is `true`.
- `threshold` : Threshold for considering the collected charge as a hit. Defaults to 600 electrons.
- `threshold_smearing` : Standard deviation of the Gaussian uncertainty in the threshold charge value. Defaults to 30 electrons.
- `qdc_resolution` : Resolution of the QDC in units of bits. Thus, a value of 8 would translate to a QDC range of 0 – 255. A value of 0bit switches off the QDC simulation and returns the actual charge in electrons. Defaults to 0.
- `qdc_smearing` : Standard deviation of the Gaussian noise in the ADC conversion (after applying the threshold). Defaults to 300 electrons.
- `qdc_slope` : Slope of the QDC calibration in electrons per ADC unit (unit: “e”). Defaults to 10e.
- `qdc_offset` : Offset of the QDC calibration in electrons. In order to simulate a ToT (time-over-threshold) device, this offset should be configured to the negative value of the threshold. Defaults to 0.
- `allow_zero_qdc`: Allows the QDC to return a value of zero if enabled, otherwise the minimum value returned is one. Defaults to `false`. When enabled special care should be taken when analyzing data since charge-weighted cluster position interpolation might return unexpected results.
- `tdc_resolution` : Resolution of the TDC in units of bits. Thus, a value of 8 would translate to a TDC range of 0 – 255. A value of 0bit switches off the TDC simulation and returns the actual time of arrival in nanoseconds. Defaults to 0.
- `tdc_smearing` : Standard deviation of the Gaussian noise in the TDC conversion. Defaults to 50 ps.
- `tdc_slope` : Slope of the TDC calibration in nanoseconds per TDC unit (unit: “ns”). Defaults to 10ns.
- `tdc_offset` : Offset of the TDC calibration in nanoseconds. Defaults to 0.
- `allow_zero_tdc`: Allows the TDC to return a value of zero if enabled, otherwise the minimum value returned is one. Defaults to `false`.
- `output_plots` : Enables output histograms to be generated from the data in every step (slows down simulation considerably). Disabled by default.
- `output_plots_scale` : Set the x-axis scale of charge-related output plot, defaults to 30ke.

- `output_plots_timescale` : Set the x-axis scale of time-related output plot, defaults to 300ns.
- `output_plots_bins` : Set the number of bins for the output plot histograms, defaults to 100.

8.5.3 Usage

The default configuration is equal to the following:

```
[DefaultDigitizer]
electronics_noise = 110e
threshold = 600e
threshold_smearing = 30e
qdc_smearing = 300e
```

8.6 DepositionCosmics

8.6.1 Description

This module simulates cosmic ray particle shower distributions and their energy deposition in all sensors of the setup. The cosmic ray particle showers are simulated using the Cosmic-ray shower generator (CRY) [76], the generated particles are transported through the setup by Geant4. More detailed information about CRY can be found in its physics description [77] and user manual [78].

This module inherits functionality from the *DepositionGeant4* module and several of its parameters have their origin there. A detailed description of these configuration parameters can be found in the respective module documentation. The parameter `number_of_particles` here refers to full shower developments instead of individual particles, there can be multiple particles per shower. The number of electron/hole pairs created by a given energy deposition is calculated using the mean pair creation energy [31], fluctuations are modeled using a Fano factor assuming Gaussian statistics [32]. Default values of both parameters for different sensor materials are included and automatically selected for each of the detectors. A full list of supported materials can be found elsewhere in the manual. These can be overwritten by specifying the parameters `charge_creation_energy` and `fano_factor` in the configuration.

The coordinate system for this module defines the `z` axis orthogonal to the earth surface, pointing upwards. This means shower particles travel along the negative `z` axis and all detectors should be placed below the incidence plane at `z = 0`. The area on which incident particles will be simulated is automatically inferred from the total setup size, and the next larger set of tabulated data available is selected. Data are tabulated for areas of 1m, 3m, 10m, 30m, 100m, and 300m. Particles outside the selected window are dropped.

The first shower particle arriving in the event has a timestamp of `0ns`, all subsequent particles of the same shower have the appropriate spacing in time. It should be noted that the time difference between the arrival of different particles of the same shower

can amount up to hundreds of microseconds. If this behavior is not desired, all particle timestamps can be forced to `0ns` by enabling the `reset_particle_time` switch.

The total time elapsed in the CRY simulation for the given number of showers is stored in the module configuration under the key `total_time_simulated`. If the ROOTObjectWriter is used to store the simulation result, this value is available from the output file. In other cases, the value can be obtained from the log output of the run.

8.6.2 Dependencies

This module inherits from and therefore requires the *DepositionGeant4* module as well as an installation Geant4.

8.6.3 Parameters

- `data_path`: Directory to read the tabulated input data for the CRY framework from. By default, this is the standard installation path of the data files shipped with the framework.
- `reset_particle_time`: Boolean to force resetting all particle timestamps to `0ns`, even from different particles from the same shower. Defaults to `false`, i.e. the first particle of a shower bears a timestamp of `0ns` and all subsequent particles retain their time difference to the first one.

Relevant parameters inherited from *DepositionGeant4*

- `physics_list`: Geant4-internal list of physical processes to simulate, defaults to FTFP_BERT_LIV. More information about possible physics list and recommendations for defaults are available on the Geant4 website [79].
- `enable_pai`: Determines if the Photoabsorption Ionization model is enabled in the sensors of all detectors. Defaults to false.
- `pai_model`: Model can be **pai** for the normal Photoabsorption Ionization model or **paiphoton** for the photon model. Default is **pai**. Only used if *enable_pai* is set to true.
- `charge_creation_energy` : Energy needed to create a charge deposit. Defaults to the energy needed to create an electron-hole pair in the respective sensor material (e.g. 3.64 eV for silicon sensors, [31]). A full list of supported materials can be found elsewhere in the manual.
- `fano_factor`: Fano factor to calculate fluctuations in the number of electron/hole pairs produced by a given energy deposition. Defaults are provided for different sensor materials, e.g. a value of 0.115 for silicon [32]. A full list of supported materials can be found elsewhere in the manual.
- `max_step_length` : Maximum length of a simulation step in every sensitive device. Defaults to 1um.
- `range_cut` : Geant4 range cut-off threshold for the production of gammas, electrons and positrons to avoid infrared divergence. Defaults to a fifth of the shortest pixel feature, i.e. either pitch or thickness.

- `cutoff_time` : Maximum lifetime of particles to be propagated in the simulation. This setting is passed to Geant4 as user limit and assigned to all sensitive volumes. Particles and decay products are only propagated and decayed up the this time limit and all remaining kinetic energy is deposited in the sensor it reached the time limit in. Defaults to 221s (to ensure proper gamma creation for the Cs137 decay). Note: Neutrons have a lifetime of 882 seconds and will not be propagated in the simulation with the default `cutoff_time`.
- `number_of_particles` : Number of cosmic ray showers to generate in a single event. Defaults to one.
- `output_plots` : Enables output histograms to be generated from the data in every step (slows down simulation considerably). Disabled by default.
- `output_plots_scale` : Set the x-axis scale of the output plot, defaults to 100ke.

CRY Framework Parameters

- `latitude`: Latitude for which the incident particles from cosmic ray showers should be simulated. Should be between `90.0` (north pole) and `-90.0` (south pole). Defaults to `53.0` (DESY).
- `date`: Date for the simulation to account for the 11-year cycle of solar activity and related change in cosmic ray flux. Should be given as string in the form `month-day-year` and defaults to the last day of 2020, i.e. `12-31-2020`.
- `return_neutrons`: Boolean to select whether neutrons should be returned to Geant4. Defaults to `true`.
- `return_protons`: Boolean to select whether protons should be returned to Geant4. Defaults to `true`.
- `return_gammas`: Boolean to select whether gammas should be returned to Geant4. Defaults to `true`.
- `return_electrons`: Boolean to select whether electrons should be returned to Geant4. Defaults to `true`.
- `return_muons`: Boolean to select whether muons should be returned to Geant4. Defaults to `true`.
- `return_pions`: Boolean to select whether pions should be returned to Geant4. Defaults to `true`.
- `return_kaons`: Boolean to select whether kaons should be returned to Geant4. Defaults to `true`.
- `altitude`: Altitude for which the shower particles should be simulated. Possible values are `0m`, `2100m` and `11300m`, defaults to sea level, i.e. `0m`. It should be noted that the particle incidence plane is always located at `z = 0` independent of the simulated altitude.
- `min_particles`: Minimum number of particles required for a shower to be considered. Defaults to `1`.
- `max_particles`: Maximum number of particles in a shower before additional particles are cut off. Defaults to `100000`
- `area`: Side length of the squared area for which incident particles are simulated. This can maximally be `300m`. By default, the maximum size is automatically derived from the dimensions of the detector setup of the current simulation.

8.6.4 Usage

```
[DepositionCosmics]
physics_list = FTFP_BERT_LIV
number_of_particles = 2
max_step_length = 10.0um
return_kaons = false
altitude = 0m
```

8.6.5 Licenses

CRY is published under a 3-Clause BSD-like license, which is available in the file `cry/COPYRIGHT.TXT`. The original software can be obtained from `https://nuclear.llnl.gov/simulation/`.

8.7 DepositionGeant4

8.7.1 Description

Module which deposits charge carriers in the active volume of all detectors. It acts as wrapper around the Geant4 logic and depends on the global geometry constructed by the GeometryBuilderGeant4 module. It initializes the physical processes to simulate a particle source that will deposit charge carriers for every event simulated. The number of electron/hole pairs created by a given energy deposition is calculated using the mean pair creation energy [31], fluctuations are modeled using a Fano factor assuming Gaussian statistics [32]. Default values of both parameters for different sensor materials are included and automatically selected for each of the detectors. A full list of supported materials can be found elsewhere in the manual. These can be overwritten by specifying the parameters `charge_creation_energy` and `fano_factor` in the configuration.

Source Shapes

The source can be defined in two different ways using the `source_type` parameter: with pre-defined shapes or with a Geant4 macro file. Pre-defined shapes are point, beam, square or sphere. For the square and sphere, the particle starting points are distributed homogeneously over the surfaces. By default, the particle directions for the square are random, as would be for a squared radioactive source. For the sphere, unless a focus point is set, the particle directions follow the cosine-law defined by Geant4 [80] and the field inside the sphere is hence isotropic.

To define more complex sources or angular distributions, the user can create a macro file with Geant4 commands. These commands are those defined for the GPS source and are explained in the Geant4 website [80]. In order to avoid collisions with internal configurations, the command `/gps/number` should be replaced by the configuration parameter `number_of_particles` in this module in order to correctly execute the Geant4 event loop.

All source positions defined in the macro via the commands `/gps/position` and `/gps/pos/centre` are used to automatically extend the Geant4 world volume to always include the sources.

Particles, Ions and Radioactive Decays

The particle type can be set via a string (`particle_type`) or by the respective PDG code (`particle_code`). Refer to the Geant4 webpage [81] for information about the available types of particles and the PDG particle code definition [72] for a list of the available particles and PDG codes.

Radioactive sources can be simulated simply by setting their isotope name in the `particle_type` parameter, and optionally by setting the source energy to zero for a decay in rest. The `G4RadioactiveDecay` package [82] is used to simulate the decay of the radioactive nuclei. Secondary ions from the decay are not further treated and the decay chain is interrupted, e.g. Am241 only undergoes its alpha decay without the decay of its daughter nucleus of Np237 being simulated. The full decay chain can be simulated if the `cutoff_time` is set to the appropriate value for this chain. Radioactive isotopes are forced to decay immediately in order to allow sensible measurements of arrival and deposition times. Currently, the following radioactive isotopes are supported: `Fe55`, `Am241`, `Sr90`, `Co60`, `Cs137`. Note that for `Cs137` the `cutoff_time` has to be set to 221 seconds for the decay to work properly.

Ions can be used as particles by setting their atomic properties, i.e. the atomic number Z, the atomic mass A, their charge Q, the excitation energy E and whether or not they should decay instantly via the following syntax:

```
particle_type = "ion/Z/A/Q/E/D"
```

where `Z` and `A` are unsigned integers, `Q` is a signed integer, `E` a floating point value with units, e.g. `13eV`, and `D` is `true` for instant decay or `false` else.

Energy Deposition and Charge Carrier creation

For all particles passing the sensitive device of the detectors, the energy loss is converted into deposited charge carriers in every step of the Geant4 simulation. Optionally, the Photoabsorption Ionization model (PAI) can be activated to improve the modeling of thin sensors [83]. The information about the truth particle passage is also fully available, with every deposit linked to a MCParticle. Each trajectory which passes through at least one detector is also registered and stored as a global MCTrack. MCParticles are linked to their respective tracks and each track is linked to its parent track, if available.

A range cut-off threshold for the production of gammas, electrons and positrons is necessary to avoid infrared divergence. By default, Geant4 sets this value to 700um or even 1mm, which is most likely too coarse for precise detector simulation. In this module, the range cut-off is automatically calculated as a fifth of the minimal feature size of a single pixel, i.e. either to a fifth of the smallest pitch of a fifth of the sensor thickness, if smaller. This behavior can be overwritten by explicitly specifying the range cut via the `range_cut` parameter. The propagation of any particle is stopped at the value of

the parameter `cutoff_time`. In case the particle is stopped in a sensitive volume, the remaining kinetic energy is deposited in this sensor.

The module supports the propagation of charged particles in a magnetic field if defined via the MagneticFieldReader module.

With the `output_plots` parameter activated, the module produces histograms of the total deposited charge per event for every sensor in units of kilo-electrons. The scale of the plot axis can be adjusted using the `output_plots_scale` parameter and defaults to a maximum of 100ke.

8.7.2 Dependencies

This module requires an installation Geant4.

8.7.3 Parameters

- `physics_list`: Geant4-internal list of physical processes to simulate, defaults to FTFP_BERT_LIV. More information about possible physics list and recommendations for defaults are available on the Geant4 website [79].
- `enable_pai`: Determines if the Photoabsorption Ionization model is enabled in the sensors of all detectors. Defaults to false.
- `pai_model`: Model can be **pai** for the normal Photoabsorption Ionization model or **paiphoton** for the photon model. Default is **pai**. Only used if *enable_pai* is set to true.
- `charge_creation_energy` : Energy needed to create a charge deposit. Defaults to the energy needed to create an electron-hole pair in the respective sensor material (e.g. 3.64 eV for silicon sensors, [31]). A full list of supported materials can be found elsewhere in the manual.
- `fano_factor`: Fano factor to calculate fluctuations in the number of electron/hole pairs produced by a given energy deposition. Defaults are provided for different sensor materials, e.g. a value of 0.115 for silicon [32]. A full list of supported materials can be found elsewhere in the manual.
- `max_step_length` : Maximum length of a simulation step in every sensitive device. Defaults to 1um.
- `range_cut` : Geant4 range cut-off threshold for the production of gammas, electrons and positrons to avoid infrared divergence. Defaults to a fifth of the shortest pixel feature, i.e. either pitch or thickness.
- `particle_type` : Type of the Geant4 particle to use in the source (string). Refer to the Geant4 documentation [81] for information about the available types of particles.
- `particle_code` : PDG code of the Geant4 particle to use in the source.
- `source_energy` : Mean kinetic energy of the generated particles.
- `source_energy_spread` : Energy spread of the source.
- `source_position` : Position of the particle source in the world geometry.
- `source_type` : Shape of the source: **beam** (default), **point**, **square**, **sphere**, **macro**.

- `cutoff_time` : Maximum lifetime of particles to be propagated in the simulation. This setting is passed to Geant4 as user limit and assigned to all sensitive volumes. Particles and decay products are only propagated and decayed up the this time limit and all remaining kinetic energy is deposited in the sensor it reached the time limit in. Defaults to 221s (to ensure proper gamma creation for the Cs137 decay). Note: Neutrons have a lifetime of 882 seconds and will not be propagated in the simulation with the default `cutoff_time`.
- `record_all_tracks` : Switch to enable the recording of all Geant4 tracks in the event. By default, this parameter is set to `false` and MCTrack objects are only generated for particles interacting with sensor material, not those that never interact with any detector.
- `geant4_tracking_verbosity` : Verbosity level for Geant4 tracking, defaults to `0`. Higher levels mean more output. It should be noted that the respective log output is redirected to the logging level set via the `log_level_g4cout` parameter in the *GeometryBuilderGeant4* module.
- `number_of_particles` : Number of particles to generate in a single event. Defaults to one particle.
- `deposit_in_frontside_implants` : Boolean to select whether charge carriers should be generated in frontside implants. Defaults to `true`.
- `deposit_in_backside_implants` : Boolean to select whether charge carriers should be generated in backside implants. Defaults to `false`.
- `output_plots` : Enables output histograms to be generated from the data in every step (slows down simulation considerably). Disabled by default.
- `output_plots_scale` : Set the x-axis scale of the output plot, defaults to 100ke.

Parameters for source `beam`

- `beam_size` : Width of the Gaussian beam profile.
- `beam_divergence` : Standard deviation of the particle angles in x and y from the particle beam
- `beam_direction` : Direction of the beam as a unit vector.
- `flat_beam` : Boolean to change your Gaussian beam profile to a flat beam profile. If true, the `beam_size` gives the radius of the beam profile. Defaults to false.

Parameters for source `square`

- `square_side` : Length of the square side.
- `square_angle` : Cone opening angle defining the maximum submission angle. Defaults to `180deg`, i.e. emission into one full hemisphere.

Parameters for source `sphere`

- `sphere_radius` : Radius of the sphere source (particles start only from the surface).
- `sphere_focus_point` : Focus point of the sphere source. If not specified, the radiation field is isotropic inside the sphere.

Parameters for source `macro`

- `file_name` : Path to the Geant4 source macro file.

Note for Developers

This module is used as base for other deposition modules using Geant4 for particle tracking, e.g. DepositionCosmics or DepositionGenerator. Since some of these modules might have a sequence requirement for event processing, this module is a `SequentialModule` but waives the sequence requirement in its constructor. Any derived module that requires a strict sequence has to call `waive_sequence_requirement(false)` in its constructor to overwrite this setting.

8.7.4 Usage

A possible default configuration to use, simulating a beam of 120 GeV pions with a divergence in x, is the following:

```
[DepositionGeant4]
physics_list = FTFP_BERT_LIV
particle_type = "pi+"
source_energy = 120GeV
source_position = 0 0 -1mm
source_type = "beam"
beam_direction = 0 0 1
beam_divergence = 3mrad 0mrad
number_of_particles = 1
```

A radioactive point source of Iron-55 could be simulated by the following configuration:

```
[DepositionGeant4]
physics_list = FTFP_BERT_LIV
particle_type = "Fe55"
source_energy = 0eV
source_position = 0 0 -1mm
source_type = "point"
number_of_particles = 1
```

A Xenon-132 ion beam could be simulated using the following configuration:

```
[DepositionGeant4]
physics_list = FTFP_BERT_LIV
particle_type = "ion/54/132/0/0eV/false"
source_energy = 10MeV
source_position = 0 0 -1mm
source_type = "beam"
beam_direction = 0 0 1
number_of_particles = 1
```

8.8 DepositionGenerator

8.8.1 Description

This module allows to read primary particles produced by Monte Carlo event generators from files in different data formats, and to emit them to a Geant4 `ParticleGun`. The particles are then tracked through the setup using Geant4, and the resulting energy deposits are converted to `DepositedCharge` objects and dispatched to the subsequent simulation chain. The different file formats can be selected via the `model` parameter, the path to the data file has to be provided via the `file_name` configuration parameter.

Events are read consecutively from the generator event data and event number are matched. This means that the event with number 5 in Allpix Squared will contain the data from event number 5 of the generator data file. If events are missing in the generator data, no primary particles are generated in Allpix Squared and the event remains empty.

This module inherits functionality from the *DepositionGeant4* module and several of its parameters have their origin there. A detailed description of these configuration parameters can be found in the respective module documentation. The number of electron/hole pairs created by a given energy deposition is calculated using the mean pair creation energy 31, fluctuations are modeled using a Fano factor assuming Gaussian statistics 32. Default values of both parameters for different sensor materials are included and automatically selected for each of the detectors. A full list of supported materials can be found elsewhere in the manual. These can be overwritten by specifying the parameters `charge_creation_energy` and `fano_factor` in the configuration.

8.8.2 Dependencies

This module inherits from and therefore requires the *DepositionGeant4* module as well as an installation of Geant4. In addition, an installation of the HepMC3 library is required for the module to support the formats `HepMC3`, `HepMC2`, `HepMC ROOTIO` as well as `HepMC ROOTIO TTree`.

8.8.3 Parameters

- `model`: Input data model. Currently supported is the data format of the 84 Monte Carl generator (`GENIE`) as well as the `HepMC3`, `HepMC2`, `HepMCROOT`, `HepMCTTree` data formats written by the HepMC3 library 85.
- `file_name`: Path to the input data file to be read.

Relevant parameters inherited from *DepositionGeant4*

- `physics_list`: Geant4-internal list of physical processes to simulate, defaults to FTFP_BERT_LIV. More information about possible physics list and recommendations for defaults are available on the Geant4 website 79.
- `enable_pai`: Determines if the Photoabsorption Ionization model is enabled in the sensors of all detectors. Defaults to false.

- `pai_model`: Model can be **pai** for the normal Photoabsorption Ionization model or **paiphoton** for the photon model. Default is **pai**. Only used if *enable_pai* is set to true.
- `charge_creation_energy` : Energy needed to create a charge deposit. Defaults to the energy needed to create an electron-hole pair in the respective sensor material (e.g. 3.64 eV for silicon sensors, 31). A full list of supported materials can be found elsewhere in the manual.
- `fano_factor`: Fano factor to calculate fluctuations in the number of electron/hole pairs produced by a given energy deposition. Defaults are provided for different sensor materials, e.g. a value of 0.115 for silicon 32. A full list of supported materials can be found elsewhere in the manual.
- `max_step_length` : Maximum length of a simulation step in every sensitive device. Defaults to 1um.
- `range_cut` : Geant4 range cut-off threshold for the production of gammas, electrons and positrons to avoid infrared divergence. Defaults to a fifth of the shortest pixel feature, i.e. either pitch or thickness.
- `cutoff_time` : Maximum lifetime of particles to be propagated in the simulation. This setting is passed to Geant4 as user limit and assigned to all sensitive volumes. Particles and decay products are only propagated and decayed up the this time limit and all remaining kinetic energy is deposited in the sensor it reached the time limit in. Defaults to 221s (to ensure proper gamma creation for the Cs137 decay). Note: Neutrons have a lifetime of 882 seconds and will not be propagated in the simulation with the default `cutoff_time`.
- `output_plots` : Enables output histograms to be generated from the data in every step (slows down simulation considerably). Disabled by default.
- `output_plots_scale` : Set the x-axis scale of the output plot, defaults to 100ke.

8.8.4 Usage

```
[DepositionGenerator]
physics_list = FTFP_BERT_LIV
max_step_length = 10.0um
model = "GENIE"
file_name = "genie_input_data.root"
```

8.9 DepositionLaser

8.9.1 Description

This deposition generator is mostly intended for simulations of laser-TCT experiments. It generates charge, deposited by absorption of a laser pulse in detector bulk. This module is not dependent on Geant4. Instead, it implements tracking algorithms and simulations of corresponding physical phenomena by its own, using internal Allpix geometry.

Current implementation assumes that the laser pulse is a bunch of point-like photons, each traveling in a straight line. A lookup table [86] is used to determine absorption and refraction coefficients in silicon for a given wavelength. The only supported sensor material is silicon, unless optical properties are explicitly set (see *Parameters*).

Multiple photons are produced in *one* event, thus a *single* event models a *single* laser pulse.

Tracking features:

- photons are absorbed in detector bulk on physically correct depth
- each photon is assumed to create exactly one e-h pair
- photons refract on silicon-air interface
- tracks are terminated when a photon leaves *first encountered* sensitive volume
- tracks are terminated if a passive object is hit (the only supported passive object type is `box`) Verbose information on tracking for each photon is printed if this module is run with `DEBUG` logging level.

Initial direction and starting timestamp for every photon in the bunch are generated to mimic spatial and temporal distributions of delivered intensity of a real laser pulse.

Two options for beam geometry are currently available: `cylindrical` and `converging`. For both options, transversal beam profiles will have a gaussian shape. For a `cylindrical` beam, all tracks are parallel to the set beam direction. For a `converging` beam, track directions would have isotropic distribution (but with a limit on a max angle between the track and the set beam direction).

NB: convention on global time zero for this module contradicts the general convention of the Allpix Squared. For this module, global t=0 is chosen in such a way that the mean value of temporal distribution is *always* positioned at *4 standard deviations* w.r.t. the global t=0. Thus, there is not necessarily a particle that is created exactly when the global time starts. Although, the following Allpix Squared conventions still apply:

- No particles have a negative timestamp.
- Local time zero for each detector is a moment when the first particle that creates a hit in this detectors enters its bulk.

As a result, this module yields `DepositedCharge` instances for each detector, with them having physically correct spatial and temporal distribution.

8.9.2 Parameters

- `number_of_photons`: number of incident photons, generated in *one* event. Defaults to 10000. The total deposited charge will also depend on wavelength and geometry.
- `group_photons`: if specified, incident photons will be grouped in buckets of given size, decreasing amount of `DepositedCharge` instances (but keeping total amount of deposited charge the same), thus reducing load on the propagation module.
- `wavelength` of the laser. If specified, it is used to retrieve sensor optical properties from the lookup table (data is available for the range of 250 – 1450 nm). The only supported material is silicon.
- `data_path`: Directory to read the tabulated input data for the absorption on silicon. By default, this is the standard installation path of the data files shipped with the framework.
- `absorption_length` and `refractive_index`: if both are specified, given values are used instead of the lookup table. This also allows use of sensor materials other than silicon.

- `pulse_duration`: gaussian width of pulse temporal profile. Defaults to 0.5 ns.
- `source_position`: a 3D position vector.
- `beam_direction`: a 3D direction vector.
- `beam_geometry`: either `cylindrical` or `converging`
- `beam_waist`: standard deviation of transversal beam intensity distribution at focus. Defaults to 20 um.
- `focal_distance`: needs to be specified for `converging` beam. This distance is *as it would be in air*. In silicon, beam shape will effectively stretch along its direction due to refraction and the actual focus will be further away from the source.
- `beam_convergence_angle`: max angle between tracks and `beam_direction`. Needs to be specified for a `converging` beam.
- `output_plots`: if set `true`, this module will produce histograms to monitor beam shape and also 3D distributions of charges, deposited in each detector. Histograms would look sensible even for one-event runs. Defaults to `false`.

8.9.3 Usage

A simulation pipeline to build an analog detector response would include `DepositionLaser`, `TransientPropagation` and `PulseTransfer`. Usually it is enough to run just a single event (or a few). While multithreading is supported by this module, one should note that the pipeline for each event is quite computationally intensive and runs with only one event do not gain any additional performance from multi-threaded execution.

Such pipeline is expected to produce pulse shapes, comparable with experimentally obtained ones. An example of `DepositionLaser` configuration is shown below.

```
[Allpix]
detectors_file = "geometry.conf"
# A single event is often enough
number_of_events = 1
multithreading = false

[DepositionLaser]
log_level = "INFO"

# Standard wavelength for IR TCT lasers
wavelength = 1064nm

number_of_photons = 50000
pulse_duration = 1ns

# Geometry parameters of the beam
source_position = 0 0 -5mm
beam_direction = 0 0 1
beam_geometry = "converging"
beam_waist = 10um
focal_distance = 5mm
beam_convergence_angle = 20deg
```

```
output_plots = true
```

8.10 DepositionPointCharge

8.10.1 Description

Module which deposits a defined number of charge carriers at a specific point within the active volume the detector. The number of charge carriers to be deposited can be specified in the configuration.

Two different source types are available:

- The `point` source deposits charge carriers at a specific point in the sensor, which can be configured via the `position` parameter with three dimensions. The number of charge carriers deposited can be adjusted using the `number_of_charges` parameter.
- The `mip` model allows to deposit charge carriers along a straight line through the sensor, perpendicular to its surface. Charge carriers are deposited linearly along this line with a configurable number of electron-hole pairs per length. The number of steps through the sensor can be configured using the `number_of_steps` parameter, the position can be given in two dimensions via the `position` parameter and the number of charge carriers per length are taken from the `number_of_charges` parameter.

This module supports three different deposition models:

- In the `fixed` model, charge carriers are always deposited at exactly the same position, specified via the `position` parameter, in every event of the simulation. This model is mostly interesting for development of new charge transport algorithms, where the initial position of charge carriers should be known exactly.
- In the `scan` model, the position where charge carriers are deposited changes with every event. The scanning positions are distributed such, that the volume of one pixel cell is homogeneously scanned. The total number of positions is taken from the total number of events configured for the simulation. If this number doesn't allow for a full illumination, a warning is printed, suggesting a different number of events. The pixel volume to be scanned is always placed at the center of the active sensor area. The scan model can be used to generate sensor response templates for fast simulations by generating a lookup table from the final simulation results.
- In the `spot` model, charge carriers are deposited in a Gaussian spot around the configured position. The sigma of the Gaussian distribution in all coordinates can be configured via the `spot_size` parameter. Charge carriers are only deposited inside the active sensor volume.

Monte Carlo particles are generated at the respective positions, bearing a particle ID of -1. All charge carriers are deposited at time zero, i.e. at the beginning of the event.

8.10.2 Parameters

- `model`: Model according to which charge carriers are deposited. For `fixed`, charge carriers are deposited at a specific point for every event. For `scan`, the point where charge carriers are deposited changes for every event. For `spot`, depositions are smeared around the configured position.
- `number_of_charges`: Number of charges deposited. This refers to the total number of charge carriers for the source type `point` and defaults to 1. For the `mip` source type, this value is interpreted as charge carriers per length deposited in the sensor and defaults to `80/um`. It should be noted that without units specified, this value will be interpreted in the framework base units, in this case `/mm`.
- `number_of_steps`: Number of steps over the full sensor thickness at which charge carriers are deposited. Only used for `mip` source type. Defaults to 100.
- `source_type`: Modeled source type for the deposition of charge carriers. For `point`, charge carriers are deposited at the position given by the `position` parameter. For `mip`, charge carriers are deposited along a line through the full sensor thickness. Defaults to `point`.
- `position`: Position in local coordinates of the sensor, where charge carriers should be deposited. Expects three values for local-x, local-y and local-z position in the sensor volume and defaults to `0um 0um 0um`, i.e. the center of first (lower left) pixel. Only used for the `fixed` and model. When using source type `mip`, providing a 2D position is sufficient since it only uses the x and y coordinates. If used in scan mode, it allows you to shift the origin of each deposited charge by adding this value.
- `spot_size`: Width of the Gaussian distribution used to smear the position in the `spot` model. Only one value is taken and used for all three dimensions.

8.10.3 Usage

Example configuration for a point source at a defined position around which charge carriers are deposited with a Gaussian distribution:

```
[DepositionPointCharge]
source_type = "point"
model = "spot"
position = -10um 10um 0um
spot_size = 3um
number_of_steps = 100
```

Example configuration for a MIP-like energy deposition along a line at a fixed position, with 63 electron-hole pairs deposited per micrometer of sensor material:

```
[DepositionPointCharge]
source_type = "mip"
model = "fixed"
position = -10um 10um
number_of_steps = 100
number_of_charges = 63/um
```

8.11 DepositionReader

8.11.1 Description

This module allows to read in energy depositions in a sensor volume produced with a different program, e.g. with Geant4 in a standalone simulation of the respective experiment. The detector geometry for Allpix Squared should resemble the global positions of the detectors of interest in the original simulation.

The assignment of energy deposits to a specific detector in the Allpix Squared simulation is performed using the volume name of the detector element in the original simulation. Hence, the naming of the detector in the geometry file has to match its name in the input data file. In order to simplify the aggregation of individual detector element volumes from the original simulation into a single detector, this modules provides the `detector_name_chars` parameter. It allows matching of the detector name to be performed on a sub-string of the original volume name.

Only energy deposits within a valid volume are considered, i.e. where a matching detector with the same name can be found in the geometry setup. The global coordinates are then translated to local coordinates of the given detector. If these are outside the sensor, the energy deposit is discarded and a warning is printed. The number of electron/hole pairs created by a given energy deposition is calculated using the mean pair creation energy [31], fluctuations are modeled using a Fano factor assuming Gaussian statistics [32]. Default values of both parameters for different sensor materials are included and automatically selected for each of the detectors. A full list of supported materials can be found elsewhere in the manual. These can be overwritten by specifying the parameters `charge_creation_energy` and `fano_factor` in the configuration.

Track and parent ids of the individual particles which created the energy depositions allow to carry on some of the Monte Carlo particle information from the original simulation. Monte Carlo particle objects are created for each unique track id, the start and end positions are set to the first and last appearance of the particle, respectively. A parent id of zero should be used for the primary particle of the simulation, and all track ids have to be recorded before they can be used as parent id.

With the `output_plots` parameter activated, the module produces histograms of the total deposited charge per event for every sensor in units of kilo-electrons. The scale of the plot axis can be adjusted using the `output_plots_scale` parameter and defaults to a maximum of 100ke.

Currently two data sources are supported, ROOT trees and CSV text files. Their expected formats are explained in detail in the following.

ROOT Trees

Data in ROOT trees are interpreted as follows. The tree with name `tree_name` is opened from the provided ROOT file, and information of energy deposits is read from its individual branches. By default the expected branch names and types are:

- `event` (integer): Branch for the event number.
- `energy` (double): Branch for the energy deposition information.

- `time` (double): Branch for the time information when the energy deposition took place, calculated from the start of the event.
- `position.x` (double): Leaf of the branch for the `x` position of the energy deposit in global coordinates of the setup.
- `position.y` (double): Leaf of the branch for the `y` position of the energy deposit in global coordinates of the setup.
- `position.z` (double): Leaf of the branch for the `z` position of the energy deposit in global coordinates of the setup.
- `detector` (char array): Branch for the detector or volume name in which the energy was deposited.
- `pdg_code` (integer): Branch for the PDG code particle id if the Monte Carlo particle producing this energy deposition.
- `track_id` (integer): Branch for the track id of the current Monte Carlo particle.
- `parent_id` (integer): Branch for the id of the parent Monte Carlo particle which created the current one.

Entries are read from all branches synchronously and accumulated in the same event until the event id read from the `event` branch changes.

By default, the event numbers need to be sorted with ascending order. This can be disabled by setting `require_sequential_events` to `false`. This is useful when running simulations in mutli-threading mode and merging datasets in the end. Currently only supported in ROOT files.

If the parameters `assign_timestamps` or `create_mcparticles` are set to `false`, no attempt is made in reading the respective branches, independently whether they are present or not.

Different branch names can be configured using the `branch_names` parameter. It should be noted that new names have to be provided for all branches, i.e. ten names, and that the order of the names has to reflect the order of the branches as listed above to allow for correct assignment. If `assign_timestamps` or `create_mcparticles` are set to `false`, their branch names (`time` and `track_id`, `parent_id`, respectively) should be omitted from the branch name list. Individual leafs of branches can be assigned using the dot notation, e.g. `energy.Edep` to access a leaf of the branch `energy` to retrieve the energy deposit information.

CSV Files

Data in CSV-formatted text files are interpreted as follows. Empty lines as well as lines starting with a hash (`#`) are ignored, all other lines are interpreted either as event header if they start with `E`, or as energy deposition:

```
Event: <N>
<PID>,<T>,<E>,<X>,<Y>,<Z>,<V>,<TRK>,<PRT>
<PID>,<T>,<E>,<X>,<Y>,<Z>,<V>,<TRK>,<PRT>
# ...
# For example:
211, 3.234674e+01, 1.091620e-02, -2.515335e+00, 4.427924e+00,
↪  -2.497500e-01, MyDetector, 1, 0
```

```
211, 3.234843e+01, 1.184756e-02, -2.528475e+00, 4.453544e+00,
 ↪  -2.445500e-01, MyDetector, 2, 1

Event: <N+1>
<PID>,<T>,<E>,<X>,<Y>,<Z>,<V>,<TRK>,<PRT>
# ...
```

where <N> is the current event number, <PID> is the PDG particle ID [72], <T> the time of deposition, calculated from the beginning of the event, <E> is the deposited energy, <[X-Z]> is the position of the energy deposit in global coordinates of the setup, and <V> the detector name (volume) the energy deposit should be assigned to. The values are interpreted in the default framework units unless specified otherwise via the configuration parameters of this module. <TRK> represents the track id of the particle track which has caused this energy deposition, and <PRT> the id of the parent particle which created this particle.

If the parameters `assign_timestamps` or `create_mcparticles` are set to `false`, the parsing assumes that the respective columns <T> and <TRK>, <PRT> are not present in the CSV file.

The file should have its end-of-file marker (EOF) in a new line, otherwise the last entry will be ignored.

8.11.2 Parameters

- `model`: Format of the data file to be read, can either be `csv` or `root`.
- `file_name`: Location of the input data file. The appropriate file extension will be appended if not present, depending on the `model` chosen either `.csv` or `.root`.
- `tree_name`: Name of the input tree to be read from the ROOT file. Only used for the `root` model.
- `branch_names`: List of names of the ten branches to be read from the input ROOT file. Only used for the `root` model. The default names and their content are listed above in the *ROOT Trees* section.
- `detector_name_chars`: Parameter which allows selecting only a sub-string of the stored volume name as detector name. Could be set to the number of characters from the beginning of the volume name string which should be taken as detector name. E.g. `detector_name_chars = 7` would select `sensor0` from the full volume name `sensor0_px3_14` read from the input file. This is especially useful if the initial simulation in Geant4 has been performed using parameterized volume placements e.g. for individual pixels of a detector. Defaults to `0` which takes the full volume name.
- `charge_creation_energy` : Energy needed to create a charge deposit. Defaults to the energy needed to create an electron-hole pair in the respective sensor material (e.g. 3.64 eV for silicon sensors, [31]). A full list of supported materials can be found elsewhere in the manual.
- `fano_factor`: Fano factor to calculate fluctuations in the number of electron/hole pairs produced by a given energy deposition. Defaults are provided for different sensor materials, e.g. a value of 0.115 for silicon [32]. A full list of supported materials can be found elsewhere in the manual.

- `unit_length`: The units length measurements read from the input data source should be interpreted in. Defaults to the framework standard unit `mm`.
- `unit_time`: The units time measurements read from the input data source should be interpreted in. Defaults to the framework standard unit `ns`.
- `unit_energy`: The units energy depositions read from the input data source should be interpreted in. Defaults to the framework standard unit `MeV`.
- `assign_timestamps`: Boolean to select whether or not time information should be read and assigned to energy deposits. If `false`, all timestamps of deposits are set to 0. Defaults to `true`.
- `create_mcparticles`: Boolean to select whether or not Monte Carlo particle IDs should be read and MCParticle objects created, defaults to `true`.
- `output_plots` : Enables output histograms to be generated from the data in every step (slows down simulation considerably). Disabled by default.
- `output_plots_scale` : Set the x-axis scale of the output plot, defaults to 100ke.

8.11.3 Usage

An example for reading energy depositions from a ROOT file tree named `hitTree`, using only the first five characters of the volume name as detector identifier and meter as unit length, is the following:

```
[DepositionReader]
model = "root"
file_name = "g4_energy_deposits.root"
tree_name = "hitTree"
detector_name_chars = 5
unit_length = "m"
branch_names = ["event", "energy.Edep", "time", "position.x",
 ↪  "position.y", "position.z", "detector", "PDG_code", "track_id",
 ↪  "parent_id"]
```

8.12 DetectorHistogrammer

8.12.1 Description

This module provides an overview of the produced simulation data for a quick inspection and simple checks. For more sophisticated analyses, the output from one of the output writers should be used to make the necessary information available.

Within the module, clustering of the input hits is performed. Looping over the PixelHits, hits being adjacent to an existing cluster are added to this cluster. Clusters are merged if there are multiple adjacent clusters. If the PixelHit is free-standing, a new cluster is created.

This module serves as a quick “mini-analysis” and creates the histograms listed below. The Monte Carlo truth position provided by the `MCParticle` objects is used as track reference position. An additional uncertainty can be added by configuring a track resolution, with which every cluster residual is convolved. This makes it possible to perform a quick test beam-like analysis. For technical reasons, this offset is drawn randomly from a Gaussian

distribution independently for the resolution and the efficiency measurement. **Note:** If a non-zero track resolution is used, pixel matrix edge effects may appear as particles hit the sensor excess.

- A hitmap of all pixels in the pixel grid, displaying the number of times a pixel has been hit during the simulation run.
- A cluster map indicating the cluster positions for the whole simulation run.
- Distribution of the total number of pixel hits (event size) per event.
- Distribution of the total number of clusters found per event.
- Distributions of the cluster size in x, y and the total cluster size.
- Mean cluster size and cluster sizes in x and y as function of the in-pixel impact position of the primary particle.
- Residual distribution in x and y between the center-of-gravity position of the cluster and the primary particle.
- Residual map for residuals in x, y, and combined between the center-of-gravity position of the cluster and the primary particle. These maps allow to see if the residuals are smaller or larger on some part of the detector compared to others.
- Mean absolute deviations of the residual as function of the in-pixel impact position of the primary particle. Histograms both for a 2D representation of the pixel cell as well as the projections (residual X vs position X, residual Y vs position Y, residual X vs position Y, residual Y vs position X) are produced.
- Efficiency map of the detector
- Efficiency as function of the in-pixel impact position of the primary particle. Histograms both for a 2D representation of the pixel cell as well as the projections (efficiency vs position X, efficiency vs position Y) are produced.
- Total cluster, pixel and event charge distributions.
- Mean total cluster charge as function of the in-pixel impact position of the primary particle.
- Mean seed pixel charge as a function of the in-pixel impact position of the primary particle.

8.12.2 Parameters

- `granularity`: 2D integer vector defining the number of bins along the x and y axis for in-pixel maps. Defaults to the pixel pitch in micro meters, e.g. a detector with 100um x 100um pixels would be represented in a histogram with `100 * 100 = 10000` bins.
- `granularity_local`: 2D integer vector defining the number of bins for each pixel along the x and y axis for maps in local coordinates where particle positions are used as reference. Defaults to `1 1` corresponding to a single bin per pixel.
- `max_cluster_charge`: Upper limit for the cluster charge histogram, defaults to `50ke`.
- `track_resolution`: Assumed track resolution the Monte Carlo truth is smeared with. Expects two values for the resolution in local-x and local-y directions and defaults to `0um 0um`, i.e. no smearing.
- `matching_cut`: Required maximum matching distance between cluster position and particle position for the efficiency measurement. Expected two values and defaults to three times the pixel pitch in each dimension.

8.12.3 Usage

This module is normally bound to a specific detector to plot, for example to the 'dut':

```
[DetectorHistogrammer]
name = "dut"
granularity = 100, 100
```

8.13 DopingProfileReader

8.13.1 Description

Adds a doping profile to the detector from one of the supported sources. By default, detectors do not have a doping profile applied. A doping profile is required for simulating the lifetime of charge carriers. It is not used for the calculation of the electric field inside the sensor. The profile is extrapolated along z such that if a position outside the sensor is queried, the last value available at the sensor surface is returned. This precludes edge effects from charge carriers moving at the sensor surfaces.

The following models for the doping profile can be used:

- For **constant**, a constant doping profile is set in the sensor
- For **regions**, the sensor is segmented into slices along the local z-direction. In each slice, a constant doping concentration is used. The user provides the depth of each slice and the corresponding concentration.
- For **mesh**, a file containing a doping profile map in APF or INIT format is parsed.

8.13.2 Parameters

- `model` : Type of the doping profile, either **constant**, **regions** or **mesh**.
- `file_name` : Location of file containing the doping profile in one of the supported field file formats. Only used if the *model* parameter has the value **mesh**.
- `field_mapping`: Description of the mapping of the field onto the sensor or pixel cell. Possible values are `SENSOR` for sensor-wide mapping, `PIXEL_FULL`, indicating that the map spans the full 2D plane and the field is centered around the pixel center, `PIXEL_HALF_TOP` or `PIXEL_HALF_BOTTOM` indicating that the field only contains only one half-axis along y, `HALF_LEFT` or `HALF_RIGHT` indicating that the field only contains only one half-axis along x, or `PIXEL_QUADRANT_I`, `PIXEL_QUADRANT_II`, `PIXEL_QUADRANT_III`, `PIXEL_QUADRANT_IV` stating that the field only covers the respective quadrant of the 2D pixel plane. In addition, the `PIXEL_FULL_INVERSE` mode allows loading full-plane field maps which are not centered around a pixel cell but the corner between pixels. Only used if the *model* parameter has the value **mesh**.
- `field_scale`: Scaling factor of the electric field in x- and y-direction. By default, the scaling factors are set to `{1, 1}` and the field is used with its physical extent stated in the field data file.

- `field_offset`: Offset of the field in x- and y-direction. With this parameter and the mapping mode `SENSOR`, the field can be shifted e.g. by half a pixel pitch to accommodate for fields which have been simulated starting from the pixel center. The shift is applied in positive direction of the respective coordinate. Only used if the *model* parameter has the value **mesh**.
- `doping_concentration` : Value for the doping concentration. If the *model* parameter has the value **constant** a single number should be provided. If the *model* parameter has the value **regions** a matrix is expected, which provides the sensor depth and doping concentration in each row.
- `doping_depth` : Thickness of the doping profile region. The doping profile is extrapolated in the region below the `doping_depth`. Only used if the *model* parameter has the value **mesh**.

8.13.3 Plotting parameters

- `output_plots` : Determines if output plots should be generated. Disabled by default.
- `output_plots_steps` : Number of bins in both x- and y-direction in the 2D histogram used to plot the doping concentration in the detectors. Only used if `output_plots` is enabled.
- `output_plots_project` : Axis to project the doping concentration on to create the 2D histogram. Either **x**, **y** or **z**. Only used if `output_plots` is enabled. Default is **x** (i.e. producing a slice of the **yz** plane).
- `output_plots_projection_percentage` : Percentage on the projection axis to plot the doping concentration profile. For example if *output_plots_project* is **x** and this parameter is set to 0.5, the profile is plotted in the **yz** plane at the x-coordinate in the middle of the sensor. Default is 0.5.
- `output_plots_single_pixel`: Determines if the whole sensor has to be plotted or only a single pixel. Defaults to true (plotting a single pixel).

8.13.4 Usage

```
[DopingProfileReader]
model = "mesh"
file_name = "example_doping_profile.apf"
```

8.14 Dummy

8.14.1 Description

Description of this module

8.14.2 Parameters

- `param`: *explanation with optional default*

8.14.3 Usage

Example how to use this module

8.15 ElectricFieldReader

8.15.1 Description

Adds an electric field to the detector from one of the supported sources. By default, detectors do not have an electric field applied.

The reader provides the following models for electric fields:

- For **constant** electric fields it add a constant electric field in the z-direction towards the pixel implants. This is not physical but might aid in developing and testing new charge propagation algorithms.
- For **linear** electric fields, the field has a constant slope determined by the bias voltage and the depletion voltage. The sensor is depleted either from the implant or the back side, the direction of the electric field depends on the sign of the bias voltage (with negative bias voltage the electric field vector points towards the backplane and vice versa). The sign of depletion voltage is always ignored. If the sensor is depleted from the implant side, the absolute value of the electric field is calculated using the formula

  $$E(z) = \frac{|U_{bias}| - |U_{depl}|}{d} + 2\frac{|U_{depl}|}{d}\left(1 - \frac{z}{d}\right),$$

  where d is the thickness of the sensor, and U_{depl}, U_{bias} are the depletion and bias voltages, respectively. In case of a depletion from the back side, the absolute value of the electric field is calculated as

  $$E(z) = \frac{|U_{bias}| - |U_{depl}|}{d} + 2\frac{|U_{depl}|}{d}\left(\frac{z}{d}\right).$$

- For **parabolic** electric fields, a parabola is defined in order to emulate a double-peaked field such as the electric fields observed in sensors after irradiation. The parabola is calculated from the position z_{min} and value E_{min} of the minimum field in the sensor and the field value at the readout electrode, E_{max}. The parameters of parabolic equation $E(z) = az^2 + bz + c$ then resolve to:

  $$a = \frac{E_{max} - E_{min}}{z_{min}^2 + (d/2)^2 - dz_{min}} \qquad b = -2az_{min} \qquad c = E_{max} - a((d/2)^2 - dz_{min}),$$

  where d is the sensor thickness and z the position along the z-axis in local coordinates, from $-d/2$ to $+d/2$. The direction of the electric field is determined by the sign of the field parameters.

- For electric fields from **mesh files** in the *INIT* or *APF* formats it parses a file containing an electric field map in the APF format or the legacy INIT format also used by the PixelAV software [87]. An example of a electric field in this format can be found in *etc/example_electric_field.init* in the repository. An explanation of the format is available in the source code of this module, a converter tool for electric fields from adaptive TCAD meshes is provided with the framework. Fields of different sizes can be used and mapped onto the pixel matrix using the `field_scale` parameter. By default, the module reads the size of the field from the file. If the field size and pixel pitch do not match, a warning is printed.
- The **custom** field model allows to specify arbitrary analytic field functions for a single or all three vector components of the electric field. For this, the `field_functions` parameter configured with either one formula which is then used for the `z` component of the field vector, or with three functions representing the three components of the field vector. Using the `field_parameters` configuration, values for the free parameters defined in the formulae can be set. For the parameters units are supported and parsed. Each of the field vector components has access to its own free parameters as well as all three coordinates `x`, `y` and `z` which are defined as the position within the respective pixel.

The `depletion_depth` parameter can be used to control the thickness of the depleted region inside the sensor. This can be useful for devices such as HV-CMOS sensors, where the typical depletion depth but not necessarily the full depletion voltage are know. It should be noted that `depletion_voltage` and `depletion_depth` are mutually exclusive parameters and only one at a time can be specified. The alias `field_depth` can be used instead, as this parameter is the depth that the field will be created over. If the parameter is smaller than the field from an imported mesh, the field will be compressed in the z-direction.

Furthermore the module can plot the electric field profile on an projection axis normal to the x,y or z-axis at a particular plane in the sensor. Additional plots comprise the individual field vector components as well as the field magnitude and can be enabled and controlled with the plotting parameters listed below.

8.15.2 Parameters

- `model` : Type of the electric field model, either **linear**, **constant**, **parabolic**, **custom** or **mesh**.
- `depletion_depth` : Thickness of the depleted region. Used for all electric fields. When using the depletion depth for the **linear** model, no depletion voltage can be specified. Defaults to the full sensor thickness. The alias `field_depth` can be used for improved readability when using the model **mesh** (as the depletion depth in an externally generated field may be smaller than the field depth).

Parameters for models `linear` and `constant`

- `bias_voltage` : Voltage over the whole sensor thickness. Used to calculate the electric field for the models **constant** and **linear**.

- `depletion_voltage` : Indicates the voltage at which the sensor is fully depleted. Used to calculate the electric field if the *model* parameter is equal to **linear**.
- `deplete_from_implants` : Indicates whether the sensor is depleted from the implants or the back side for the **linear** model. Defaults to true (depletion from the implant side).

Parameters for model `parabolic`

- `minimum_field` : Value of the electric field in the minimum.
- `minimum_position` : Position of the electric field minimum along `z`, in local coordinates. Required to be located within the sensor volume.
- `maximum_field` : Value of the electric field at the electrode.

Parameters for model `mesh`

- `file_name` : Location of file containing the meshed electric field data.
- `field_mapping`: Description of the mapping of the field onto the sensor or pixel cell. Possible values are `SENSOR` for sensor-wide mapping, `PIXEL_FULL`, indicating that the map spans the full 2D plane and the field is centered around the pixel center, `PIXEL_HALF_TOP` or `PIXEL_HALF_BOTTOM` indicating that the field only contains only one half-axis along `y`, `HALF_LEFT` or `HALF_RIGHT` indicating that the field only contains only one half-axis along `x`, or `PIXEL_QUADRANT_I`, `PIXEL_QUADRANT_II`, `PIXEL_QUADRANT_III`, `PIXEL_QUADRANT_IV` stating that the field only covers the respective quadrant of the 2D pixel plane. In addition, the `PIXEL_FULL_INVERSE` mode allows loading full-plane field maps which are not centered around a pixel cell but the corner between pixels.
- `field_scale`: Scaling factor of the electric field in x- and y-direction. By default, the scaling factors are set to `{1, 1}` and the field is used with its physical extent stated in the field data file. To scale the field in the z-direction, the parameter `field_depth` can be used.
- `field_offset`: Offset of the field in x- and y-direction. With this parameter and the mapping mode `SENSOR`, the field can be shifted e.g. by half a pixel pitch to accommodate for fields which have been simulated starting from the pixel center. The shift is applied in positive direction of the respective coordinate.

Parameters for model `custom`

- `field_functions` : Single equation (for a field vector along the `z` axis only) or array of three equations (for the three components of a vector field). All three coordinates `x`, `y`, and `z` can be used, parameters need to be specified in consecutively numbered square brackets (`[0]`, `[1]`), starting with `[0]` for each of the equations.
- `field_parameters` : Array of values for the parameters of any equation defined in `field_equations`. Units can be used. The number of parameters given must match the sum of the number of free parameters from all defined equations.

8.15.3 Plotting parameters

- `output_plots` : Determines if output plots should be generated. Disabled by default.
- `output_plots_steps` : Number of bins in both x- and y-direction in the 2D histogram used to plot the electric field in the detectors. Only used if `output_plots` is enabled.
- `output_plots_project` : Axis to project the 3D electric field on to create the 2D histogram. Either **x**, **y** or **z**. Only used if `output_plots` is enabled.
- `output_plots_projection_percentage` : Percentage on the projection axis to plot the electric field profile. For example if *output_plots_project* is **x** and this parameter is set to 0.5, the profile is plotted in the Y,Z-plane at the X-coordinate in the middle of the sensor. Default is 0.5.
- `output_plots_single_pixel`: Determines if the whole sensor has to be plotted or only a single pixel. Defaults to true (plotting a single pixel).

8.15.4 Usage

An example to add a linear field with a bias voltage of -150 V and a full depletion voltage of -50 V to all the detectors, apart from the detector named 'dut' where a specific meshed field from an INIT file is added, is given below

```
[ElectricFieldReader]
model = "linear"
bias_voltage = -150V
depletion_voltage = -50V

[ElectricFieldReader]
name = "dut"
model = "mesh"
# Should point to the example electric field in the repositories etc
 ↪  directory
file_name = "example_electric_field.init"
```

This example uses the parabolic field shape and defines a minimum field and position as well as the field at the electrode:

```
[ElectricFieldReader]
model = "parabolic"
# In local coordinates of the sensor, i.e. 100um below the center of the
 ↪  sensor along z:
minimum_position = -100um
minimum_field = 5200V/cm
maximum_field = 10000V/cm
```

An example for a custom field definition is given below. Here, a one-dimensional field is defined, which will be automatically applied to the z-axis of the detector. Care should be take to use the proper variables in the formula, in this case z for the respective coordinate.

```
[ElectricFieldReader]
model = "custom"
field_function = "[0]*z*z + [1]"
field_parameters = 12500V/mm/mm/mm, 5000V/cm
```

And finally, a three-dimensional custom field is defined with varying number of parameters per equation and using different coordinates for the three dimensions of the field vector:

```
[ElectricFieldReader]
model = "custom"
# Parabolic in x and y, linear in z:
field_function = "[0]*x*y","[0]*x*y","[0]*z + [1]"
field_parameters = 12500V/mm/mm/mm, 12500V/mm/mm/mm, 6000V/cm/cm,
 ↪  5000V/cm
```

8.16 GDMLOutputWriter

8.16.1 Description

Constructs a GDML output file of the geometry if this module is added. This feature is to be considered experimental as the GDML implementation of Geant4 is incomplete.

8.16.2 Dependencies

This module requires an installation `Geant4_GDML`. This option can be enabled by configuring and compiling Geant4 with the option `-DGEANT4_USE_GDML=ON`

8.16.3 Parameters

- `file_name` : Name of the data file to create, relative to the output directory of the framework. The file extension .gdml will be appended if not present. Defaults to `Output.gdml`

8.16.4 Usage

Creating a GDML output file with the name `myOutputfile.gdml`:

```
[GDMLOutputWriter]
file_name = myOutputfile
```

8.17 GenericPropagation

8.17.1 Description

Simulates the propagation of electrons and/or holes through the sensitive sensor volume of the detector. It allows to propagate sets of charge carriers together in order to speed up the simulation while maintaining the required accuracy. The propagation process for these sets is fully independent and no interaction is simulated. The maximum size of the set of propagated charges and thus the accuracy of the propagation can be controlled via the `charge_per_step` parameter. The maximum number of charge groups to be propagated for a single deposit position can be controlled via the `max_charge_groups` parameter.

The propagation consists of a combination of drift and diffusion simulation. The drift is calculated using the charge carrier velocity derived from the charge carrier mobility and the magnetic field via a calculation of the Lorentz drift. The correct mobility for either electrons or holes is automatically chosen, based on the type of the charge carrier under consideration. Thus, also input with both electrons and holes is treated properly. The mobility model can be chosen using the `mobility_model` parameter, and a list of available models can be found in the user manual.

This module implements charge multiplication by impact ionization. The multiplication model can be chosen using the `multiplication_model` parameter, the list of available models can be found in the user manual. By default, the model defaults to `none` and impact ionization is switched off, generating unity gain. To simulate impact ionization, the number of newly generated electron-hole pairs is calculated for every propagation step and every charge carrier in the group, based on drawing a random number from a geometric distribution. This represents a stepwise approach to the avalanche generation process. The charge of a charge group is increased by the number of impact ionization processes per step and opposite-type charge carriers are generated at the end of the step, if the opposite-type charge carrier is selected to be propagated (see below).

The two parameters `propagate_electrons` and `propagate_holes` allow to control which type of charge carrier is propagated to their respective electrodes. Either one of the carrier types can be selected, or both can be propagated. It should be noted that this will slow down the simulation considerably since twice as many carriers have to be handled and it should only be used where sensible. The direction of the propagation depends on the electric and magnetic fields field configured, and it should be ensured that the carrier types selected are actually transported to the implant side. For linear electric fields, a warning is issued if a possible misconfiguration is detected.

A fourth-order Runge-Kutta-Fehlberg method [22] with fifth-order error estimation is used to integrate the particle propagation in the electric and magnetic fields. After every Runge-Kutta step, the diffusion is accounted for by applying an offset drawn from a Gaussian distribution calculated from the Einstein relation

$$\sigma = \sqrt{\frac{2k_bT}{e}\mu t}$$

using the carrier mobility μ, the temperature T and the time step t. The propagation stops when the set of charges reaches any surface of the sensor.

The charge carrier lifetime can be simulated using the doping concentration of the sensor. The recombination model is selected via the `recombination_model` parameter, the default value `none` is equivalent to not simulating finite lifetimes. This feature can only be enabled if a doping profile has been loaded for the respective detector using the DopingProfileReader module. In each step, the doping-dependent charge carrier lifetime is determined, from which a survival probability is calculated. The survival probability is calculated at each step of the propagation by drawing a random number from an uniform distribution with $0 \leq r \leq 1$ and comparing it to the expression dt/τ, where dt is the time step of the last charge carrier movement.

Trapping of charge carriers can be enabled by setting a trapping model via the parameter `trapping_model`. The default value is `none`, corresponding to no charge carrier trapping being simulated. All models require the 1MeV-neutron equivalent fluence to be set via the parameter `fluence`. Some models include temperature-dependent scaling of trapping probabilities, and the corresponding temperature is taken from the `temperature` parameter. The trapping probability is calculated at each step of the propagation by drawing a random number from an uniform distribution with $0 \leq r \leq 1$ and comparing it to the expression $1 - e^{-dt/\tau_{eff}}$, where dt is the time step of the last charge carrier movement and τ_{eff} the effective trapping time constant. A list of available models can be found in the user manual.

Detrapping of charge carriers can be enabled by setting a detrapping model via the parameter `detrapping_model`. The default value is `none`, corresponding to no charge carrier detrapping being simulated. A list of available models can be found in the user manual.

The propagation module also produces a variety of output plots. These include a 3D line plot of the path of all separately propagated charge carrier sets from their point of deposition to the end of their drift, with nearby paths having different colors. In this coloring scheme, electrons are marked in blue colors, while holes are presented in different shades of orange. In addition, a 3D GIF animation for the drift of all individual sets of charges (with the size of the point proportional to the number of charges in the set) can be produced. Finally, the module produces 2D contour animations in all the planes normal to the X, Y and Z axis, showing the concentration flow in the sensor. It should be noted that generating the animations is time-consuming and should be switched off even when investigating drift behavior.

8.17.2 Dependencies

This module requires an installation of Eigen3.

8.17.3 Parameters

- `temperature` : Temperature of the sensitive device, used to estimate the diffusion constant and therefore the strength of the diffusion. Defaults to room temperature (293.15K).
- `mobility_model`: Charge carrier mobility model to be used for the propagation. Defaults to `jacoboni`, a list of available models can be found in the documentation.

- `recombination_model`: Charge carrier lifetime model to be used for the propagation. Defaults to `none`, a list of available models can be found in the documentation. This feature requires a doping concentration to be present for the detector.
- `trapping_model`: Model for simulating charge carrier trapping from radiation-induced damage. Defaults to `none`, a list of available models can be found in the documentation. All models require explicitly setting a fluence parameter.
- `fluence`: 1MeV-neutron equivalent fluence the sensor has been exposed to.
- `detrapping_model`: Model for simulating charge carrier detrapping from radiation-induced damage. Defaults to `none`, a list of available models can be found in the documentation.
- `charge_per_step` : Maximum number of charge carriers to propagate together. Divides the total number of deposited charge carriers at a specific point into sets of this number of charge carriers and a set with the remaining charge carriers. A value of 10 charges per step is used by default if this value is not specified.
- `max_charge_groups`: Maximum number of charge groups to propagate from a single deposit point. Temporarily increases the value of `charge_per_step` to reduce the number of propagated groups if the deposit is larger than the value `max_charge_groups*charge_per_step`, thus reducing the negative performance impact of unexpectedly large deposits. The default value is 1000 charge groups. If it is set to 0, there is no upper limit on the number of charge groups propagated.
- `spatial_precision` : Spatial precision to aim for. The timestep of the Runge-Kutta propagation is adjusted to reach this spatial precision after calculating the uncertainty from the fifth-order error method. Defaults to 0.25nm.
- `timestep_start` : Timestep to initialize the Runge-Kutta integration with. Appropriate initialization of this parameter reduces the time to optimize the timestep to the *spatial_precision* parameter. Default value is 0.01ns.
- `timestep_min` : Minimum step in time to use for the Runge-Kutta integration regardless of the spatial precision. Defaults to 1ps.
- `timestep_max` : Maximum step in time to use for the Runge-Kutta integration regardless of the spatial precision. Defaults to 0.5ns.
- `integration_time` : Time within which charge carriers are propagated. After exceeding this time, no further propagation is performed for the respective carriers. Defaults to the LHC bunch crossing time of 25ns.
- `propagate_electrons` : Select whether electron-type charge carriers should be propagated to the electrodes. Defaults to true.
- `propagate_holes` : Select whether hole-type charge carriers should be propagated to the electrodes. Defaults to false.
- `ignore_magnetic_field`: The magnetic field, if present, is ignored for this module. Defaults to false.
- `multiplication_model`: Model used to calculate impact ionization parameters and charge multiplication. Defaults to `none` which corresponds to unity gain, a list of available models can be found in the documentation.
- `multiplication_threshold`: Threshold field above which charge multiplication is calculated. Defaults to `100kV/cm`.
- `max_multiplication_level`: Maximum level depth of the generated impact ionization charge multiplication shower after which the generation of further multiplication charge carrier levels is prohibited. This number represents the maximum number of daughter charge carrier groups that can be produced by one initial charge carrier group. This does not concern the size of the charge group itself but solely the level of

generation. If a group generates a secondary group through impact ionization, the depth is 1. If this secondary group again creates charge carriers when propagating, the level is 2 and so on. The default value is 5.

8.17.4 Plotting parameters

- `output_plots` : Determines if simple output plots should be generated for a monitoring of the simulation flow. Disabled by default.
- `output_linegraphs` : Determines if line graphs should be generated for every event. This causes a significant slow down of the simulation, it is not recommended to enable this option for runs with more than a couple of events. Disabled by default.
- `output_linegraphs_collected` : Determine whether to also generate line graphs *only* for charge carriers that have reached the implant side within the allotted integration time. Defaults to `false`. This requires `output_linegraphs` to be active.
- `output_linegraphs_recombined` : Boolean flag to select whether line graphs should also be generated only from charge carriers that have recombined with the lattice during the integration time. Defaults to `false`. This requires `output_linegraphs` to be active.
- `output_linegraphs_trapped` : Boolean flag to select whether line graphs should also be generated only from charge carriers that have been trapped during their motion through the sensor. Defaults to `false`. This requires `output_linegraphs` to be active.
- `output_plots_step` : Timestep to use between two points plotted. Indirectly determines the amount of points plotted. Defaults to *timestep_max* if not explicitly specified.
- `output_plots_theta` : Viewpoint angle of the 3D animation and the 3D line graph around the world X-axis. Defaults to zero.
- `output_plots_phi` : Viewpoint angle of the 3D animation and the 3D line graph around the world Z-axis. Defaults to zero.
- `output_plots_use_pixel_units` : Determines if the plots should use pixels as unit instead of metric length scales. Defaults to false (thus using the metric system).
- `output_plots_use_equal_scaling` : Determines if the plots should be produced with equal distance scales on every axis (also if this implies that some points will fall out of the graph). Defaults to true.
- `output_plots_align_pixels` : Determines if the plot should be aligned on pixels, defaults to false. If enabled the start and the end of the axis will be at the split point between pixels.
- `output_animations` : In addition to the other output plots, also write a GIF animation of the charges drifting towards the electrodes. This is extremely slow and writing the animation takes a considerable amount of time, therefore defaults to false. This option also requires `output_linegraphs` to be enabled.
- `output_animations_time_scaling` : Scaling for the animation used to convert the actual simulation time to the time step in the animation. Defaults to 1.0e9, meaning that every nanosecond of the simulation is equal to an animation step of a single second.
- `output_animations_marker_size` : Scaling for the markers on the animation, defaults to one. The markers are already internally scaled to the charge of their

step, normalized to the maximum charge.
- `output_animations_contour_max_scaling` : Scaling to use for the contour color axis from the theoretical maximum charge at every single plot step. Default is 10, meaning that the maximum of the color scale axis is equal to the total amount of charges divided by ten (values above this are displayed in the same maximum color). Parameter can be used to improve the color scale of the contour plots.
- `output_animations_color_markers`: Determines if colors should be for the markers in the animations, defaults to false.

8.17.5 Usage

A example of generic propagation for all sensors of type *Timepix* at room temperature using packets of 25 charges is the following:

```
[GenericPropagation]
type = "timepix"
temperature = 293K
charge_per_step = 25
```

8.18 GeometryBuilderGeant4

8.18.1 Description

Constructs the Geant4 geometry from the internal geometry description. First, the world frame with a configurable margin and material is constructed. Then all passive materials and detectors using their internal detector models and passive material models are created and placed within the world frame or a specified mother volume (only for passive materials), which corresponds to another passive volume. The descriptions of all detectors and passive volumes have to be specified within the geometry configuration.

All available detector models are fully supported.

Passive Volumes

For passive materials, the implemented models are “box”, “cylinder”, “sphere” as well as “gdml”. The dimensions of the individual volumes are defined by the following parameters for the specific models and to be set within the corresponding section of the geometry configuration:

For each model, a set of specific size parameters need to be given, of which some are optional.

Box: A rectangular box which can be massive or have an hole in the middle along the z-axis.

- The `size` of the box is an XYZ vector which defines the total size of the box.
- (Optional) The `inner_size` of the box is an XYZ vector which defines the size of the volume that will be removed at the center of the original box volume. Defaults to `0mm 0mm 0mm` (no volume removed).
- (Optional) The `thickness` of the box is a value which defines the thickness of the walls of a box. This has a similar effect as the parameter `inner_size`, and such they can't be used together. Defaults to 0mm.

Cylinder: A cylindrical tube which can be massive or have an hole in the middle along the z-axis.

- The `outer_radius` of the cylinder is the total radius of the cylinder (in the XY-plane).
- The `length` of the cylinder is the total length of the cylinder (in the Z-direction).
- (Optional) The `inner_radius` of the cylinder is the radius of the inner cylinder. Defaults to 0mm.
- (Optional) The `starting_angle` of the cylinder is the angle at which circumference of the cylinder will start. 0 degrees refers to the point along the positive x-axis and the angle moves counter clockwise. Defaults to 0deg.
- (Optional) The `arc_length` of the cylinder is the arc-length of the circumference that will be drawn, starting from the given `starting_angle`. Defaults to 360deg which is the full circumference.

Note that the if the `arc_length` is set to 360 degrees, the Allpix Squared framework will always draw the full circumference, regardless of the value of `starting_angle`.

Sphere: A full or partly made sphere with an inner- and outer radius.

- The `outer_radius` of the sphere is the total radius of the sphere in all directions.
- (Optional) The `inner_radius` of the sphere is the radius of the inner sphere. Defaults to 0mm.
- (Optional) The `starting_angle_phi` of the sphere is the azimuthal angle at which circumference of the sphere will start in the XY-plane. 0 degrees refers to the point along the positive x-axis and the angle moves counter clockwise. Defaults to 0deg.
- (Optional) The `arc_length_phi` of the sphere is the arc-length of the circumference that will be drawn, starting from the given `starting_angle_phi` in the XY-plane. Defaults to 360deg which is the full circumference.
- (Optional) The `starting_angle_theta` of the sphere is the polar angle at which the `arc_length_theta` will start. 0 degrees refers to the point along the positive z-axis. Defaults to 0deg.
- (Optional) The `arc_length_theta` of the sphere is the arc-length of the polar angle which will be rotated around the z-axis to build the sphere, starting from the given `starting_angle_theta`. Defaults to 100deg which creates the full circle.

Note that `arc_length_phi` works the same as the `arc_length` from the cylinder, but the `arc_length_theta` works different. The Allpix Squared framework will only draw the full circle if `starting_angle_theta` = 0deg, and `arc_length_theta` = 180deg. In

all other situations, the sphere will start at `starting_angle_theta` and continue the `arc_length_theta` until `arc_length_theta` + `starting_angle_theta` = 180deg. After this it will stop. The necessary module errors and warnings have been included to make sure the user will know will and won't be build.

Note: If the VisualizationGeant4 module is used in conjunction with and `arc_length_theta` different from 180deg, the Visualization GUI will show an error "Inconsistency in bounding boxes for solid". The origin of this error is unknown but the error can be ignored.

GDML: This model allows to load arbitrary GDML files [88] as passive materials. All volumes from the GDML file which are contained within the world volume are processed and added to the geometry of the simulation. The only parameter specific to this model is `file_name` which should provide the path to the GDML file to be read.

This functionality requires Geant4 to be built with GDML support enabled. This can be enabled via CMake when compiling Geant4 using

```
cmake -DDGEANT4_USE_GDML=ON ..
```

Visualization Options

For each of the above mentioned models, a color and opacity can be added to the passive material.

- The `color` of the passive material is given in an R G B vector, where each color value is between 0 and 1. Defaults to `color = 0 0 1` (blue).
- The `opacity` of the passive material is given as a number between 0 and 1, where 0 is completely transparent, and 1 is completely opaque.

Materials

The following materials are pre-defined and can directly be used for the world volume, detector support layers as well as passive volumes: This module can create support layers and passive volumes of the following materials:

- Materials listed by Geant4:
  - air
  - aluminum
  - beryllium
  - copper
  - kapton
  - lead
  - lithium
  - plexiglass
  - silicon
  - germanium
  - tungsten
  - gallium arsenide

  - cadmium telluride
  - nickel
  - gold
  - titanium

- Composite or custom materials:

  - carbon fiber
  - epoxy
  - fused silica
  - PCB G-10
  - paper (cellulose)
  - solder
  - polystyrene
  - ppo foam
  - cadmium zinc telluride
  - diamond
  - silicon carbide
  - titanium grade 5
  - vacuum

Furthermore, this module can automatically load any material defined in the Geant4 material database [89]. This comprises both simple materials and pre-defined NIST compounds. It should be noted that when loading a material from the Geant4 material database, the name comparison is case sensitive. Names can be provided with or without `G4_` prefix.

8.18.2 Dependencies

This module requires an installation of Geant4.

8.18.3 Parameters

- `world_material` : Material of the world, should either be **air** or **vacuum**. Defaults to **air** if not specified.
- `world_margin_percentage` : Percentage of the world size to add to every dimension compared to the internally calculated minimum world size. Defaults to 0.1, thus 10%.
- `world_minimum_margin` : Minimum absolute margin to add to all sides of the internally calculated minimum world size. Defaults to zero for all axis, thus not requiring any minimum margin.
- `log_level_g4cerr`: Target logging level for Geant4 messages from the G4cerr (error) stream. Defaults to `WARNING`.
- `log_level_g4cout`: Target logging level for Geant4 messages from the G4cout stream. Defaults to `TRACE`.

8.18.4 Usage

To create a Geant4 geometry using vacuum as world material and with always exactly one meter added to the minimum world size in every dimension, the following configuration could be used:

```
[GeometryBuilderGeant4]
world_material = "vacuum"
world_margin_percentage = 0
world_minimum_margin = 1m 1m 1m
```

8.19 InducedTransfer

8.19.1 Description

Combines individual sets of propagated charges together to a set of charges on the sensor pixels by calculating the total induced charge during their drift on neighboring pixels by calculating the difference in weighting potential. This module requires a propagation of both electrons and holes in order to produce sensible results and only works in the presence of a weighting potential.

The induced charge on neighboring pixel implants is defined the Shockley-Ramo theorem [25, 26] as the difference in weighting potential between the end position x_{final} retrieved from the `PropagatedCharge` and the initial position $x_{initial}$ of the charge carrier obtained from the `DepositedCharge` object in the history. The total induced charge is calculated by multiplying the potential difference with the charge of the carrier, viz.

$$Q_n^{ind} = \int_{t_{initial}}^{t_{final}} I_n^{ind} = q\left(\phi(x_{final}) - \phi(x_{initial})\right)$$

The resulting induced charge is summed for all propagated charge carriers and returned as a `PixelCharge` object. The number of neighboring pixels taken into account can be configured using the `distance` parameter.

8.19.2 Parameters

- `distance`: Maximum distance of pixels to be considered for current induction, calculated from the pixel the charge carrier under investigation is below. A distance of `1` for example means that the induced current for the closest pixel plus all neighbors is calculated. It should be noted that the time required for simulating a single event depends almost linearly on the number of pixels the induced charge is calculated for. Usually, for Cartesian sensors a 3x3 grid (9 pixels, distance 1) should suffice since the weighting potential at a distance of more than one pixel pitch often is small enough to be neglected while the simulation time is almost tripled for `distance = 2` (5x5 grid, 25 pixels). To just calculate the induced current in the one pixel the charge carrier is below, `distance = 0` can be used. Defaults to `1`.

8.19.3 Usage

```
[InducedTransfer]
distance = 1
```

8.20 LCIOWriter

8.20.1 Description

Writes pixel hit data to LCIO file, compatible with the EUTelescope analysis framework [90].

If the `geometry_file` parameter is set to a non-empty string, a matching GEAR XML file is created from the simulated detector geometry and written to the simulation output directory. This GEAR file can be used with EUTelescope directly to reconstruct particle trajectories.

Optionally, if `dump_mc_truth` is set to true, this module will create Monte Carlo truth collections in the output LCIO file.

8.20.2 Parameters

- `file_name`: name of the LCIO file to write, relative to the output directory of the framework. The extension **.slcio** should be added. Defaults to `output.slcio`.
- `geometry_file` : name of the output GEAR file to write the EUTelescope geometry description to. Defaults to `allpix_squared_gear.xml`
- `pixel_type`: EUtelescope pixel type to create. Options: EUTelSimpleSparsePixelDefault = 1, EUTelGenericSparsePixel = 2, EUTelTimepix3SparsePixel = 5 (Default: EUTelGenericSparsePixel)
- `detector_name`: Detector name written to the run header. Default: "EUTelescope"
- `dump_mc_truth`: Export the Monte Carlo truth data. Default: "false"

Only one of the following options must be used, if none is specified `output_collection_name` will be used with its default value.

- `output_collection_name`: Name of the LCIO collection containing the pixel data. Detectors will be assigned ascending sensor ids starting with 0. Default: "zsdata_m26"
- `detector_assignment`: A matrix with three entries each row: `["detector_name", "output_collection", "sensor_id"]`, one row for each detector. This allows to assign different output collections and sensor ids within the same set-up. `detector_name` is the detector's name as specified in the geometry file, `output_collection` the desired LCIO collection name and `sensor_id` the id used in the exported LCIO data. Sensor ids must be unique.

If only one detector is present in the `detector_assignment`, the value has to be encapsulated in extra quotes, i.e. `[["mydetector", "zsdata_test", "123"]]`.

8.20.3 Usage

```
[LCIOWriter]
file_name = "run000123-converter.slcio"
```

Using the `detector_assignment` to write into two collections and assigning sensor id 20 to the device under test. Further, exporting the Monte Carlo truth data and writing the GEAR file:

```
[LCIOWriter]
file_name = "run000123-converter.slcio"
geometry_file = "run000123-gear.xml"
dump_mc_truth = true
detector_assignment = ["telescope0", "zsdata_m26", "0"], ["mydut",
 ↪  "zsdata_dut", "20"], ["telescope1", "zsdata_m26", "1"]
```

8.21 MagneticFieldReader

8.21.1 Description

Unique module, adds a magnetic field to the full volume, including the active sensors. By default, the magnetic field is turned off.

The magnetic field reader only provides constant magnetic fields, read in as a three-dimensional vector. The magnetic field is forwarded to the GeometryManager, enabling the magnetic field for the particle propagation via Geant4, as well as to all detectors for enabling a Lorentz drift during the charge propagation.

8.21.2 Parameters

- `model` : Type of the magnetic field model, currently only **constant** possible.
- `magnetic_field` : Vector describing the magnetic field.

8.21.3 Usage

An example is given below

```
[MagneticFieldReader]
model = "constant"
magnetic_field = 500mT 3.8T 0T
```

8.22 ProjectionPropagation

8.22.1 Description

The module projects the deposited electrons (or holes) to the sensor surface and applies a randomized, simplified diffusion. It can be used to save computing time at the cost of precision.

The diffusion of the charge carriers is realized by placing sets of a configurable number of electrons in positions drawn as a random number from a two-dimensional Gaussian distribution around the projected position at the sensor surface. The diffusion width is based on an approximation of the drift time, using an analytical approximation for the integral of the mobility in a linear electric field. Here, the charge carrier mobility parametrization of Jacoboni [43] is used. The integral is calculated as follows, with $\mu_0 = V_m/E_c$:

$$\begin{aligned} t &= \int \frac{1}{v} ds \\ &= \int \frac{1}{\mu(s)E(s)} ds \\ &= \int \frac{\left(1 + \left(\frac{E(S)}{E_c}\right)^{\beta}\right)^{1/\beta}}{\mu_0 E(s)} ds \end{aligned}$$

Here, β is set to 1, inducing systematic errors less than 10%, depending on the sensor temperature configured. With the linear approximation to the electric field as $E(s) = ks + E_0$ it is

$$\begin{aligned} t &= \frac{1}{\mu_0} \int \left(\frac{1}{E(s)} + \frac{1}{E_c}\right) ds \\ &= \frac{1}{\mu_0} \int \left(\frac{1}{ks + E_0} + \frac{1}{E_c}\right) ds \\ &= \frac{1}{\mu_0} \left[\frac{\ln(ks + E_0)}{k} + \frac{s}{E_c}\right]_a^b \\ &= \frac{1}{\mu_0} \left[\frac{\ln(E(s))}{k} + \frac{s}{E_c}\right]_a^b \end{aligned}$$

Since the approximation of the drift time assumes a linear electric field, this module cannot be used with any other electric field configuration.

Depending on the parameter `diffuse_deposit`, deposited charge carriers in a sensor region without electric field are either not propagated, or a single, three-dimensional diffusion step prior to the propagation of these charge carriers, corresponding to the `integration_time` is enabled. Charge carriers diffusing into the electric field will be placed at the border between the undepleted and the depleted regions with the corresponding offset in time and then be propagated to the sensor surface.

The charge carrier lifetime can be simulated using the doping concentration of the sensor. The recombination model is selected via the `recombination_model` parameter,

the default value `none` is equivalent to not simulating finite lifetimes. This feature can only be enabled if a doping profile has been loaded for the respective detector using the DopingProfileReader module. This module only supports doping profiles of type **constant**. The doping-dependent charge carrier lifetime is determined once and the survival probability is calculated by drawing a random number from an uniform distribution with $0 \leq r \leq 1$ and comparing it to the expression t/τ, where t is the total propagation time of the charge carrier to the sensor surface. Charge carriers which would recombine before reaching the surface are removed from the simulation.

Lorentz drift in a magnetic field is not supported. Hence, in order to use this module with a magnetic field present, the parameter `ignore_magnetic_field` can be set.

8.22.2 Parameters

- `temperature`: Temperature in the sensitive device, used to estimate the diffusion constant and therefore the width of the diffusion distribution.
- `recombination_model`: Charge carrier lifetime model to be used for the propagation. Defaults to `none`, a list of available models can be found in the documentation. This feature requires a doping concentration to be present for the detector.
- `charge_per_step`: Maximum number of electrons placed for which the randomized diffusion is calculated together, i.e. they are placed at the same position. Defaults to 10.
- `max_charge_groups`: Maximum number of charge groups to propagate from a single deposit point. Temporarily increases the value of `charge_per_step` to reduce the number of propagated groups if the deposit is larger than the value `max_charge_groups*charge_per_step`, thus reducing the negative performance impact of unexpectedly large deposits. The default value is 1000 charge groups. If it is set to 0, there is no upper limit on the number of charge groups propagated.
- `propagate_holes`: If set to `true`, holes are propagated instead of electrons. Defaults to `false`. Only one carrier type can be selected since all charges are propagated towards the implants.
- `ignore_magnetic_field`: Enables the usage of this module with a magnetic field present, resulting in an unphysical propagation w/o Lorentz drift. Defaults to false.
- `integration_time` : Time within which charge carriers are propagated. If the total drift time exceeds, the respective carriers are ignored and do not contribute to the signal. Defaults to the LHC bunch crossing time of 25ns.
- `diffuse_deposit`: Enables a diffusion prior to the propagation for charge carriers deposited in a region without electric field. Defaults to `false`.

8.22.3 Plotting parameters

- `output_plots` : Determines if simple output plots should be generated for a monitoring of the simulation flow. Disabled by default.
- `output_linegraphs` : Determines if line graphs should be generated for every event. This causes a significant slow down of the simulation, it is not recommended to enable this option for runs with more than a couple of events. Disabled by default.

- `output_plots_theta` : Viewpoint angle of the 3D animation and the 3D line graph around the world X-axis. Defaults to zero.
- `output_plots_phi` : Viewpoint angle of the 3D animation and the 3D line graph around the world Z-axis. Defaults to zero.
- `output_plots_use_pixel_units` : Determines if the plots should use pixels as unit instead of metric length scales. Defaults to false (thus using the metric system).
- `output_plots_use_equal_scaling` : Determines if the plots should be produced with equal distance scales on every axis (also if this implies that some points will fall out of the graph). Defaults to true.
- `output_plots_align_pixels` : Determines if the plot should be aligned on pixels, defaults to false. If enabled the start and the end of the axis will be at the split point between pixels.

8.22.4 Usage

```
[ProjectionPropagation]
temperature = 293K
charge_per_step = 10
output_plots = 1
```

8.23 PulseTransfer

8.23.1 Description

This module combines propagated charges into pulses at individual pixel implants. It works in two different modes.

If the propagated charges provide pulse information themselves, e.g. generated by the TransientPropagation module, these pulses are summed for each pixel implant.

If the propagated charges do not contain pulse information, pulses are formed using the charge carrier arrival times at the pixel implants. This necessitates the configuration of the time granularity via the `timestep` parameter as well as the region from which charge carriers are accepted via `max_depth_distance`. It should be noted that this does not represent a time-resolved simulation of the signal formation but can only serve as approximation. Furthermore, by the restriction to the implant regions by enabling `collect_from_implant`, only the charge carrier type collected at the implants is taken into account. In case no implants are defined, charge carriers are collected from the pixel surface and the parameter `max_depth_distance` can be used to control the depth from which charge carriers are taken into account.

Combines individual induced charge pulses generated by propagated charges to one total pulse per pixel. This prepares the pulse for processing in the front-end electronics.

Pulse graph for every pixel seeing a signal is generated if `output_pulsegraphs` is enabled. One graph depicts the induced charge per time step of the simulation, i.e. the current, while the second graph shows the accumulated charge since the beginning of the event. A third graph provides the absolute induced charge per time, disregarding the polarity of the respective signal. It should be noted that generating per-pixel pulses will generate

several pulse graphs per event and might result in a slow-down of the simulation process as well as a large module root file.

8.23.2 Parameters

- `output_plots` : Determines if simple output plots such as the total and per-pixel induced charge should be generated for a monitoring of the simulation flow. Disabled by default.
- `output_plots_scale` : Set the x-axis scale of the output histograms, defaults to 30ke.
- `output_plots_bins` : Set the number of bins for the output histograms, defaults to 100.
- `output_pulsegraphs`: Determines if pulse graphs should be generated for every event. This creates several graphs per event, depending on how many pixels see a signal, and can slow down the simulation. It is not recommended to enable this option for runs with more than a couple of events. Disabled by default.
- `timestep`: Time step for the pulse to be generated from charge carrier arrival times. Only used if no pulse information is available for the propagated charge object. Default value is 0.01ns.
- `max_depth_distance` : Maximum distance in depth, i.e. normal to the sensor surface at the implant side, for a propagated charge to be taken into account in case the detector has no implants defined. Only used if no pulse information is available for the propagated charge object. Defaults to `5um`.
- `collect_from_implant`: Only consider charge carriers within the implant region of the respective detector instead of the full surface of the sensor. Only used if no pulse information is available for the propagated charge object. Should only be used with non-linear electric fields and defaults to `false`.

8.23.3 Usage

The default configuration is equal to the following:

```
[PulseTransfer]
```

8.24 RCEWriter

8.24.1 Description

Reads in the PixelHit messages and saves them in the RCE format, appropriate for the Proteus telescope reconstruction software [91]. An event tree and a sensor tree and their branches are initialized in the module's `initialize()` method. The event tree is initialized with the appropriate branches, while a sensor tree is created for each detector and the branches initialized from a struct storing the tree and branch information for every sensor. Initially, the program loops over all PixelHit messages and then over all the hits within the message, and writes data to the tree branches in the RCE format. If there are no hits, the event is saved with nHits = 0, with the other fields empty.

8.24.2 Parameters

- `file_name` : Name of the data file to create, relative to the output directory of the framework. The file extension `.root` will be appended if not present. Defaults to `rce-data.root`.
- `device_file` : Name of the output device file in the Proteus toml format. The file extension `.toml` will be appended if not present. Defaults to `device.toml`.
- `geometry_file` : Name of the output geometry file in the Proteus toml format. The file extension `.toml` will be appended if not present. Defaults to `geometry.toml`.

8.24.3 Usage

To create the default file an instantiation without arguments can be placed at the end of the main configuration:

```
[RCEWriter]
```

8.25 ROOTObjectReader

8.25.1 Description

Converts all object data stored in the ROOT data file produced by the ROOTObjectWriter module back in to messages (see the description of ROOTObjectWriter for more information about the format). Reads all trees defined in the data file that contain Allpix objects. Creates a message from the objects in the tree for every event.

If the requested number of events for the run is less than the number of events the data file contains, all additional events in the file are skipped. If more events than available are requested, a warning is displayed and the other events of the run are skipped.

Currently it is not yet possible to exclude objects from being read. In case not all objects should be converted to messages, these objects need to be removed from the file before the simulation is started.

8.25.2 Parameters

- `file_name` : Location of the ROOT file containing the trees with the object data. The file extension `.root` will be appended if not present.
- `include` : Array of object names (without `allpix::` prefix) to be read from the ROOT trees, all other object names are ignored (cannot be used simultaneously with the *exclude* parameter).
- `exclude`: Array of object names (without `allpix::` prefix) not to be read from the ROOT trees (cannot be used simultaneously with the *include* parameter).
- `ignore_seed_mismatch`: If set to true, a mismatch between the core random seed in the configuration file and the input data is ignored, otherwise an exception is thrown. This also covers the case when the core random seed in the configuration file is missing. Default is set to false.

8.25.3 Usage

This module should be placed at the beginning of the main configuration. An example to read only PixelCharge and PixelHit objects from the file *data.root* is:

```
[ROOTObjectReader]
file_name = "data.root"
include = "PixelCharge", "PixelHit"
```

8.26 ROOTObjectWriter

8.26.1 Description

Reads all messages dispatched by the framework that contain Allpix objects. Every message contains a vector of objects, which is converted to a vector to pointers of the object base class. The first time a new type of object is received, a new tree is created bearing the class name of this object. For every combination of detector and message name, a new branch is created within this tree. A leaf is automatically created for every member of the object. The vector of objects is then written to the file for every event it is dispatched, saving an empty vector if an event does not include the specific object.

If the same type of messages is dispatched multiple times, it is combined and written to the same tree. Thus, the information that they were separate messages is lost. It is also currently not possible to limit the data that is written to file. If only a subset of the objects is needed, the rest of the data should be discarded afterwards.

The event number and the event seed for the random number generator are written to a tree named Event.

In addition to the objects, both the configuration and the geometry setup are written to the ROOT file. The main configuration file is copied directly and all key/value pairs are written to a directory *config* in a subdirectory with the name of the corresponding module. All the detectors are written to a subdirectory with the name of the detector in the top directory *detectors*. Every detector contains the position, rotation matrix and the detector model (with all key/value pairs stored in a similar way as the main configuration).

8.26.2 Parameters

- `file_name` : Name of the data file to create, relative to the output directory of the framework. The file extension `.root` will be appended if not present.
- `include` : Array of object names (without `allpix::` prefix) to write to the ROOT trees, all other object names are ignored (cannot be used together simultaneously with the *exclude* parameter).
- `exclude`: Array of object names (without `allpix::` prefix) that are not written to the ROOT trees (cannot be used together simultaneously with the *include* parameter).

8.26.3 Usage

To create the default file (with the name *data.root*) containing trees for all objects except for PropagatedCharges, the following configuration can be placed at the end of the main configuration:

```
[ROOTObjectWriter]
exclude = "PropagatedCharge"
```

To read back a value of the configuration (here the Allpix Squared version used in the simulation), the following command can be executed on the output file, here named *data.root*:

```
root -l data.root -e '(*(string*)_file0->GetDirectory("config/Allpix")-
 ↪  >GetKey("version")->ReadObj())'
```

8.27 SimpleTransfer

8.27.1 Description

Combines individual sets of propagated charges together to a set of charges on the sensor pixels and thus prepares them for processing by the detector front-end electronics. The module does a simple direct mapping to the nearest pixel, ignoring propagated charges that are too far away from the implants or outside the pixel grid. Timing information for the pixel charges is currently not yet produced, but can be fetched from the linked propagated charges.

If one or more implants are defined for the respective detector model, the `collect_from_implant` option can be turned on in order to only pick charge carriers from the implant volume and ignore everything outside this region. Only charge carriers from front-side implants are collected, charge carriers on back-side implants are dropped. Since this will lead to unexpected and undesired behavior when using linear electric fields, this option can only be used when using fields with an x/y dependence (i.e. field maps imported from TCAD). In case no implants are defined, charge carriers are collected from the pixel surface and the parameter `max_depth_distance` can be used to control the depth from which charge carriers are taken into account.

A histogram of charge carrier arrival times is generated if `output_plots` is enabled. The range and granularity of this plot can be configured.

8.27.2 Parameters

- `max_depth_distance` : Maximum distance in depth, i.e. normal to the sensor surface at the implant side, for a propagated charge to be taken into account in case the detector has no implants defined. Defaults to `5um`.
- `collect_from_implant`: Only consider charge carriers within the implant region of the respective detector instead of the full surface of the sensor. Should only be used with non-linear electric fields and defaults to `false`.

- `output_plots`: Determines if output plots should be generated. Disabled by default.
- `output_plots_step`: Bin size of the arrival time histogram in units of time. Defaults to `0.1ns`.
- `output_plots_range`: Total range of the arrival time histogram. Defaults to `100ns`.

8.27.3 Usage

For a typical simulation, a *max_depth_distance* a few micro meters should be sufficient, leading to the following configuration:

```
[SimpleTransfer]
max_depth_distance = 5um
```

8.28 TextWriter

Description

This module allows to write any object from the simulation to a plain ASCII text file. It reads all messages dispatched by the framework containing Allpix objects. The data content of each message is printed into the text file, while events are separated by an event header:

```
=== <event number> ===
```

and individual detectors by the detector marker:

```
--- <detector name> ---
```

The `include` and `exclude` parameters can be used to restrict the objects written to file to a certain type.

Parameters

- `file_name` : Name of the data file to create, relative to the output directory of the framework. The file extension `.txt` will be appended if not present.
- `include` : Array of object names (without `allpix::` prefix) to write to the ASCII text file, all other object names are ignored (cannot be used together simultaneously with the *exclude* parameter).
- `exclude`: Array of object names (without `allpix::` prefix) that are not written to the ASCII text file (cannot be used together simultaneously with the *include* parameter).

Usage

To create the default file (with the name *data.txt*) containing entries only for PixelHit objects, the following configuration can be placed at the end of the main configuration:

```
[TextWriter]
include = "PixelHit"
```

8.29 TransientPropagation

8.29.1 Description

Simulates the transport of electrons and holes through the sensitive sensor volume of the detector. It allows to propagate sets of charge carriers together in order to speed up the simulation while maintaining the required accuracy. The propagation process for these sets is fully independent and no interaction is simulated. The maximum size of the set of propagated charges and thus the accuracy of the propagation can be controlled via the `charge_per_step` parameter. The maximum number of charge groups to be propagated for a single deposit position can be controlled via the `max_charge_groups` parameter.

The propagation consists of a combination of drift and diffusion simulation. The drift is calculated using the charge carrier velocity derived from the charge carrier mobility and the magnetic field via a calculation of the Lorentz drift. The mobility model can be chosen using the `mobility_model` parameter, and a list of available models can be found in the user manual. This module implements charge multiplication by impact ionization. The multiplication model can be chosen using the `multiplication_model` parameter, the list of available models can be found in the user manual. By default, the model defaults to `none` and impact ionization is switched off, generating unity gain. To simulate impact ionization, the number of newly generated electron-hole pairs is calculated for every propagation step and every charge carrier in the group, based on drawing a random number from a geometric distribution. This represents a stepwise approach to the avalanche generation process. The charge of a charge group is increased by the number of impact ionization processes per step and opposite-type charge carriers are generated at the end of the step.

A fourth-order Runge-Kutta-Fehlberg method [22] is used to integrate the particle motion through the electric and magnetic fields. After every Runge-Kutta step, the diffusion is accounted for by applying an offset drawn from a Gaussian distribution calculated from the Einstein relation

$$\sigma = \sqrt{\frac{2k_bT}{e}\mu t}$$

using the carrier mobility μ, the temperature T and the time step t. The propagation stops when the set of charges reaches any surface of the sensor.

The charge transport is parameterized in time and the time step each simulation step takes can be configured. For each step, the induced charge on the neighboring pixel implants is calculated via the Shockley-Ramo theorem [25, 26] by taking the difference in weighting potential between the current position x_1 and the previous position x_0 of the charge carrier

$Q_n^{ind} = \int_{t_0}^{t_1} I_n^{ind} = q\left(\phi(x_1) - \phi(x_0)\right)$

and multiplying it with the charge. The resulting pulses are stored for every set of charge carriers individually and need to be combined for each pixel using a transfer module.

The charge carrier lifetime can be simulated using the doping concentration of the sensor. The recombination model is selected via the `recombination_model` parameter, the default value `none` is equivalent to not simulating finite lifetimes. This feature can only be enabled if a doping profile has been loaded for the respective detector using the DopingProfileReader module. In each step, the doping-dependent charge carrier lifetime is determined, from which a survival probability is calculated. The survival probability is calculated at each step of the propagation by drawing a random number from an uniform distribution with $0 \leq r \leq 1$ and comparing it to the expression dt/τ, where dt is the time step of the last charge carrier movement.

Trapping of charge carriers can be enabled by setting a trapping model via the parameter `trapping_model`. The default value is `none`, corresponding to no charge carrier trapping being simulated. All models require the 1MeV-neutron equivalent fluence to be set via the parameter `fluence`. Some models include temperature-dependent scaling of trapping probabilities, and the corresponding temperature is taken from the `temperature` parameter. The trapping probability is calculated at each step of the propagation by drawing a random number from an uniform distribution with $0 \leq r \leq 1$ and comparing it to the expression $1 - e^{-dt/\tau_{eff}}$, where dt is the time step of the last charge carrier movement and τ_{eff} the effective trapping time constant. A list of available models can be found in the user manual.

Detrapping of charge carriers can be enabled by setting a detrapping model via the parameter `detrapping_model`. The default value is `none`, corresponding to no charge carrier detrapping being simulated. A list of available models can be found in the user manual.

The module can produces a variety of plots such as total integrated charge plots as well as histograms on the step length and observed potential differences. Furthermore, the module can generate a 3D line plot of the path of all separately propagated charge carrier sets from their point of deposition to the end of their drift, with nearby paths having different colors. In this coloring scheme, electrons are marked in blue colors, while holes are presented in different shades of orange. In addition, a 3D GIF animation for the drift of all individual sets of charges (with the size of the point proportional to the number of charges in the set) can be produced. Finally, the module produces 2D contour animations in all the planes normal to the X, Y and Z axis, showing the concentration flow in the sensor. It should be noted that generating the animations is time-consuming and should be switched off even when investigating drift behavior.

8.29.2 Parameters

- `temperature`: Temperature of the sensitive device, used to estimate the diffusion constant and therefore the strength of the diffusion. Defaults to room temperature (293.15K).
- `mobility_model`: Charge carrier mobility model to be used for the propagation. Defaults to `jacoboni`, a list of available models can be found in the documentation.

- `recombination_model`: Charge carrier lifetime model to be used for the propagation. Defaults to `none`, a list of available models can be found in the documentation. This feature requires a doping concentration to be present for the detector.
- `trapping_model`: Model for simulating charge carrier trapping from radiation-induced damage. Defaults to `none`, a list of available models can be found in the documentation. All models require explicitly setting a fluence parameter.
- `detrapping_model`: Model for simulating charge carrier detrapping from radiation-induced damage. Defaults to `none`, a list of available models can be found in the documentation.
- `fluence`: 1MeV-neutron equivalent fluence the sensor has been exposed to.
- `charge_per_step`: Maximum number of charge carriers to propagate together. Divides the total number of deposited charge carriers at a specific point into sets of this number of charge carriers and a set with the remaining charge carriers. A value of 10 charges per step is used by default if this value is not specified.
- `max_charge_groups`: Maximum number of charge groups to propagate from a single deposit point. Temporarily increases the value of `charge_per_step` to reduce the number of propagated groups if the deposit is larger than the value `max_charge_groups*charge_per_step`, thus reducing the negative performance impact of unexpectedly large deposits. The default value is 1000 charge groups. If it is set to 0, there is no upper limit on the number of charge groups propagated.
- `timestep`: Time step for the Runge-Kutta integration, representing the granularity with which the induced charge is calculated. Default value is 0.01ns.
- `integration_time`: Time within which charge carriers are propagated. After exceeding this time, no further propagation is performed for the respective carriers. Defaults to the LHC bunch crossing time of 25ns.
- `distance`: Maximum distance of pixels to be considered for current induction, calculated from the pixel the charge carrier under investigation is below. A distance of `1` for example means that the induced current for the closest pixel plus all neighbors is calculated. It should be noted that the time required for simulating a single event depends almost linearly on the number of pixels the induced charge is calculated for. Usually, for Cartesian sensors a 3x3 grid (9 pixels, distance 1) should suffice since the weighting potential at a distance of more than one pixel pitch often is small enough to be neglected while the simulation time is almost tripled for `distance = 2` (5x5 grid, 25 pixels). To just calculate the induced current in the one pixel the charge carrier is below, `distance = 0` can be used. Defaults to `1`.
- `ignore_magnetic_field`: The magnetic field, if present, is ignored for this module. Defaults to false.
- `multiplication_model`: Model used to calculate impact ionization parameters and charge multiplication. Defaults to `none` which corresponds to unity gain, a list of available models can be found in the documentation.
- `multiplication_threshold`: Threshold field above which charge multiplication is calculated. Defaults to `100kV/cm`.
- `max_multiplication_level`: Maximum level depth of the generated impact ionization charge multiplication shower after which the generation of further multiplication charge carrier levels is prohibited. This number represents the maximum number of daughter charge carrier groups that can be produced by one initial charge carrier group. This does not concern the size of the charge group itself but solely the level of generation. If a group generates a secondary group through impact ionization, the depth is `1`. If this secondary group again creates charge carriers when propagating,

the level is 2 and so on. The default value is 5.

8.29.3 Plotting parameters

- output_plots : Determines if simple output plots should be generated for a monitoring of the simulation flow. Disabled by default.
- output_linegraphs : Determines if line graphs should be generated for every event. This causes a significant slow down of the simulation, it is not recommended to enable this option for runs with more than a couple of events. Disabled by default.
- output_linegraphs_collected : Determine whether to also generate line graphs *only* for charge carriers that have reached the implant side within the allotted integration time. Defaults to false. This requires output_linegraphs to be active.
- output_linegraphs_recombined : Boolean flag to select whether line graphs should also be generated only from charge carriers that have recombined with the lattice during the integration time. Defaults to false. This requires output_linegraphs to be active.
- output_linegraphs_trapped : Boolean flag to select whether line graphs should also be generated only from charge carriers that have been trapped during their motion through the sensor. Defaults to false. This requires output_linegraphs to be active.
- output_plots_step : Timestep to use between two points plotted. Indirectly determines the amount of points plotted. Defaults to *timestep_max* if not explicitly specified.
- output_plots_theta : Viewpoint angle of the 3D animation and the 3D line graph around the world X-axis. Defaults to zero.
- output_plots_phi : Viewpoint angle of the 3D animation and the 3D line graph around the world Z-axis. Defaults to zero.
- output_plots_use_pixel_units : Determines if the plots should use pixels as unit instead of metric length scales. Defaults to false (thus using the metric system).
- output_plots_use_equal_scaling : Determines if the plots should be produced with equal distance scales on every axis (also if this implies that some points will fall out of the graph). Defaults to true.
- output_plots_align_pixels : Determines if the plot should be aligned on pixels, defaults to false. If enabled the start and the end of the axis will be at the split point between pixels.
- output_animations : In addition to the other output plots, also write a GIF animation of the charges drifting towards the electrodes. This is extremely slow and writing the animation takes a considerable amount of time, therefore defaults to false. This option also requires output_linegraphs to be enabled.
- output_animations_time_scaling : Scaling for the animation used to convert the actual simulation time to the time step in the animation. Defaults to 1.0e9, meaning that every nanosecond of the simulation is equal to an animation step of a single second.
- output_animations_marker_size : Scaling for the markers on the animation, defaults to one. The markers are already internally scaled to the charge of their step, normalized to the maximum charge.

- `output_animations_contour_max_scaling` : Scaling to use for the contour color axis from the theoretical maximum charge at every single plot step. Default is 10, meaning that the maximum of the color scale axis is equal to the total amount of charges divided by ten (values above this are displayed in the same maximum color). Parameter can be used to improve the color scale of the contour plots.
- `output_animations_color_markers`: Determines if colors should be for the markers in the animations, defaults to false.

8.29.4 Usage

```
[TransientPropagation]
temperature = 293K
charge_per_step = 10
output_plots = true
timestep = 0.02ns
```

8.30 VisualizationGeant4

8.30.1 Description

Constructs a viewer to display the constructed Geant4 geometry. The module supports all type of viewers included in Geant4, but the default Qt visualization with the OpenGL viewer is recommended as long as the installed Geant4 version supports it. It offers the best visualization experience.

The module allows for changing a variety of parameters to control the output visualization both for the different detector components and the particle beam.

Both detectors and passive materials will be displayed. If the material of a passive material is the same as the material of its `mother_volume`, the passive material will not be shown in the visualization. In the case that the material is the same as the material of the world frame, the material will have a white color instead of the default blue in the visualization.

This module does not support multithreading and will force the simulation chain to be executed on a single thread when activated.

8.30.2 Dependencies

This module requires an installation of Geant4.

8.30.3 Parameters

- `mode` : Determines the mode of visualization. Options are **gui** which starts a Qt visualization window containing the driver (as long as the chosen driver supports it), **terminal** starts both the visualization viewer and a Geant4 terminal or **none** which only starts the driver itself (and directly closes it if the driver is asynchronous). Defaults to **gui**.
- `driver` : Geant4 driver used to visualize the geometry. All the supported options can be found online [92] and depend on the build options of the Geant4 version used. The default **OGL** should normally be used with the **gui** option if the visualization should be accumulated, otherwise **terminal** is the better option. Other than this, only the **VRML2FILE** driver has been tested. This driver should be used with *mode* equal to **none**. Defaults to the OpenGL driver **OGL**.
- `accumulate` : Determines if all events should be accumulated and displayed at the end, or if only the last event should be kept and directly visualized (if the driver supports it). Defaults to true, thus accumulating events and only displaying the final result.
- `accumulate_time_step` : Time step to sleep between events to allow for time to display if events are not accumulated. Only used if *accumulate* is disabled. Default value is 100ms.
- `simple_view` : Determines if the visualization should be simplified, not displaying the pixel matrix and other parts which are replicated multiple times. Default value is true. This parameter should normally not be changed as it will cause a considerable slowdown of the visualization for a sensor with a typical number of channels.
- `background_color` : Color of the background of the viewer. Defaults to *white*.
- `view_style` : Style to use to display the elements in the geometry. Options are **wireframe** and **surface**. By default, all elements are displayed as solid surface.
- `opacity` : Default opacity percentage of all detector elements, only used if the *view_style* is set to display solid surfaces. The default value is 0.4, giving a moderate amount of opacity.
- `display_trajectories` : Determines if the trajectories of the primary and secondary particles should be displayed. Defaults to *true*.
- `hidden_trajectories` : Determines if the trajectories should be hidden inside the detectors. Only used if the *display_trajectories* is enabled. Default value of the parameter is true.
- `trajectories_color_mode` : Configures the way, trajectories are colored. Options are either **generic** which colors all trajectories in the same way, **charge** which bases the color on the particle's charge, or **particle** which colors the trajectory based on the type of the particle. The default setting is *charge*.
- `trajectories_color` : Color of the trajectories if *trajectories_color_mode* is set to **generic**. Default value is *blue*.
- `trajectories_color_positive` : Visualization color for positively charged particles. Only used if *trajectories_color_mode* is equal to **charge**. Default is *blue*.
- `trajectories_color_neutral` : Visualization color for neutral particles. Only used if *trajectories_color_mode* is equal to **charge**. Default is *green*.
- `trajectories_color_negative` : Visualization color for negatively charged particles. Only used if *trajectories_color_mode* is equal to **charge**. Default is *red*.
- `trajectories_particle_colors` : Array of combinations of particle ID and color

used to determine the particle colors if *trajectories_color_mode* is equal to **particle**. Refer to the Geant4 documentation [81] for details about the IDs of particles.
- `trajectories_draw_step` : Determines if the steps of the trajectories should be plotted. Enabled by default. Only used if *display_trajectories* is enabled.
- `trajectories_draw_step_size` : Size of the markers used to display a trajectory step. Defaults to 2 points. Only used if *trajectories_draw_step* is enabled.
- `trajectories_draw_step_color` : Color of the markers used to display a trajectory step. Default value *red*. Only used if *trajectories_draw_step* is enabled.
- `draw_hits` : Determines if hits in the detector should be displayed. Defaults to false. Option is only useful if Geant4 hits are generated in a module.
- `macro_init` : Optional Geant4 macro to execute during initialization. Whenever possible, the configuration parameters above should be used instead of this option.
- `display_limit` : Sets the `displayListLimit` of the visualization GUI, in case the geometry which has to be loaded is too complex for the GUI to be displayed with the current size Display List. Defaults to 1000000.
- `line_segments` : Sets the number of line segments to approximate a circle with. This parameter can be used when simulating radial detectors to visualize their curved edges with more precision. Defaults to 250.
- `viewpoint_thetaphi` : Sets the theta and phi angles for the viewpoint. Defaults to -70deg 20deg.

8.30.4 Usage

An example configuration providing a wireframe viewing style with the same color for every particle and displaying the result after every event for 2s is provided below:

```
[VisualizationGeant4]
mode = "none"
view_style = "wireframe"
trajectories_color_mode = "generic"
accumulate = 0
accumulate_time_step = 2s
```

To view event number N of a simulation, use a fixed random seed and `number_of_events = 1` and `skip_events = N-1` under the `[Allpix]` header. Note that executing `/run/beamOn 1` in the Qt visualization window or Geant4 terminal does **not** show the next event of the Allpix Squared simulation, but rather a random event.

8.31 WeightingPotentialReader

8.31.1 Description

Adds a weighting potential (Ramo potential) to the detector from one of the supported sources. By default, detectors do not have a weighting potential applied. This module support two types of weighting potentials.

Weighting potential map

Using the **mesh** model of this module allows reading in from a file, e.g. from an electrostatic TCAD simulation. A converter tool for fields from adaptive TCAD meshes is provided with the framework. The map is expected to be symmetric around the reference pixel the weighting potential is calculated for, the size of the field is taken from the file header.

The potential field map needs to be three-dimensional. Otherwise the induced current on neighboring pixels along the missing component will always be exactly the same as the actual pixel under which the charge is present because the same weighting potential is samples - with a two-dimensional field, distances in the third dimension are always zero. This will lead to unphysical results and a multiplication of the total charge. If this behavior is desirable, or e.g. only a single row of pixels is simulated, the check can be omitted by setting `ignore_field_dimensions = true`.

A warning is printed if the size does not correspond to a multiple of the pixel size. While this is not a problem in general, it might hint at a wrong potential map being used.

Generating a weighting potential using TCAD and Allpix Squared The weighting potential is calculated by taking the difference of the electrostatic potentials arising from applying two slightly different bias voltages to one electrode. The steps below outline how to create a weighting potential from TCAD simulations.

1. Produce two TCAD fields that differ slightly in one collection electrode bias voltage, for instance for 0.1 V or 0.01 V, with all the other electrodes grounded. Export the two resulting electrostatic potentials into separate files.
2. Use the `mesh_converter` tool to extract the electrostatic potential from both configurations. Working with the converted files in INIT format is advisable as is human readable and this makes the process of writing a macro for the calculation simpler.
3. Calculate the difference between entries from both files, and divide them by the difference in collection electrode bias voltage in order to normalize them to the range `[0, 1]`.
4. Verify that the values are within a range from 0 to 1, which is the physical range of a weighting potential.
5. Save the resulting file with the same format and import it into Allpix Squared using the `[WeightingPotentialReader]` module and the **mesh** model.

Weighting potential of a pad

When setting the **pad** model, the weighting potential of a pixel in a plane condenser is calculated numerically from first principles, following the procedure described in detail in [27]. It should be noted that this calculation is comparatively **slow and takes about a factor 100 longer** than a lookup from a pre-calculated field map. A tool to generate the field map using the method described herein is provided in the software repository.

The weighting potential is calculated via Green's reciprocity theorem, the integral part of the expression are ignored. In [27] it has been shown that the uncertainty on the weighting potential is smaller than

$|\Delta\phi_w| < \frac{V_w}{8\pi} \frac{w_x w_y}{d^2} \frac{1}{N^2} \frac{z}{d}$,

where N limits the expansion of the series. In this implementation, a value of $N = 100$ is used. Following these calculations, the weighting potential is given by

$\phi_w / V_w = \frac{1}{2\pi} f(x, y, z) - \frac{1}{2\pi} \sum_{n=1}^{N} [f(x, y, 2nd - z) - g_z(x, y, 2nd + z)]$

with

$$f(x, y, u) = \arctan\left(\frac{x_1 y_1}{u\sqrt{x_1^2+y_1^2+u^2}}\right) + \arctan\left(\frac{x_2 y_2}{u\sqrt{x_2^2+y_2^2+u^2}}\right) - \arctan\left(\frac{x_1 y_2}{u\sqrt{x_1^2+y_2^2+u^2}}\right) - \arctan\left(\frac{x_2 y_1}{u\sqrt{x_2^2+y_1^2+u^2}}\right)$$

with $x_{1,2} = x \pm \frac{w_x}{2}$ $y_{1,2} = y \pm \frac{w_y}{2}$. The parameters $w_{x,y}$ indicate the size of the collection electrode (i.e. the implant), V_w is the potential of the electrode and d is the thickness of the sensor.

8.31.2 Parameters

- `model` : Type of the weighting potential model, either **mesh** or **pad**.
- `file_name` : Location of file containing the weighting potential in one of the supported field file formats. Only used if the *model* parameter has the value **mesh**.
- `field_mapping`: Description of the mapping of the field onto the sensor or pixel cell. Possible values are `PIXEL_FULL`, indicating that the map spans the full 2D plane and the field is centered around the pixel center, `PIXEL_HALF_TOP` or `PIXEL_HALF_BOTTOM` indicating that the field only contains only one half-axis along y, `HALF_LEFT` or `HALF_RIGHT` indicating that the field only contains only one half-axis along x, or `PIXEL_QUADRANT_I`, `PIXEL_QUADRANT_II`, `PIXEL_QUADRANT_III`, `PIXEL_QUADRANT_IV` stating that the field only covers the respective quadrant of the 2D pixel plane. In addition, the `PIXEL_FULL_INVERSE` mode allows loading full-plane field maps which are not centered around a pixel cell but the corner between pixels. Only used if the *model* parameter has the value **mesh**.
- `field_scale`: Scaling factor of the weighting potential in x- and y-direction. By default, the scaling factors are set to `{1, 1}` and the field is used with its physical extent stated in the field data file.
- `field_offset`: Offset of the field in x- and y-direction. With this parameter and the mapping mode `SENSOR`, the field can be shifted e.g. by half a pixel pitch to accommodate for fields which have been simulated starting from the pixel center. The shift is applied in positive direction of the respective coordinate. Only used if the *model* parameter has the value **mesh**.
- `ignore_field_dimensions`: If set to true, a wrong dimensionality of the input field is ignored, otherwise an exception is thrown. Defaults to false.
- `output_plots`: Determines if output plots should be generated. Disabled by default.
- `output_plots_steps` : Number of bins along the z-direction for which the weighting potential is evaluated. Defaults to 500 bins and is only used if `output_plots` is enabled.

- `output_plots_position`: 2D Position in x and y at which the weighting potential is evaluated along the z-axis. By default, the potential is plotted for the position in the pixel center, i.e. (0, 0). This parameter only affects the 1D weighting potential histogram. Only used if `output_plots` is enabled.
- `output_plots_zcut`: Position along the sensor z axis at which the 2D x-y weighting potential profile is evaluated. Defaults to `0um`, i.e. the center plane of the sensor.

8.31.3 Usage

An example to add a weighting potential form a field data file to the detector called "dut" is given below.

```
[WeightingPotentialReader]
name = "dut"
model = "mesh"
file_name = "example_weighting_field.apf"
```

9 Examples

This section provides brief descriptions of the example configurations currently provided in the Allpix Squared repository [13]. The examples are listed in alphabetical order.

9.1 ATLAS ITk Petal

This example simulates a double-sided arrangement of radial strip detectors called a "petal". It features a total of 18 detectors (9 on each side) of various types designated as R0 to R5. Such a structure is planned to be deployed in the end-cap regions of the ATLAS Inner Tracker (ITk). The entire petal structure can be viewed by enabling the `VisualizationGeant4` module.

In this simulation example, the detector is hit by a beam of 5.4 GeV electrons. A simple linear electric field model is used. Deposited charge carriers are grouped during the propagation stage by use of the `charge_per_step` parameter. Charge threshold during the digitization phase is set to approximately three times the value of electronic noise.

The `granularity` parameter controls the number of bins for in-pixel histograms. As the typical strip length of several centimeters would, by default, lead to exceedingly large histograms, the `granularity` parameter is set to `80 4`. This corresponds to a typical strip pitch of 80 um in the x direction and to 4 strip rows (in the y direction) of the radial strip detector.

9.2 Capacitive Coupling

This folder contains example files and configuration for the CapacitiveTransfer module.

The `capacitive_coupling.conf` configuration file, as it is, simulates 6 FE-I4b planes (aligned as in a telescope) with a FE-I4b as a device-under-test (DUT) between the 3rd and 4th telescope planes. This geometry is defined in the `ccpd_example_detector .conf` file. The `SimpleTransfer` module is used for the telescope planes while the `CapacitiveTransfer` module is used for the DUT. The DUT is simulated with specific angles, nominal and minimum gaps, obtained from real measurements. The simulation results, regarding the DUT, should present a lower efficiency on the bottom left corner of the DUT due to the increasing gap between the pixels, towards this direction, and consequently a smaller coupling capacitance. The coupling capacitance for each gap is retrieved from the `gap_scan_coupling_sim.root` ROOT file. More information are provided in the `CapacitiveTransfer` module documentation.

The `capacitance_matrix.txt` file contains a generic relative coupling matrix (same as in the configuration file) that can be used to simulate the cross-coupling effects

in parallel CCPDs assemblies. More information on possible configurations of the `CapacitiveTransfer` module are provided in its documentation.

9.3 Corryvreckan Output

This example demonstrates how to simulate a full telescope setup and how to store the simulation in a format readable by the Corryvreckan reconstruction framework.

The setup used in this example is the reference simulation published in the Allpix Squared paper [93]. It consists of six Timepix3 telescope planes [17] featuring planar silicon sensors with a thickness of 300um. In addition, another Timepix3 detector is placed as device under test (DUT) between the upstream arm (three planes) and downstream arm (three planes) of the telescope, with a sensor thickness of 50um. Here, the thickness is directly defined in the geometry file, overwriting the default value from the `timepix` model. All planes are randomly mis-aligned at the beginning of the simulation using the alignment precision keywords:

```
alignment_precision_position = 1mm 1mm 100um
alignment_precision_orientation = 0.2deg 0.2deg 0.2deg
```

The energy deposition module uses Geant4 to replicate the beam conditions found in the CERN SPS North Area beam lines, i.e. a 120GeV Pion beam with a Gaussian width of about 2mm.

The simulation uses different processing paths for the telescope planes and the DUT in order to configure a different electric field, a different granularity for the charge propagation and different settings for the digitization in the front-end. For this, the `type` and `name` keywords are placed in the configuration file in order to assign the respective modules to specific detectors instantiated in the geometry file.

The results for both the telescope planes and the DUT are written to a ROOT output file in the format of the Corryvreckan reconstruction framework using the CorryvreckanWriter module. Here, two additional keys have to be defined; The detector to be used as reference plane in the reconstruction and the DUT as detector to be excluded from the track fits. More information in these detector roles can be found in the Corryvreckan user manual.

9.4 Cosmic Flux

This example illustrates how the `DepositionCosmics` module is used to model the flux of cosmic muons in Allpix Squared. Python analysis code is included to calculate the flux through the detector from the MCParticle objects.

9.4.1 Usage

First change into the example directory. Run the simulation example from here:

```
allpix -c cosmic_flux.conf
```

To analyze the tracks of the MCParticles, issue

```
python analysis/analysis.py -l
 ↪  PATH_TO_ALLPIX_INSTALL/lib/libAllpixObjects.so
```

The library flag is only required when your `allpix-squared/lib` path is not in `LD_LIBRARY_PATH`. Notice that the analysis script relies on the python modules `uncertainties`, `tqdm`, `matplotlib` and `numpy`.

9.5 EUDET Telescope

This example demonstrates the simulation of EUDET-type Beam Telescopes, making use of the `ProjectionPropagation` module and approximated simulation and sensor parameters tuned to measurements.

The simulation setup represents a beam telescope consisting of six Mimosa26 sensors, MAPS sensors with a small depletion zone. The thickness used herein is 50 um.

The particle propagation and charge deposition are performed via the `DepositionGeant4` module, using an electron beam with an energy of 5 GeV. A standard physics list is chosen.

The electric field of the sensor is approximated with a depletion depth of 15 um and a bias voltage of -4 V.

The charge collection in such sensors is heavily influenced by the charge carriers that are not created in, but diffuse into the depleted volume, often with a strong lateral component leading to cluster sizes larger than one. To approximate this behaviour and obtain a realistic detector response, while simultaneously maintaining a low simulation time, the `diffuse_deposit` parameter of the `ProjectionPropagation` module is used. An integration time of 20 ns is used as an approximation, tuned to experimental data.

The digitization parameters correspond to information from the sensor developers.

9.6 EUDET with RD53a DUT

This example is similar to the EUDET-type telescope example but with extra DUTs added to match the DESY testbeam setup with RD53a modules. The setup consists of six telescope planes of MIMOSA26-type (EUDET beam telescope) and two RD53a modules centered in between the telescope arms: DUT0 defined with 50x50um^2 pitch and DUT1 defined with 20x100um^2 pitch. Furthermore, one FEI4 reference plane is added as the last plane as in real testbeam setup.

The goal of this setup is to simulate the performance of RD53a modules with testbeam setup and to study multiple scattering effects with passive and extra material. For this

purpose, a box made of plexiglass is introduced in the geometry, but the user can also try other materials within the same range of radiation length, such as polystyrene or styrofoam.

A linear electric field is applied to all sensors, with the DUTs and the reference plane on a higher bias voltage than the telescope planes. More complex electric fields can be added by the user by altering the configuration of the respective `ElectricFieldReader` modules. The propagation of charge carriers is performed using the `ProjectionPropagation` module for the MIMOSA26 sensors of the EUDET telescope planes, and using the `GenericPropagation` module for the DUT and reference planes. This ensures minimum computing requirements for the telescope planes while providing a more detailed simulation for the detectors of interest.

The `LCIOWriter` module is placed at the end of the simulation chain in order to write the results of the simulation to a file in the LCIO format that can be used as input for the reconstruction software EUTelescope. Here, it is important to assign the name `zsdata` to the respective data for EUTelescope to properly recognize it. In order to reconstruct the simulation with the Corryvreckan framework, the user can replace this module with the `CorryvreckanWriter` module.

9.7 Fast Simulation

This example is a simulation chain optimized for speed. A setup like this is well suited for unirradiated standard planar silicon detectors, where a linear electric field is a good approximation.

The setup consists of six Timepix-type detectors with a sensor thickness of 300um arranged in a telescope-like structure. The charge deposition is performed by Geant4 using a standard physics list (with the EmStandard_opt3 option) suited for tracking detectors. The Geant4 stepping length is chosen rather coarse with 10um.

The detector setup contains the position and orientation of the telescope planes, which are divided into an upstream and downstream arm and are inclined in both X and Y to increase charge sharing. In addition, the alignment precision in position and orientation is specified in order to randomly misalign the setup and allow reconstruction without tracking artifacts from pixel-perfect alignment.

The main speedup compared to other setups comes from the usage of the `ProjectionPropagation` module to simulate the charge carrier propagation. A setting of `charge_per_step = 100` is chosen over the default of 10 charge carriers to further reduce the CPU load. With a sensor thickness of 300um and an most probable energy deposition of more than 20'000 charge carriers, no impact on the precision is to be expected.

Also the exclusion of `DepositedCharge` and `PropagatedCharge` objects from the output trees help in speeding up the simulation and in keeping the output file size low.

9.8 GDML Passive Material

This example demonstrates how to load passive material structures defined in GDML files into the simulation. Two separate GDML files are placed via the detector setup description file and loaded at startup. All contained volumes are added to the simulation

The setup consists of two silicon detectors and the additional volumes from the GDML files. Some of the volumes are placed between the particle beam origin and the detectors in order to have the pions interact with the material.

9.9 Magnetic Field

This example demonstrates the charged particle propagation inside a sensor with a magnetic field applied.

Two CMS Pixel Detector single chip modules are placed in a 3.8 T magnetic field, of which the rear one is turned to 19 deg. This results in mostly 2 pixel clusters in the front sensor due to the Lorentz drift, while the rotation of the second sensor cancels out the Lorentz drift, resulting in mostly 1 pixel clusters.

For better performance, disable the output plots for the `GenericPropagation` module.

9.10 Passive Volume

This example showcases the definition of passive volumes in the geometry file.

The file `example_detector_passive.conf` contains a detector of the type `test`, as well as several passive objects, identified via the key-parameter pair `role = "passive"`. The example shows the three basic objects implemented, while for the volume “sphere1” the “box1” is defined as `mother_volume`. This implies that the sphere is integrated into the box and that the given position (and orientation, if applicable) are interpreted as specifications relative to the position and orientation of the box mother volume.

Optionally, the `VisualizationGeant4` can be used to visualize these objects.

All other modules operate with standard parameters.

9.11 Precise DUT Simulation

This example combines features from the “Fast Simulation” and the “TCAD Field Simulation” examples. The setup consists of six telescope planes of Timepix-type detectors for reference tracks and a device under test (DUT), in this case a CLICpix2 detector, in the center of the telescope between the two arms. The goal of this setup is to demonstrate how to perform a fast simulation on the telescope planes while maintaining a high precision on the DUT.

For this propose, the telescope follows the example of the “fast simulation” and employs a linear electric field and the `ProjectionPropagation` module for charge carrier transport.

To assign this module only to the telescope planes, the `type` keyword is used to restrict the module to instances of Timepix detectors.

For the DUT the `ElectricFieldReader` module providing the TCAD field features the `name` keyword assigning this module instance to the DUT detector only. This named module instance takes precedence over the other instance with the linear electric field. The `GenericPropagation` module also has to be assigned to the DUT because it would otherwise also be instantiated for the Timepix telescope detectors. Here, the `charge_per_step` setting has been reduced to 10 for the DUT since the CLICpix2 prototype features a sensor of 50um thickness and the additional precision might improve the agreement with data.

All further modules in the simulation chain are again unnamed and without type specification since they are supposed to be executed for all detectors likewise.

9.12 Radial Strip Detector

This example simulates a radial strip detector, which features a trapezoidal shape and strips fanning out radially from a common focal point. Such detectors are planned to be deployed in the end-cap regions of the ATLAS Inner Tracker (ITk). This example uses the model of a specific ITk module, called the R0. You can see the radial detector used in this simulation by enabling the `VisualizationGeant4` module.

Detector models for radial strip detectors are implemented in the `RadialStripDetectorModel` class. A radial strip detector model is defined using four parameters for each strip row. In addition, its `geometry` has to be defined as `radial_strip`.

```
type = "monolithic"
geometry = "radial_strip"
number_of_strips = 1026, 1026, 1154, 1154
angular_pitch = 0.19309mrad, 0.19309mrad, 0.17169mrad, 0.17169mrad
inner_pitch = 74.4um, 78.1um, 73.6um, 78.5um
strip_length = 19mm, 24mm, 29mm, 32mm
```

In this simulation example, the detector is hit by a beam of 5.4 GeV electrons. A simple linear electric field model is used. Deposited charge carriers are grouped during the propagation stage by use of the `charge_per_step` parameter. The `CapacitiveTransfer` module handles the simulation of cross-talk by transferring a portion of collected charge to adjacent strips in the same row. Charge threshold during the digitization phase is set to approximately three times the value of electronic noise.

The `granularity` parameter controls the number of bins for in-pixel histograms. As the typical strip length of several centimeters would, by default, lead to exceedingly large histograms, the `granularity` parameter is set to `80 4`. This corresponds to a typical strip pitch of 80 um in the x direction and to 4 strip rows (in the y direction) of the radial strip detector.

9.13 Replay Simulation

This example demonstrates the possibility of reading data files from previous simulation runs and replaying the messages to the framework, dispatching them to modules with altered parameters. In this case, the output of the fast simulation example is reprocessed with a new charge threshold in the digitization step.

Since this example requires input data from another simulation, it has to be executed using the following command:

```
allpix -c replay_simulation.conf -o
 ↪  ROOTObjectReader.file_name=<input_file>
```

where `<input_file>` should be replaced with the absolute path of the data file generated by the fast simulation example. Alternatively, this parameter can be set directly in the configuration file of the example.

The main advantage of replaying a simulation is, that late stages of the simulation chain can be repeatedly executed without having to regenerate the full event. In the present case, only the `PixelCharge` objects, i.e. the charge collected at each amplifier input of the pixel are read from the input file as indicated by the `include` keyword. These objects are then dispatched for every event, and the subsequent modules listening to this object type receive them just as if they have been generated from scratch.

The `DefaultDigitizer` module then performs the digitization of the charges, but this time with a different threshold than in the original "fast simulation" example. Finally, the `ROOTObjectWriter` stores the newly digitized `PixelHit` objects to a new data file.

A quick speed comparison of running the initial fast simulation and re-running the digitization step of the simulation using the replay technique reveals event generation frequencies of about 70 Hz versus 970 Hz, respectively, i.e. a speed-up factor larger than 10 on a single core of a standard Intel CPU.

9.14 Simple Diode

This example simulates an Am241 alpha source using a native Geant4 GPS macro. The source is defined as a disk from which mono-energetic 5.4 MeV alphas are emitted. This approximates the Am241 alpha spectrum. The source emits the alpha particles isotropically.

A diode-type detector is placed below the source, shielded with an additional sheet of paper of 200um thickness with a pinhole in it to let the alpha pass. The goal is to reproduce the aperture effect seen with alpha particles, where the detected spectrum shows a dependency on the pinhole size due to the different path lengths of the alpha particles in the air as a function of the incident angle at the diode. Small pinhole diameters restrict the incidence angles to be more or less vertical, while larger pinhole diameters also allow alphas at larger angles to pass. The resulting longer path length of these particles results in a larger energy loss before reaching the diode detector.

The charge deposition is performed by Geant4 using a standard physics list and a stepping length of 10um. The `ProjectionPropagation` module with a setting of

`charge_per_step = 500` is used to simulate the charge carrier propagation and the simulation result is stored to file. The `model_paths` parameter is set to add this directory to the search path for detector models.

Optionally, the `VisualizationGeant4` can be used to visualize these objects.

9.15 Source Measurement

This example simulates an Iron-55 source using Geant4's radioactive decay simulation. The particle type is set to `Fe55` to use the isotope, the source energy configured as `0eV` for a decay in rest. A point-like particle source is used.

A Medipix-type detector is placed below the source, shielded with an additional sheet of aluminum with a thickness of 8mm. No misalignment is added but the absolute position and orientation of the detector is specified.

The setup of the simulation chain follows the "fast simulation example: The charge deposition is performed by Geant4 using a standard physics list and a stepping length of 10um. The `ProjectionPropagation` module with a setting of `charge_per_step = 100` is used to simulate the charge carrier propagation and the simulation result is stored to file excluding `DepositedCharge` and `PropagatedCharge` objects to keep the output file size low.

9.16 TCAD Field Simulation

This example follows the "fast simulation" example but now replaces the simplified linear electric field with an actual TCAD-simulated electric field. For this reason, the `ProjectionPropagation` module is replaced by `GenericPropagation` as the former only allows for linear fields owing to the simplifications made in the drift calculations.

The setup is unchanged compared to the "fast simulation example" and consists of six Timepix-type detectors with a sensor thickness of 300um arranged in a telescope-like structure, inclined planes for charge sharing, and a defined alignment precision. The charge deposition is also performed by Geant4 with a stepping length of 10um.

Again, `DepositedCharge` and `PropagatedCharge` objects are not written to the output file as information about these objects cannot be accessed in data and thus are rarely used in the analysis of the simulation.

10 Module & Detector Development

This chapter provides a few brief recipes for developing new simulation modules and detector models for the Allpix Squared framework. Before starting the development, the "How to contribute" Section should be consulted for further information on the development process and code contributions.

10.1 Coding and Naming Conventions

The code base of the Allpix Squared is well-documented and follows concise rules on naming schemes and coding conventions. This enables maintaining a high quality of code and ensures maintainability over a longer period of time. In the following, some of the most important conventions are described. In case of doubt, existing code should be used to infer the coding style from.

10.1.1 Naming Schemes

The following coding and naming conventions should be adhered to when writing code which eventually should be merged into the main repository.

- **Namespace**: The `allpix` namespace is to be used for all classes which are part of the framework, nested namespaces may be defined. It is encouraged to make use of `using namespace allpix;` in implementation files only for this namespace. Especially the namespace `std` should always be referred to directly at the function to be called, e.g. `std::string test`. In a few other cases, such as `ROOT::Math`, the `using` directive may be used to improve readability of the code.
- **Class names**: Class names are typeset in CamelCase, starting with a capital letter, e.g. `class ModuleManager`. Every class should provide sensible Doxygen documentation for the class itself as well as for all member functions.
- **Member functions**: Naming conventions are different for public and private class members. Public member function names are typeset as camelCase names without underscores, e.g. `getElectricFieldType()`. Private member functions use lower-case names, separating individual words by an underscore, e.g. `create_detector_modules(...)`. This allows to visually distinguish between public and restricted access when reading code.

  In general, public member function names should follow the `get/set` convention, i.e. functions which retrieve information and alter the state of an object should be marked accordingly. Getter functions should be made `const` where possible to allow usage of constant objects of the respective class.

- **Member variables**: Member variables of classes should always be private and accessed only via respective public member functions. This allows to change the class implementation and its internal members without requiring to rewrite code which accesses them. Member names should be typeset in lower-case letters, a trailing underscore is used to mark them as member variables, e.g. `bool terminate_`. This immediately sets them apart from local variables declared within a function.

10.1.2 Formatting

A set of formatting rules is applied to the code base in order to avoid unnecessary changes from different editors and to maintain readable code. It is vital to follow these rules during development in order to avoid additional changes to the code, just to adhere to the formatting. There are several options to integrate this into the development workflow:

- Many popular editors feature direct integration either with `clang-format` or their own formatting facilities.
- A build target called `make format` is provided if the `clang-format` tool is installed. Running this command before committing code will ensure correct formatting.
- This can be further simplified by installing the *git hook* provided in the directory `/etc/git-hooks/`. A hook is a script called by `git` before a certain action. In this case, it is a pre-commit hook which automatically runs `clang-format` in the background and offers to update the formatting of the code to be committed. It can be installed by calling

  ```
  ./etc/git-hooks/install-hooks.sh
  ```

  once.

The formatting rules are defined in the `.clang-format` file in the repository in machine-readable form (for `clang-format`, that is) but can be summarized as follows:

- The column width should be 125 characters, with a line break afterwards.
- New scopes are indented by four whitespaces, no tab characters are to be used.
- Namespaces are indented just as other code is.
- No spaces should be introduced before parentheses `()`.
- Included header files should be sorted alphabetically.
- The pointer asterisk should be left-aligned, i.e. `int* foo` instead of `int *foo`.

The continuous integration automatically checks if the code adheres to the defined format as described in Section 11.3.

10.2 Building Modules Outside the Framework

Allpix Squared provides CMake modules which allow to build modules for the framework outside the actual code repository. The macros required to build a module are provided through the CMake modules and are automatically included when using the `FIND_PACKAGE(Allpix)` CMake command. By this, modules can easily be moved into and out from the module directory of the framework without requiring changes to its `CMakeLists.txt`.

A minimal CMake setup for compiling and linking external modules to the core and object library of the Allpix Squared framework is the following:

```
CMAKE_MINIMUM_REQUIRED(VERSION 3.6.3 FATAL_ERROR)

FIND_PACKAGE(Allpix 2.2 REQUIRED)

ALLPIX_DETECTOR_MODULE(MODULE_NAME)
ALLPIX_MODULE_SOURCES(${MODULE_NAME} MySimulationModule.cpp)
```

All dependencies of the framework such as ROOT or Boost.Random are automatically added as CMake targets and can be used by the module. The required `CMAKE_CXX_STANDARD` is automatically inferred from the settings used to build the framework. Additional libraries can be linked to the module using the standard CMake command

```
TARGET_LINK_LIBRARIES(${MODULE_NAME} MyExternalLibrary)
```

A more complex CMake structure, suited to host multiple external modules, is provided in a separate repository [94].

In order to load modules which have been compiled and installed in a different location than the ones shipped with the framework itself, the respective search path has to be configured properly in the Allpix Squared main configuration file:

```
[AllPix]
# Library search paths
library_directories = "~/allpix-modules/build", "/opt/apsq-modules"
```

The relevant parameter is described in detail in Section 3.4.

10.3 Implementing a New Module

Owing to its modular structure, the functionality of the Allpix Squared can easily be extended by adding additional modules which can be placed in the simulation chain. Since the framework serves a wide community, modules should be as generic as possible, i.e. not only serve the simulation of a single detector prototype but implement the necessary algorithms such that they are re-usable for other applications. Furthermore, it may be beneficial to split up modules to support the modular design of Allpix Squared.

Before starting the development of a new module, it is essential to carefully read the documentation of the framework module manager which can be found in Section 5.3.

The basic steps to implement a new module, hereafter referred to as `ModuleName`, are the following:

1. Initialization of the code for the new module, using the script `etc/scripts/make_module.sh` in the repository. The script will ask for the name of the model and the type (unique or detector-specific). It creates the directory with a minimal example to get started together with the rough outline of its documentation in `README.md`.

2. Before starting to implement the actual module, it is recommended to update the introductory documentation in `README.md`. No additional documentation has to be provided, as this file is automatically included in the user manual. It should be written in GitLab Flavored Markdown (GLFM) [95], so that formulae can also be included (see the spec entry). The Doxygen documentation in `<ModuleName>.hpp` should also be extended to provide a basic description of the module.

3. Finally, the constructor and `init`, `run` and/or `finalize` methods can be written, depending on the requirements of the new module.

Additional sources of documentation which may be useful during the development of a module include:

- The framework documentation in Chapter 4 for an introduction to the different components of the framework.
- The module documentation in Chapter 8 for a description of the functionality of other modules already implemented, and to look for similar modules which can help during development.
- The Doxygen (core) reference documentation included in the framework [5].
- The latest version of the source code of all modules and the Allpix Squared core itself.

Any module potentially useful for other users should be contributed back to the main repository after is has been validated. It is strongly encouraged to send a merge request through the mechanism provided by the software repository [13].

10.3.1 Files of a Module

Every module directory should at minimum contain the following documents (with `ModuleName` replaced by the name of the module):

- `CMakeLists.txt`: The build script to load the dependencies and define the source files of the library.
- `README.md`: Full documentation of the module.
- `<ModuleName>Module.hpp`: The header file of the module.
- `<ModuleName>Module.cpp`: The implementation file of the module.

These files are discussed in more detail below. By default, all modules added to the `src/modules/` directory will be built automatically by CMake. If a module depends on additional packages which not every user may have installed, one can consider adding the following line to the top of the module's `CMakeLists.txt`:

```
ALLPIX_ENABLE_DEFAULT(OFF)
```

General guidelines and instructions for implementing new modules are provided in Section 10.3.

CMakeLists.txt

Contains the build description of the module with the following components:

1. On the first line either `ALLPIX_DETECTOR_MODULE(MODULE_NAME)` or `ALLPIX_UNIQUE_MODULE (MODULE_NAME)` depending on the type of module defined. The internal name of the module is automatically saved in the variable `${MODULE_NAME}` which should be used as an argument to other functions. Another name can be used by overwriting the variable content, but in the examples below, `${MODULE_NAME}` is used exclusively and is the preferred method of implementation.
2. The following lines should contain the logic to load possible dependencies of the module (below is an example to load Geant4). Only ROOT is automatically included and linked to the module.
3. A line with `ALLPIX_MODULE_SOURCES(${MODULE_NAME} <sources>)` defines the module source files. Here, `sources` should be replaced by a list of all source files relevant to this module.
4. Possible lines to include additional directories and to link libraries for dependencies loaded earlier.
5. A line with `ALLPIX_MODULE_REQUIRE_GEANT4_INTERFACE(${MODULE_NAME})` adds the Geant4 interface library as explained in Section 14.1.
6. A line to register the directory with module tests, for example `tests` as in `{ ALLPIX_MODULE_TESTS(${MODULE_NAME} "tests")`.
7. A line containing `ALLPIX_MODULE_INSTALL(${MODULE_NAME})` to set up the required target for the module to be installed to.

A simple `CMakeLists.txt` for a module named `Test` which requires Geant4 is provided below as an example.

```
# Define module and save name to MODULE_NAME
# Replace by ALLPIX_DETECTOR_MODULE(MODULE_NAME) to define a detector
 ↪  module
ALLPIX_UNIQUE_MODULE(MODULE_NAME)

# Load Geant4
FIND_PACKAGE(Geant4 REQUIRED)

# Add the sources for this module
```

```
ALLPIX_MODULE_SOURCES(${MODULE_NAME}
    TestModule.cpp
)

# Add Geant4 to the include directories
TARGET_INCLUDE_DIRECTORIES(${MODULE_NAME} SYSTEM PRIVATE
 ↪  ${Geant4_INCLUDE_DIRS})

# Allpix Geant4 interface is required for this module
ALLPIX_MODULE_REQUIRE_GEANT4_INTERFACE(${MODULE_NAME})

# Link the Geant4 libraries to the module library
TARGET_LINK_LIBRARIES(${MODULE_NAME} ${Geant4_LIBRARIES})

# Register module tests
ALLPIX_MODULE_TESTS(${MODULE_NAME} "tests")

# Provide standard install target
ALLPIX_MODULE_INSTALL(${MODULE_NAME})
```

README.md

The `README.md` serves as the documentation for the module and should be written in GitLab Flavored Markdown (GLFM) [95]. It is automatically included in the user manual in Chapter 8.

The `README.md` should follow the structure indicated in the `README.md` file of the `DummyModule` in `src/modules/Dummy`, and should contain at least the following sections:

- A YAML header with the name of the module (`title`), a short description of the module (`description`) the maintainer (`module_maintainer`) and status (`module_status`) of the module.

  If the module is working and well-tested, the status of the module should be `Functional`. By default, new modules are given the status `Immature`. The maintainer should mention the full name of the module maintainer, with their email address in parentheses. A minimal header is therefore:

  ```
  title: "ModuleName"
  description: "Some short description"
  module_maintainer: "John Doe (<john.doe@example.com>)"
  module_status: "Functional"
  ```

  In addition, the input (`module_input`) and output (`module_output`) objects of the module should be given as well.

- An H2-size section named **Description**, containing a short description of the module.

- An H2-size section named **Parameters**, with all available configuration parameters of the module. The parameters should be briefly explained in an itemised list with the name of the parameter set as an inline code block.
- An H2-size section with the title **Usage** which should contain at least one simple example of a valid configuration for the module.

For advances features in GLFM such as citations and formulae, see `doc/README.md` in the project repository [13].

`<ModuleName>Module.hpp` and `<ModuleName>Module.cpp`

All modules should consist of both a header file and a source file. In the header file, the module is defined together with all of its methods. Brief Doxygen documentation should be added to explain what each method does. The source file should provide the implementation of every method and also its more detailed Doxygen documentation. Methods should only be declared in the header and defined in the source file in order to keep the interface clean.

10.3.2 Module structure

All modules must inherit from the `Module` base class, which can be found in `src/core/module/Module.hpp`. The module base class provides two base constructors, a few convenient methods and several methods which the user is required to override. Each module should provide a constructor using the fixed set of arguments defined by the framework; this particular constructor is always called during by the module instantiation logic. These arguments for the constructor differ for unique and detector modules.

For unique modules, the constructor for a `TestModule` should be:

```
TestModule(Configuration& config, Messenger* messenger, GeometryManager*
 ↪  geo_manager)
    : Module(config) {}
```

For detector modules, the first two arguments are the same, but the last argument is a `std::shared_ptr` to the linked detector. It should always forward this detector to the base class together with the configuration object. Thus, the constructor of a detector module is:

```
TestModule(Configuration& config, Messenger* messenger,
 ↪  std::shared_ptr<Detector> detector)
    : Module(config, std::move(detector)) {}
```

The pointer to a Messenger can be used to bind variables to either receive or dispatch messages as explained in Section 4.6. The constructor should be used to bind required messages, set configuration defaults and to throw exceptions in case of failures. Unique modules can access the GeometryManager to fetch all detector descriptions, while detector modules directly receive a link to their respective detector.

In addition to the constructor, each module can override the following methods:

- `initialize()`: Called once per module from the main thread after loading and constructing all modules and before starting the event loop. This method can for example be used to initialize histograms.
- `initializeThread()`: Called after global initialization but before event processing and gives the possibility to initialize worker thread-specific members for modules. If multithreading is used, this method is called by each worker thread separately; if the simulation is run single-threaded, it is called once by the main thread.
- `run(Event* event)`: Called for every event in the simulation, with a pointer to the current event object as parameter. An exception should be thrown for serious errors, otherwise a warning should be logged.
- `finalizeThread()`: Called for each worker thread after processing all events in the run. If multithreading is used, this method is called by each worker thread separately; if the simulation is run single-threaded, it is called once by the main thread.
- `finalize()`: Called once per module from the main thread after processing all events in the run and before destructing the module. Typically used to save the output data (like histograms). Any exceptions should be thrown from here instead of the destructor.

If necessary, modules can also access the ConfigurationManager directly in order to obtain configuration information from other module instances or other modules in the framework using the `getConfigManager()` call. This allows to retrieve and e.g. store the configuration actually used for the simulation alongside the data.

If a module should be run using multithreading but requires to execute its run method in the order of event numbers, for example a module that writes to an output file, then the module can inherit from the `SequentialModule` class, without implementing additional functionality. This will ensure that the run method will receive events one-by-one and in the correct sequence.

10.4 Writing Thread-Safe Code

In Allpix Squared events are processed fully parallel on separate threads which requires some consideration when writing module code. This section briefly lists the most important aspects to take into account.

10.4.1 Member Variables

While the `initialize()` and `finalize()` of the module are guaranteed to be called sequentially, the `run()` method will be called simultaneously from different threads and for different events. Therefore, no module data members must be altered from within the `run()` function, otherwise these changes will affect other events being processed in parallel on other threads. Configuration parameters cached as member variables should therefore be set only in the `initialize()` function.

For initialization and finalization of thread-local data members, i.e. structures which have to be configured for each of the worker threads the module is executed on, the

`initializeThread()` and `finalizeThread()` methods are available. They are called once on each worker thread after the `initialize()` and before the `finalize()` methods, respectively.

10.4.2 Histograms

Allpix Squared uses ROOT histograms for collecting and storing statistics and other additional information about the simulation process. ROOT provides the template class `ROOT::TThreadedObject` which allows to use histograms in multithreaded environments but slightly alters the interface of the histogram objects. Furthermore, there have been significant changes to the class between minor release version of ROOT and it doesn't scale well with a large number of predefined threads. Therefore, Allpix Squared provides its own re-implementation of this class, `allpix::ThreadedHistogram` which also restores the original interface of the histogram classes, i.e. it is possible to instantiate, fill and store histograms the same way as in a single-threaded environment.

This class can be used as follows:

```
// Declaration of a new histogram of type "TH1D"
Histogram<TH1D> my_histogram;

// Creation of the histogram using the CreateHistogram helper method:
my_histogram = CreateHistogram<TH1D>("name", "title", 100, 0., 100.);

// Filling, setting bin contents and writing the histogram works as
 ↪  before:
my_histogram->Fill(12.);
my_histogram->SetBinContent(15, 23.);
my_histogram->Write();
```

10.4.3 Declaring a Module Thread-Safe

If a module is thread-safe, i.e. its `run()` function can be called from different threads in parallel without locking, it can be declared as thread-safe to the framework. In this case the ModuleManager will allow multithreading of calls to this module.

This declaration is done by placing the following call in the constructor of the module:

```
MyParallelModule::MyParallelModule(Configuration& config, Messenger*
 ↪  messenger, std::shared_ptr<Detector> detector)
    : Module(std::move(config), std::move(detector)) {
    // This module is thread-safe and can be called from different
 ↪  threads simultaneously:
    allow_multithreading();
}
```

By adding this statement, the module certifies to work correctly if its `run()` method is executed multiple times in parallel, for different events. This means in particular that the module will safely handle access to shared (for example static) variables as described in Section 10.4 and that it will properly assign and bind ROOT histograms

to their respective directories in the output ROOT file before the event processing starts and the `run()` method is called the first time. Access to constant operations in the `GeometryManager`, `Detector` and `DetectorModel` is always valid between various threads. In addition, sending and receiving messages is thread-safe.

Since multithreading might be disabled by other modules in the chain or by the user via the configuration file or command line, it might be required to check at runtime of the module if it is currently running in a multithreaded environment. This can be achieved with the following method:

```
MyParallelModule::run(Event* event) {
    if(multithreadingEnabled()) {
        // This module is currently running in a multithreaded
 ↪  environment
    } else {
        // This module is running in a fully sequential environment
    }
}
```

10.5 Adding a New Detector Model

Custom detector models based on the detector classes provided with Allpix Squared can easily be added to the framework. In particular Section 5.2 explains all parameters of the detector models currently available. The default models provided in the `models` directory of the repository can serve as examples. To create a new detector model, the following steps should be taken:

1. Create a new file with the name of the model followed by the `.conf` suffix (for example `your_model.conf`).
2. Add a configuration parameter `type` with the type of the model, at the moment either `monolithic` or `hybrid` for respectively monolithic sensors or hybrid models with bump bonds and a separate readout chip.
3. Add all required parameters and possibly optional parameters as explained in Section 5.2.
4. Include the detector model in the search path of the framework by adding the `model_paths` parameter to the general setting of the main configuration (see Section 3.4), pointing either directly to the detector model file or the directory containing it. It should be noted that files in this path will overwrite models with the same name in the default model folder.

Models should be contributed to the main repository to make them available to other users of the framework. To add the detector model to the framework the configuration file should be moved to the folder `models` of the repository. The file should then be added to the installation target in the `CMakeLists.txt` file of the `models` directory. Afterwards, a merge request can be created via the mechanism provided by the software repository [13].

10.6 How to contribute

Thanks for considering to contribute to Allpix Squared. Any type of merge request, ranging from small bugfixes, improvements to the documentation to entirely new functionality, is much appreciated. We, the maintainers, will try to our best to look carefully at every merge request.

If you only want to submit an issue, that is also welcome, please continue directly to the issue tracker [6] to open a ticket.

The following is a set of guidelines that will help both you as submitter as well as us maintainers to make it as easy as possible to contribute changes.

10.6.1 Core and modules

Allpix Squared is split up in a slim core, providing base functionality as configuration, detector geometry, management of modules, messaging as well as various utilities. The actual chain of simulation is developed in independent modules. Please try to separate any merge request for improvements or changes to the core from the implementation and updates of modules. Generally any kind of separable simulation functionality should be implemented in its own module and submitted as a individual merge request. Also try to submit individual merge request for independent changes to allow us to review them separately.

If you have any doubt about the best way to implement new functionality or how to split it up, please open an issue with the discussion tag on the issue tracker [6]. Also please do open an incomplete merge request as soon as possible with the "Draft" label to allow for early discussion.

10.6.2 Getting started

Please follow the next steps to setup your system for contributing. Note that these are slightly different from the normal installation instructions.

1. Fork the repository by clicking on "Fork" on the main repository [13].
2. Clone your local fork using `git clone https://gitlab.cern.ch/allpix-squared/allpix-squared.git` (when using HTTPS, this has to be changed accordingly for SSH or KRB5)
3. Install the latest version of the *clang* package with the *clang-format* and *clang-tidy* programs.
4. Follow the build instructions using CMake explained in the User's manual.

If you don't have an account for CERN's GitLab instance (restricted to CERN associates), you can fork the GitHub repository as well.

10.6.3 Making changes

Now you can start making changes and adding new functionality to the code.

1. Run `etc/git-hooks/install-hooks.sh` from the repository top folder to install the git-hook that automatically updates the format of the code to comply with the coding style.
2. Create a new branch from master with a description of the change using `git checkout -b my-new-branch-name`.
3. Read the relevant sections in the User's manual before starting to make changes.
4. Implement the new code and frequently commit using `git commit -m 'my commit message'`. Please use descriptive messages explaining what changed.
5. Push the code to your local mirror using `git push --set-upstream origin`.
6. Retrieve the latest changes to the upstream master every now and then. To do this add the upstream version to your remotes using `git remote add upstream https://gitlab.cern.ch/allpix-squared/allpix-squared.git` (or the SSH or KRB5 version if preferred). This only has to be done once, the first time after cloning the repository. Afterwards you fetch the changes using `git fetch upstream`. Then you can add the change preferably using rebase with `git rebase upstream master`. If that causes problems you can use merge with `git merge upstream master`.

10.6.4 Submitting a merge request

As soon as there exists something in your branch, a merge request can be opened on the main repository. Do not forget that it is not a problem to open a merge request for incomplete implementations.

1. Retrieve the latest changes from the upstream version as explained above.
2. Optionally format the code if you did not add the git-hook from the beginning, this can be done manually by running `make format` from the build directory.
3. Go to merge request and click on "New merge request".
4. Follow the instructions. Do not forget to use the 'Draft:' prefix if your code is only partially ready. Then submit the merge request.
5. Please wait for the maintainers to give you access to the continuous integration (CI) runners that will check your code if you do not already have it.
6. Add all the specific runners on your local repository at `https://gitlab.cern.ch/your-username/allpix-squared/settings/ci_cd`.
7. The pipeline can now be restarted and the CI will check your changes. If the CI fails and gives an error please refer to the log containing a description about what went wrong. It is likely that errors will appear because Allpix Squared enforces a strict policy of compiler errors and requires full compliance of the clang-tidy "linter" tool, which frequently complains about minor changes (it might help to search for `error:` to find the actual error(s) in the output). This clang-tidy tool can also be run locally on your pc by executing `make check-lint` from the build directory. Easy changes can be fixed automatically by executing `make lint`.
8. The maintainers will look at your proposed changes and likely provide some (constructive) feedback.
9. Please continue to update the code with the received comments until every reviewer and the continuous integration is happy :)

10. Your merge request can now be merged in. Congratulations and thank you so much, you have contributed something new to the repository.

11 Development Tools & CI

The following chapter will introduce a few tools included in the framework to ease development and help to maintain a high code quality. This comprises tools for the developer to be used while coding, as well as a continuous integration (CI) and automated test cases of various framework and module functionalities.

11.1 Additional Targets

A set of testing targets in addition to the standard compilation targets are automatically created by CMake to enable additional code quality checks and testing. Some of these targets are used by the project's CI, others are intended for manual checks. Currently, the following targets are provided:

- `make format`: Invokes the `clang-format` tool to apply the project's coding style convention to all files of the code base. The format is defined in the `.clang-format` file in the root directory of the repository and mostly follows the suggestions defined by the standard LLVM style with minor modifications. Most notably are the consistent usage of four whitespace characters as indentation and the column limit of 125 characters.

- `make check-format`: Also invokes the `clang-format` tool but does not apply the required changes to the code. Instead, it returns an exit code 0 (pass) if no changes are necessary and exit code 1 (fail) if changes are to be applied. This is used by the CI.

- `make lint`: Invokes the `clang-tidy` tool to provide additional linting of the source code. The tool tries to detect possible errors (and thus potential bugs), dangerous constructs (such as uninitialized variables) as well as stylistic errors. In addition, it ensures proper usage of modern C++ standards. The configuration used for the `clang-tidy` command can be found in the `.clang-tidy` file in the root directory of the repository.

- `make check-lint`: Also invokes the `clang-tidy` tool but does not report the issues found while parsing the code. Instead, it returns an exit code 0 (pass) if no errors have been produced and exit code 1 (fail) if issues are present. This is used by the CI.

- `make cppcheck`: Runs the `cppcheck` command for additional static code analysis. The output is stored in the file `cppcheck_results.xml` in XML 2.0 format. It should be noted that some of the issues reported by the tool are to be considered false positives.

- `make cppcheck-html`: Compiles a HTML report from the defects list gathered by `make cppcheck`. This target is only available if the `cppcheck-htmlreport` executable is found in the `PATH`.
- `make package`: Creates a binary release tarball as described in Section 10.2.

11.2 Packaging

Allpix Squared comes with a basic configuration to generate tarballs from the compiled binaries using the CPack command. In order to generate a working tarball from the current Allpix Squared build, the `RPATH` of the executable should not be set, otherwise the `allpix` binary will not be able to locate the dynamic libraries. If not set, the global `LD_LIBRARY_PATH` is used to search for the required libraries:

```
mkdir build
cd build
cmake -DCMAKE_SKIP_RPATH=ON ..
make package
```

Since the CMake installation path defaults to the project's source directory, certain files are excluded from the default installation target created by CMake. This includes the detector models in the `models/` directory as well as the additional tools provided in `tools/root_analysis_macros/` folder. In order to include them in a release tarball produced by CPack, the installation path should be set to a location different from the project source folder, for example:

```
cmake -DCMAKE_INSTALL_PREFIX=/tmp ..
```

The content of the produced tarball can be extracted to any location of the file system, but requires the ROOT6 and Geant4 libraries as well as possibly additional libraries linked by individual at runtime.

For this purpose, a `setup.sh` shell script is automatically generated and added to the tarball. By default, it contains the ROOT6 path used for the compilation of the binaries. Additional dependencies, either library paths or shell scripts to be sourced, can be added via CMake for individual modules using the CMake functions described below. The paths stored correspond to the dependencies used at compile time, it might be necessary to change them manually when deploying on a different computer.

11.2.1 `ADD_RUNTIME_DEP(name)`

This CMake command can be used to add a shell script to be sourced to the setup file. The mandatory argument `name` can either be an absolute path to the corresponding file, or only the file name when located in a search path known to CMake, for example:

```
# Add "geant4.sh" as runtime dependency for setup.sh file:
ADD_RUNTIME_DEP(geant4.sh)
```

The command uses the `GET_FILENAME_COMPONENT` command of CMake with the `PROGRAM` option. Duplicates are removed from the list automatically. Each file found will be written to the setup file as

```
source <absolute path to the file>
```

11.2.2 `ADD_RUNTIME_LIB(names)`

This CMake command can be used to add additional libraries to the global search path. The mandatory argument `names` should be the absolute path of a library or a list of paths, such as:

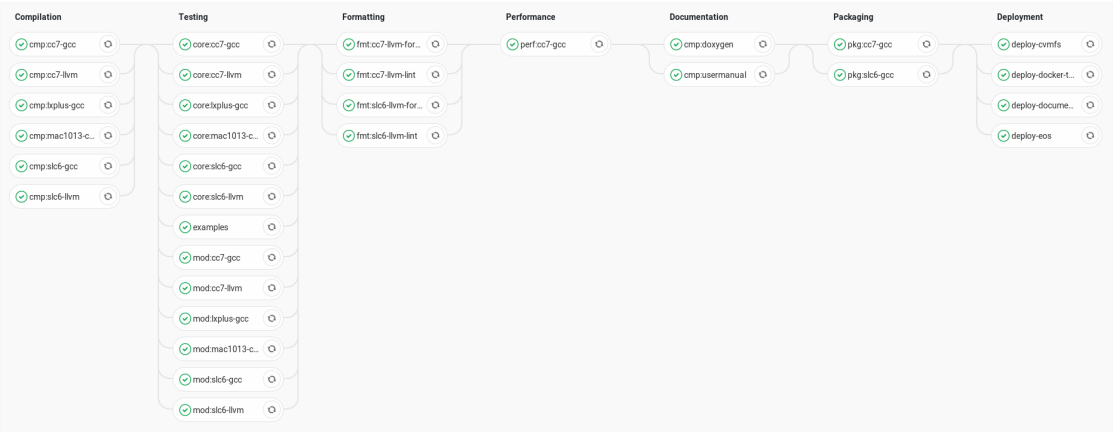
```
# This module requires the LCIO library:
FIND_PACKAGE(LCIO REQUIRED)
# The FIND routine provides all libraries in the LCIO_LIBRARIES
 ↪  variable:
ADD_RUNTIME_LIB(${LCIO_LIBRARIES})
```

The command uses the `GET_FILENAME_COMPONENT` command of CMake with the `DIRECTORY` option to determine the directory of the corresponding shared library. Duplicates are removed from the list automatically. Each directory found will be added to the global library search path by adding the following line to the setup file:

```
export LD_LIBRARY_PATH="<library directory>:$LD_LIBRARY_PATH"
```

11.3 Continuous Integration

Quality and compatibility of the Allpix Squared framework is ensured by an elaborate continuous integration (CI) which builds and tests the software on all supported platforms. The Allpix Squared CI uses the GitLab Continuous Integration features and consists of seven distinct stages as depicted in the figure below. It is configured via the `.gitlab-ci.yml` file in the repository's root directory, while additional setup scripts for the GitLab Ci Runner machines and the Docker instances can be found in the `.gitlab/ci` directory.



Typical Allpix Squared continuous integration pipeline with 34 jobs distributed over seven distinct stages. In this example, all jobs passed.

The **compilation** stage builds the framework from the source on different platforms. Currently, builds are performed on CentOS 7, CentOS 8, and macOS. On Linux type platforms, the framework is compiled with recent versions of GCC and Clang, while the latest AppleClang is used on macOS. The build is always performed with the default compiler flags enabled for the project:

```
-pedantic -Wall -Wextra -Wcast-align -Wcast-qual -Wconversion
-Wuseless-cast -Wctor-dtor-privacy -Wzero-as-null-pointer-constant
-Wdisabled-optimization -Wformat=2 -Winit-self -Wlogical-op
-Wmissing-declarations -Wmissing-include-dirs -Wnoexcept
-Wold-style-cast -Woverloaded-virtual -Wredundant-decls
-Wsign-conversion -Wsign-promo -Wstrict-null-sentinel
-Wstrict-overflow=5 -Wswitch-default -Wundef -Werror -Wshadow
-Wformat-security -Wdeprecated -fdiagnostics-color=auto
-Wheader-hygiene
```

The **testing** stage executes the framework system and unit tests described in the next chapter. Different jobs are used to run different test types. This allows to optimize the CI setup depending on the demands of the test to be executed. All tests are expected to pass, and no code that fails to satisfy all tests will be merged into the repository.

The **formatting** stage ensures proper formatting of the source code using the `clang-format` and following the coding conventions defined in the `.clang-format` file in the repository. In addition, the `clang-tidy` tool is used for "linting" of the source code. This means, the source code undergoes a static code analysis in order to identify possible sources of bugs by flagging suspicious and non-portable constructs used. Tests are marked as failed if either of the CMake targets `make check-format` or `make check-lint` fail. No code that fails to satisfy the coding conventions and formatting tests will be merged into the repository. Furthermore, also basic sanity checks are carried out on the CMake build framework code using `cmake-lint`.

The **performance** stage runs a longer simulation with several thousand events and measures the execution time. This facilitates monitoring of the simulation performance, a failing job would indicate a degradation in speed. These CI jobs run on dedicated machines with only one concurrent job. Performance tests are separated into their own CI stage because their execution is time consuming and they should only be started once proper formatting of the new code is established.

The **documentation** stage prepares this user manual as well as the Doxygen source code documentation for publication. This also allows to identify e.g. failing compilation of the LaTeX document.

The **packaging** stage wraps the compiled binaries up into distributable tarballs for several platforms. This includes adding all libraries and executables to the tarball as well as preparing the `setup.sh` script to prepare run-time dependencies using the information provided to the build system. This procedure is described in more detail in Section 10.2.

Finally, the **deployment** stage is only executed for new tags in the repository. Whenever a tag is pushed, this stages receives the build artifacts of previous stages and publishes them to the Allpix Squared project website through the EOS file [96]. More detailed information on deployments is provided in Section 10.4.

11.4 Automatic Deployment

The CI is configured to automatically deploy new versions of Allpix Squared and its user manual and code reference to different places to make them available to users. This section briefly describes the different deployment end-points currently configured and in use. The individual targets are triggered either by automatic nightly builds or by publishing new tags. In order to prevent accidental publications, the creation of tags is protected. Only users with Maintainer privileges can push new tags to the repository. For new tagged versions, all deployment targets are executed.

Software deployment to CVMFS

The software is automatically deployed to CERN's VM file system (CVMFS) [16] for every new tag. In addition, the `master` branch is built and deployed every night. New versions are published to the folder `/cvmfs/clicdp.cern.ch/software/allpix-squared/` where a new folder is created for every new tag, while updates via the `master` branch are always stored in the `latest` folder.

The deployed version currently comprises all modules as well as the detector models shipped with the framework. An additional `setup.sh` is placed in the root folder of the respective release, which allows to set up all runtime dependencies necessary for executing this version. Versions both for CentOS 7 and CentOS 8 are provided.

The deployment CI job runs on a dedicated computer with a GitLab SSH runner. Job artifacts from the packaging stage of the CI are downloaded via their ID using the script found in `.gitlab/ci/download_artifacts.py`, and are made available to the `cvclicdp` user which has access to the CVMFS interface. The job checks for concurrent deployments to CVMFS and then unpacks the tarball releases and publishes them to the CLICdp experiment CVMFS space, the corresponding script for the deployment can be found in `.gitlab/ci/gitlab_deployment.sh`. This job requires a private API token to be set as secret project variable through the GitLab interface, currently this token belongs to the service account user `ap2`.

Documentation deployment

The documentation is provided as an online version and a PDF. Both get generated from the Markdown [95] documentation found in the project repository [13]. For the PDF the plain Markdown documentation is converted via pandoc [97] and a Python script adjusting to the LaTeX for the PDF.

The CI deploys the PDF on CERN's EOS at `/eos/project/a/allpix-squared/www/usermanual/`. The PDF documentation is published for new tagged versions of the framework and for nightlies (version `latest`). The version number is attached to the file name so that the website contains the usermanual for all versions.

The CI jobs uses the `ci-web-deployer` Docker image from the CERN GitLab CI tools to access EOS, which requires a specific file structure of the artifact. All files in the artifact's `public/` folder will be published to the `www/` folder of the given project. This job requires the secret project variables `EOS_ACCOUNT_USERNAME` and `EOS_ACCOUNT_PASSWORD` to be

set via the GitLab web interface. Currently, this uses the credentials of the service account user ap2.

The online version of the documentation is included in the project's website, which uses hugo [98] to generate HTML from Markdown. The project website is hosted in its own repositroy [99] and deployed directly from the CI via GitLab Pages (see also how-to.docs.cern.ch).

If a new tag is pushed to the project repository, the CI in the project repository triggers a CI pipeline in the website repository. This pipeline clones the project repository to get the Markdown documentation, generates the HTML with hugo and deploys it. This setup allows to update information on the website without pushing a new tag to the project repository.

For more information on the documentation, see `doc/README.md` in the project repository [13].

11.5 Building Docker Images

New Allpix Squared Docker images are automatically created and deployed by the CI for every new tag and as a nightly build from the `master` branch. New versions are published to project Docker container registry [15]. Tagged versions can be found via their respective tag name, while updates via the nightly build are always stored with the `latest` tag attached.

The final Docker image is formed from two consecutive images with different layers of software added. The `deps` image contains all build dependencies such as compilers, CMake, and git as well as the two main dependencies of the framework are ROOT6 and Geant4. It derives from the latest Ubuntu LTS Docker image and can be build using the `etc/docker/Dockerfile.deps` file via the following commands:

```
docker build --file etc/docker/Dockerfile.deps           \
             --tag gitlab-registry.cern.ch/allpix-squared/\
             allpix-squared/allpix-squared-deps:vX       \
             .
docker push gitlab-registry.cern.ch/allpix-squared/\
            allpix-squared/allpix-squared-deps
```

This image is created manually and only updated when necessary, i.e. if major new version of the underlying dependencies are available. The placeholder `vX` is a version number which should be incremented when applying changes or updating software versions in the `deps` Docker image. This version number should subsequently also be adjusted in the CI pipeline (`.gitlab-ci.yml`) and the Allpix Squared Docker file (`/etc/docker/Dockerfile`) so that the correct dependencies image is picked up.

Important: The Docker image containing the dependencies should not be flattened with commands like

```
docker export <container id> | docker import - <tag name>
```

because it strips any `ENV` variables set or used during the build process. They are used to set up the ROOT6 and Geant4 environments. When flattening, their executables and data paths cannot be found in the final Allpix Squared image.

Finally, the latest revision of Allpix Squared is built using the file `etc/docker/Dockerfile`. This job is performed automatically by the continuous integration and the created containers are directly uploaded to the project's Docker registry:

```
docker build --file etc/docker/Dockerfile
 ↪  \
             --tag gitlab-registry.cern.ch/allpix-squared/allpix-squared
              ↪  \
             .
```

A short summary of potential use cases for Docker images is provided in Section 2.7.

12 Automated Testing

The build system of the framework provides a set of automated tests which are executed by the CI to ensure proper functioning of the framework and its modules.

> **Note**: Since different Geant4 versions may result in different events, a specific Geant4 version needs to be chosen for the test suite. All tests in the repository are written for Geant4 11.1 / LCG103.

The tests can also be manually invoked from the build directory of Allpix Squared with:

```
ctest
```

When executed by the CI, the results on passed and failed tests are automatically gathered and prominently displayed in merge requests along with the overall CI pipeline status. This allows a quick identification of issues without having to manually search through the log of several CI jobs.

The different subcategories of tests described below can be executed or ignored using the `-E` (exclude) and `-R` (run) switches of the `ctest` program:

```
ctest -R test_performance
```

12.1 Test Configurations

Test configuration files consist of regular Allpix Squared configuration files for a simulation, invoking the desired behavior to be tested. In addition, test files can contain tags and pass conditions as described in the subsequent section.

CMake automatically searches for Allpix Squared configuration files in the different directories and passes them to the Allpix Squared executable (cf. Section 3.5). Adding a new test is as simple as adding a new configuration file to the respective directories and specifying the pass or fail conditions based on the tags described in the following paragraphs.

Three different types of tests are distinguished:

12.1.1 Framework Functionality Tests

The framework functionality tests validate the core framework components such as seed distribution, multithreading capabilities, configuration parsing and coordinate transformations. The configuration of the tests can be found in the `etc/unittests/test_core` directory of the repository and are automatically discovered by CMake.

12.1.2 Module Functionality Tests

These tests are meant to ensure proper functioning of an individual module given a defined input and thus protect the framework against accidental changes affecting the physics simulation. Using a fixed seed (using the `random_seed` configuration keyword) together with a specific version of Geant4 [1], if necessary, allows to reproduce the same simulation event.

Typically, one single event is produced per test and the `DEBUG`-level logging output of the respective module is checked against pre-defined expectation output using regular expressions. Once modules are altered, their respective expectation output has to be adapted after careful verification of the simulation result.

Module tests are located within the individual module source folders and are only enabled if the respective module will be built. For new modules, the directory in which the test files are located needs to be registered in the main CMake file of the module as described in Section 10.3. Module test files have to start with a two-digit number and end with the file extension `.conf`, e.g. `01-mytest.conf`, to be detected.

12.1.3 Performance Tests

These tests run a set of simulations on a dedicated machine to catch any unexpected prolongation of the simulation time, e.g. by an accidentally introduced heavy operation in a hot loop. Performance tests use configurations prepared such, that one particular module takes most of the load (dubbed the *slowest instantiation* by Allpix Squared), and a few of thousand events are simulated starting from a fixed seed for the pseudo-random number generator. The `#TIMEOUT` keyword in the configuration file will ask CTest to abort the test after the given running time.

In the project CI, performance tests are limited to native runners, i.e. they are not executed on docker hosts where the hypervisor decides on the number of parallel jobs. Only one test is performed at a time.

Despite these countermeasures, fluctuations on the CI runners occur, arising from different loads of the executing machines. Thus, all performance CI jobs are marked with the `allow_failure` keyword which allows GitLab to continue processing the pipeline but will mark the final pipeline result as *passed with warnings* indicating an issue in the pipeline. These tests should be checked manually before merging the code under review.

The configuration of the tests can be found in the `etc/unittests/test_performance` directory of the repository and are automatically discovered by CMake.

12.2 Test Tags, Pass and Fail Conditions

Test tags allow to influence the execution condition of the given test configuration, or to define a required condition for passing or failing the test. These expressions are simply placed in the configuration file of the corresponding tests, a tag at the beginning of the line indicates which test tag the line corresponds to. The following tags are available:

- **Passing a test**: The expression marked with the tag `#PASS` has to be found in the output in order for the test to pass. If the expression is not found, the test fails.
- **Failing a test**: If the expression tagged with `#FAIL` is found in the output, the test fails. If the expression is not found, the test passes.
- **Depending on another test**: The tag `#DEPENDS` can be used to indicate dependencies between tests. For example, module test `ROOTObjectReader/01-reading` implements such a dependency as it uses the output of module test `ROOTObjectWriter/01-write` to read data from a previously produced Allpix Squared data file.
- **Defining a timeout**: For performance tests the runtime of the application is monitored, and the test fails if it exceeds the number of seconds defined using the `#TIMEOUT` tag.
- **Adding additional CLI options**: Additional module command line options can be specified for the `allpix` executable using the `#OPTION` tag, following the format found in Section 3.5. The `-o` flag will be added automatically. Multiple options can be supplied by repeating the `#OPTION` tag in the configuration file, only one option per tag is allowed. In exactly the same way options for the detectors can be set as well using the `#DETOPION` tag, where `-g` will be added automatically. For all other command line options to be passed to the executable, the `#CLIOPTION` can be used. Here, the complete flag and possible value needs to be passed, e.g. `-j9`.
- **Defining a test case label**: Tests can be grouped and executed based on labels, e.g. for code coverage reports. Labels can be assigned to individual tests using the `#LABEL` tag.
- **Describing the test**: Every test should bear a short description of its goal. This descriptive text can be provided via the `#DESC` tag and is required for every test. CMake will print warnings for every test missing this tag.
- **Running scripts**: Some tests require additional input which needs to be generated by a script. For this propose, commands to be executed *before* the tests starts can be provided via the `#BEFORE_SCRIPT` tag, e.g.

  ```
  #BEFORE_SCRIPT python
   ↪  @PROJECT_SOURCE_DIR@/etc/scripts/create_deposition_file.py
   ↪  --type a --detector mydetector --events 2 --steps 1 --seed 0
  ```

  to run a Python script that generates an input file read by the test.
- **Requiring external data**: Some tests require external data which needs to be downloaded before executing the test. For this purpose, the `#DATA` tag is available, which can contain file paths which will be set as required files for the test. If registered with CMake's ExternalData module, they will be downloaded automatically.

12.3 Interpretation of Pass and Fail Conditions

Multiple pass or fail conditions can be separated by a semicolon or by adding multiple `#PASS` or `#FAIL` expressions. It should however be noted that test passes or fails *if any of these conditions is met*, i.e. the conditions are combined with a logical `OR`. At least one pass or one fail conditions must be present in every test.

Pass and fail condition are not interpreted as regular expressions but relevant characters are automatically escaped. This allows to directly copy corresponding lines form the log into the respective condition without manually creating a matching regular expression. A noteworthy exception to this are line breaks. To ease matching of multi-line expressions, the newline escape sequence `\n` of any test expression is automatically expanded to `[\r\n\t ]*` to match any new line, carriage return, tab and whitespace characters following the line break.

12.4 Warning and Error Messages During Testing

If no explicit fail conditions are specified, the test will fail if any `WARNING`, `ERROR` or `FATAL` appears in the output log unless it is already part of the pass condition. For example, if a test is supposed to pass in case of an error provoked

```
(FATAL) [I:GeometryBuilderGeant4] Error during execution of run:
                                  Could not find a detector model of
                                   ↪ type 'missing_model'
                                  Please check your configuration and
                                   ↪ modules. Cannot continue.
```

The full error message including the `FATAL` has to be provided as pass condition:

```
#PASS (FATAL) [I:GeometryBuilderGeant4] Error during execution of
 ↪ run:\nCould not find a detector model of type 'missing_model'
```

If a test is expected to create multiple error or warning messages which cannot be matched with a single pass condition, the `#FAIL` parameter should be set explicitly to avoid matching the respective flags:

```
# This test created multiple WARNING messages, we exclude WARNING from
 ↪ the
# fail expression by explicitly defining it as FATAL only:
#PASS (ERROR) Multithreading disabled since the current module
 ↪ configuration does not support it
#FAIL FATAL
```

12.5 Directory Variables in Tests

Sometimes it is necessary to pass directories or file names as test input. To facilitate this, the test files can contain variables which are replaced with the respective paths before being executed. All variable names have to be enclosed in `@` symbols to be detected and

parsed correctly. Variables can be used both in test files and the auxiliary configuration files such as detector geometry definitions.

The following variables are available:

- `@TEST_DIR@`: Directory in which the current test is executed, i.e. where all output files will be placed.
- `@TEST_BASE_DIR@`: Base directory under which all tests are being executed. This can be used to reference the output files from another test. It should be noted that the respective test has to be referenced using the `#DEPENDS` keyword to ensure that it successfully ran before.
- `@PROJECT_SOURCE_DIR@`: The root directory of the project. This can for example be used to call a script provided in the `etc/scripts` directory of the repository [13].

The following example demonstrates the use of these variables. A script is called before executing the test and an input file is expected:

```
[Allpix]
detectors_file = "detector.conf"

[DepositionReader]
file_name = "@TEST_DIRECTORY@/deposition.root"

#BEFORE_SCRIPT python
 ↪  @PROJECT_SOURCE_DIR@/etc/scripts/create_deposition_file.py --type a
 ↪  --detector mydetector
```

13 FAQ

This chapter provides answers to some of the most frequently asked questions concerning usage, configuration and extension of the Allpix Squared framework.

13.1 Installation & Usage

What is the easiest way to use Allpix Squared on CERN's LXPLUS? Central installations of Allpix Squared on LXPLUS are provided via CVMFS for both supported LXPLUS operating systems, CERN CentOS 7 and CentOS 8. Please refer to Section 2.8 for the details of how to access these installations.

What is the quickest way to get a local installation of Allpix Squared? The project provides ready-to-use Docker containers which contain all dependencies such as Geant4 and ROOT. Please refer to Section 2.7 for more information on how to start and use these containers.

13.2 Configuration

How do I run a module only for one detector? This is only possible for detector modules (which are constructed to work on individual detectors). To run it on a single detector, one should add a parameter `name` specifying the name of the detector (as defined in the detector configuration file):

```
[ElectricFieldReader]
name = "dut"
model = "mesh"
file_name = "../example_electric_field.init"
```

How do I run a module only for a specific detector type? This is only possible for detector modules (which are constructed to work on individual detectors). To run it for a specific type of detector, one should add a parameter `type` with the type of the detector model (as set in the detector configuration file by the `model` parameter):

```
[ElectricFieldReader]
type = "timepix"
model = "linear"
bias_voltage = -50V
depletion_voltage = -30V
```

Please refer to Section 4.3 for more information.

How can I run the same type of module with different settings? This is possible by using the `input` and `output` parameters of a module that specify the messages of the module:

```
[DefaultDigitizer]
name = "dut0"
adc_resolution = 4
output = "low_adc_resolution"

[DefaultDigitizer]
name = "dut0"
adc_resolution = 12
output = "high_adc_resolution"
```

By default, both the input and the output of module are messages with an empty name. In order to further process the data, subsequent modules require the `input` parameter to not receive multiple messages:

```
[DetectorHistogrammer]
input = "low_adc_resolution"
name = "dut0"

[DetectorHistogrammer]
input = "high_adc_resolution"
name = "dut0"
```

Please refer to Section 4.6 for more information.

How can I temporarily ignore a module during development? The section header of a particular module in the configuration file can be replaced by the string `Ignore`. The section and all of its key/value pairs are then ignored. Modules can also be excluded from the compilation process as explained in Section 2.5.

Can I get a high verbosity level only for a specific module? Yes, it is possible to specify verbosity levels and log formats per module. This can be done by adding the `log_level` and/or `log_format` key to a specific module to replace the parameter in the global configuration sections.

Can I import an electric field from TCAD and use it for simulating propagation? Yes, the framework includes a tool to convert DF-ISE files from TCAD to an internal format which Allpix Squared can parse. More information about this tool can be found in Section 14.2, instructions to import the generated field are provided in Section 3.7.

What parameters should I consider when writig a simulation for a non-silicon sensor? While Allpix Squared implements several material-dependent default parameters, other parameters and models default to values suitable for silicon sensors. It is in any case advisable to check the following configuration parameters to ensure consistent results.

- Sensor material: The parameter `sensor_material`, to be adjusted in the corresponding detector model file, is crucial for the particle interaction simulated via Geant4 and defines further default parameters.
- Charge creation energy: The parameter `charge_creation_energy` is available in several modules for energy deposition and provides a material dependent default. For default values refer to Section 6.1.
- Fano factor: The parameter `fano_factor` is available in several modules for energy deposition and provides a material dependent default. For default values refer to Section 6.1.
- Mobility Model: The parameter `mobility_model` needs to be adapted to the sensor material by the user. Section 6.2 lists the available models.
- Recombination Model: The parameter `recombination_model` can be adapted by the user. Section 6.3 lists the available models.

13.3 Detector Models

I want to use a detector model with one or several small changes, do I have to create a whole new model for this? No, models can be specialized in the detector configuration file. To specialize a detector model, the key that should be changed in the standard detector model, e.g. like `sensor_thickness`, should be added as key to the section of the detector configuration (which already contains the position, orientation and the base model of the detector). Only parameters in the header of detector models can be changed. If support layers should be changed, or new support layers are needed, a new model should be created instead. Please refer to Section 5.2 for more information.

13.4 Data Analysis

How do I access the history of a particular object? Many objects can include an internal link to related other objects (for example `getPropagatedCharges` in the `PixelCharge` object), containing the history of the object (thus the objects that were used to construct the current object). These referenced objects are stored as special ROOT pointers inside the object, which can only be accessed if the referenced object is available in memory. In Allpix Squared this requirement can be automatically fulfilled by also binding the history object of interest in a module. During analysis, the tree holding the referenced object should be loaded and pointing to the same event entry as the object that requests the reference. If the referenced object can not be loaded, an exception is thrown by the retrieving method. Please refer to Section 7.2 for more information.

How do I access the Monte Carlo truth of a specific PixelHit? The Monte Carlo truth is part of the history of a PixelHit. This means that the Monte Carlo truth can be retrieved as described in the question above. Because accessing the Monte Carlo truth of a PixelHit is quite a common task, these references are stored directly for every new object created. This allows to retain the information without the necessity to keep the

full object history including all intermediate steps in memory. Please refer to Section 7.2 for more information.

How do I find out, which Monte Carlo particles are primary particles and which have been generated in the sensor? The Monte Carlo truth information is stored per-sensor as MCParticle objects. Each MCParticle stores, among other information, a reference to its parent. Particles which have entered the sensor from the outside world do not have parent MCParticles in the respective sensor and are thus primaries.

Using this approach it is possible, to e.g. treat a secondary particle produced in one detector as primary in a following detector.

Below is some pseudo-code to filter a list of MCParticle objects for primaries based on their parent relationship:

```
// Collect all primary particles of the event:
std::vector<const MCParticle*> primaries;

// Loop over all MCParticles available
for(auto& mc_particle : my_mc_particles) {
    // Check for possible parents:
    if(mc_particle.getParent() != nullptr) {
        // Has a parent, thus was created inside this sensor.
        continue;
    }

    // Has no parent particles in this sensor, add to primary list.
    primaries.push_back(&mc_particle);
}
```

A similar function is used e.g. in the DetectorHistogrammer module to filter primary particles and create position-resolved graphs. Furthermore, the PixelHit and PixelCharge objects provide two member functions to access Monte Carlo particles, one which returns all known particles, `getMCParticles()`, and a second function called `getPrimaryMCParticles()` which already performs the above filtering and only returns primary particle references.

How do I access data stored in a file produced with the ROOTObjectWriter from an analysis script? Allpix Squared uses ROOT trees to directly store the relevant C++ objects as binary data in the file. This retains all information present during the simulation run, including relations between different objects such as assignment of Monte Carlo particles. In order to read such a data file in an analysis script, the relevant library as well as its header have to be loaded.

In ROOT this can be done interactively by loading a data file, the necessary shared library objects and a macro for the analysis:

```
root -l data_file.root
root [1] .L ~/path/to/your/allpix-squared/lib/libAllpixObjects.so
root [2] .L analysisMacro.C+
root [3] readTree(_file0, "detector1")
```

A simple macro for reading DepositedCharges from a file and displaying their position is presented below:

```
#include <TFile.h>
#include <TTree.h>

// FIXME: adapt path to the include file of APSQ installation
#include "/path/to/your/allpix-squared/DepositedCharge.hpp"

// Read data from tree
void readTree(TFile* file, std::string detector) {

    // Read tree of deposited charges:
    TTree* dc_tree = static_cast<TTree*>(file->Get("DepositedCharge"));
    if(!dc_tree) {
        throw std::runtime_error("Could not read tree");
    }

    // Find branch for the detector requested:
    TBranch* dc_branch = dc_tree->FindBranch(detector.c_str());
    if(!dc_branch) {
        throw std::runtime_error("Could not find detector branch");
    }

    // Allocate object vector and link to ROOT branch:
    std::vector<allpix::DepositedCharge*> deposited_charges;
    dc_branch->SetObject(&deposited_charges);

    // Go through the tree event-by-event:
    for(int i = 0; i < dc_tree->GetEntries(); ++i) {
        dc_tree->GetEntry(i);
        // Loop over all deposited charge objects
        for(auto& charge : deposited_charges) {
            std::cout << "Event " << i << ": "
                      << "charge = " << charge->getCharge() << ", "
                      << "position = " << charge->getGlobalPosition()
                      << std::endl;
        }
    }
}
```

A more elaborate example for a data analysis script can be found in the `tools` directory of the repository [13] and in Section 14.3. Scripts written in both C++ and in Python are provided.

How can I convert data from the ROOTObject format to other formats? Since the ROOTObject format is the native format of Allpix Squared, the stored data can be read into the framework again. To convert it to another format, a simple pseudo-simulation setup can be used, which reads in data with one module and stores it with another.

In order to convert for example from ROOTObjects to the data format used by the Corryvreckan reconstruction framework, the following configuration could be used:

```
[Allpix]
number_of_events = 999999999
detectors_file = "telescope.conf"
random_seed_core = 0

[ROOTObjectReader]
file_name = "input_data_rootobjects.root"

[CorryvreckanWriter]
file_name = "output_data_corryvreckan.root"
reference = "mydetector0"
```

13.5 Development

How do I write my own output module? An essential requirement of any output module is its ability to receive any message of the framework. This can be implemented by defining a private `filter` function for the module as described in Section 4.6. This function will be called for every new message dispatched within the framework, and should contain code to decide whether to discard or cache a message for processing. Heavy-duty tasks such as handling data should not be performed in the `filter` routine, but deferred to the `run` function of the respective output module. The `filter` function should only decide whether to keep a message for processing or to discard it before the `run` function.

How do I process data from multiple detectors? When developing a new Allpix Squared module which processes data from multiple detectors, e.g. as the simulation of a track trigger module, this module has to be of type *unique* as described in Section 4.4. As a *detector* module, it would always only have access to the information linked to the specific detector is has been instantiated for. The module should then request all messages of the desired type using the messenger call `bindMulti` as described in Section 4.6. For PixelHit messages, an example code would be:

```
TrackTriggerModule(Configuration&, Messenger* messenger,
 ↪  GeometryManager* geo_manager) {
    messenger->bindMulti<MCTrackMessage>(this, MsgFlags::NONE);
}
std::vector<std::shared_ptr<PixelHitMessage>> messages;
```

The correct detectors have then to be selected in the `run` function of the module implementation.

How do I calculate an efficiency in a module? Calculating efficiencies always requires a reference. For hit detection efficiencies in Allpix Squared, this could be the Monte Carlo truth information available via the MCParticle objects. Since the framework only runs modules, if all input message requirements are satisfied, the message flags described in Section 4.6 have to be set up accordingly. For the hit efficiency example, two different message types are required, and the Monte Carlo truth should always be required (using `MsgFlags::REQUIRED`) while the PixelHit message should be optional:

```
MyModule::MyModule(Configuration& config, Messenger* messenger,
 ↪  std::shared_ptr<Detector> detector)
    : Module(config, detector), detector_(std::move(detector)) {

    // Bind messages
    messenger->bindSingle<PixelHitMessage>(this);
    messenger->bindSingle<MCParticleMessage>(this, MsgFlags::REQUIRED);
}
```

How do I add a new sensor material? When adding a new sensor material, additions at several positions in the code are necessary:

- Add material to list of available sensor materials in `src/core/geometry/DetectorModel.hpp`.
- If not available yet, add material to the Geant4 material manager (`src/modules/GeometryBuilderGeant4/MaterialManager.cpp`). See examples of either using a material known to Geant4 or defining compositions in the code. It should be noted that the key of the `materials_` map needs to match the name of the sensor material defined in the previous step, transformed to lower case letters.
- Define default values for the material properties listed in `src/physics/MaterialProperties.hpp`.
- Add the list of material properties to the corresponding section of the user manual (`doc/usermanual/chapters/06_models/01_material_properties.md`).

Any contribution to the framework in terms of new sensor material definitions is welcome and can be added via a dedicated merge request in the repository [13].

13.6 Debugging

What should I include in a bug report? In all bug reports, the output of `allpix --version` should always be provided as it provides vital information about the build and the system it is running on.

Ideally, you provide a minimum working example (MWE) of a config that produces the bug. To create an MWE, try to remove as much possible from your configuration files that does not change the appearance of the bug. This helps developers to understand the bug more quickly. Please provide all files to reproduce the simulation (main configuration, geometry configuration and if applicable fields and detector model configuration).

If the bug occurs only in a specific event, use the `skip_events` parameter to fast-forward to this event, fix the random seed using the `random_seed` parameter and set `number_of_events = 1`. The parameters are explained in Section 3.4.

If the bug is a crash or an unexpected error, please also provide a backtrace (see "How do I debug Allpix Squared?").

How do I debug Allpix Squared? A good first step to debugging is to increase the logging level in Allpix Squared. Start by setting the logging level to `DEBUG` in the module where you expect the bug to happen. For maximum information, you can set the logging level to `TRACE`. See Section 3.8 for details on logging.

If you encounter a bug with Geant4, see "How can I see the output of Geant4?" and "How can I enable tracking verbosity for Geant4?".

To inspect a crash or an unexpected error in detail, a debugger like gdb is a useful tool to find out where exactly the program crashed or why the error was thrown.

Assuming `config.conf` crashes Allpix Squared, a full backtrace is created like this:

```
gdb --args allpix -c config.conf
run
thread apply all backtrace full
```

If you want to create a backtrace when Allpix Squared explicitly throws an error, you can use:

```
gdb --args allpix -c config.conf
catch throw
run
backtrace
```

How can I see the output of Geant4? Geant4's output stream is configured in the GeometryBuilderGeant4 module. The output stream for Geant4's error stream is logged with logging level `WARNING`, the standard stream with logging level `TRACE`. These values can be adjusted via `log_level_g4cerr` and `log_level_g4cout` (see module documentation).

How can I enable tracking verbosity for Geant4? By setting `geant4_tracking_verbosity = 1` in the DepositionGeant4 module. For details check the parameter entry in the module documentation. Note that you also need the appropriate logging level to get the Geant4 output (see "How can I see the output of Geant4?").

13.7 Miscellaneous

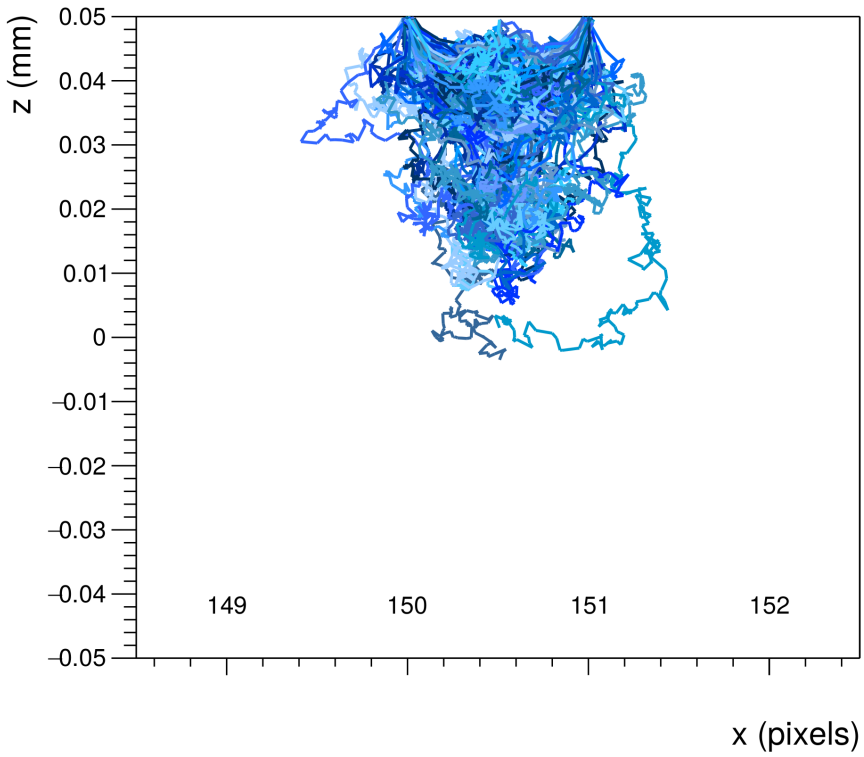
How can I produce nicely looking drift-diffusion line graphs? The GenericPropagation module offers the possibility to produce line graphs depicting the path each of the charge carrier groups have taken during the simulation. This is a useful way to visualize the drift and diffusion along field lines.

An optional parameter allows to reduce the lines drawn to those charge carrier groups which have reached the sensor surface to provide some insight into where from the collected charge carriers originate and how they reached the implants. One graph is written per event simulated, usually this option should thus only be used when simulating one or a few events but not during a production run.

In order to produce a precise enough line graph, the integration time steps have to be chosen carefully - usually finer than necessary for the actual simulation. Below is a set of settings used to simulate the drift and diffusion in a high resistivity CMOS silicon sensor. Settings of the module irrelevant for the line graph production have been omitted.

```
[GenericPropagation]
charge_per_step = 5
timestep_min = 1ps
timestep_max = 5ps
timestep_start = 1ps
spatial_precision = 0.1nm

output_linegraphs = true
output_plots_step = 100ps
output_plots_align_pixels = true
output_plots_use_pixel_units = true

# Optional to only draw charge carrier groups which reached the implant
 ↪  side:
# output_plots_lines_at_implants = true
```

With these settings, a graph of similar precision to the one presented in the figure below can be produced. The required time stepping size and number of output plot steps varies greatly with the sensor and its applied electric field. The number of charge carriers per group can be used to vary the density of lines drawn. Larger groups result in fewer lines.



Drift and diffusion visualization of charge carrier groups being transported through a high-resistivity CMOS silicon sensor. The plot shows the situation after an integration time of 20 nanoseconds, only charge carrier groups which reached the implant side of the sensor are drawn.

Why does GeometryBuilderGeant4 warn me about reduced performance with disabled multithreading? You might have see this log message:

```
Using Geant4 modules without multithreading might reduce performance
↪ when using complex geometries, please check the documentation for
↪ details
```

You might want to set `multithreading=true` and `workers=1` instead of instead of `multithreading=false` if this is allowed by the module configuration.

The reason behind message this is explained more detailed in Section 14.1.

14 Additional Tools & Resources

This chapter briefly describes tools provided with the Allpix Squared framework, which might be re-used in new modules or in standalone code.

14.1 Framework Tools

The following tools are part of the Allpix Squared framework and are located in the `src/tools` directory. They are intended as centralized components which can be shared between different modules rather than independent tools.

14.1.1 ROOT and Geant4 utilities

The framework provides a set of methods to ease the integration of ROOT and Geant4 in the framework. An important part is the extension of the custom conversion `to_string` and `from_string` methods from the internal string utilities (see Section 4.8) to support internal ROOT and Geant4 classes. This allows to directly read configuration parameters to these types, making the code in the modules both shorter and cleaner. In addition, more conversions functions are provided together with other useful utilities such as the possibility to display a ROOT vector with units and a thin wrapper for thread-safe ROOT histograms.

14.1.2 Geant4 Interface

The framework provides an interfacing library with Geant4 that provides alternative run managers to be used by modules interested in using Geant4 as follows:

1. `MTRunManager`: A run manager for multithreaded event processing. Internally, it creates thread-local managers to handle operations for each calling thread independently. It also maintains a stable seed distribution mechanism to ensure results are the same regardless of the number of threads that use the manager in parallel.
2. `RunManager`: A run manager for sequential event processing. It uses the same seeding mechanism as the multithreaded version so they can be used interchangeably depending on whether multithreading is enabled or not, while ensuring identical results.

The DepositionGeant4 module uses `MTRunManager` to be able to call the `BeamOn` method in parallel on multiple threads thus benefiting from the multithreading feature while the VisualizationGeant4 module uses `RunManager` to be able to visualize the particles passage through the detectors.

Note: The `MTRunManager` significantly reduces Geant4's run initialization time (this happens before every event in Allpix Squared) compared to Geant4's stock run managers (see Bugzilla/Geant4 2527 for details).
It is not feasible to implement this improvement in the single-threaded `RunManager` since it directly inherits from Geant4's stock `G4RunManager`. With this run manager, the run initialization time scales with the complexity of the geometry and can - *in the worst case scenario* - take significantly more time than the actual simulation itself.
Thus it is recommended to use multithreading when using Geant4 in Allpix Squared if allowed by the module configuration. Allpix Squared allows to use multithreading with only one worker as alternative to `multithreading=false`, though it is suggested to benchmark the performance for both cases to find the optimal setting for the given geometry.

14.1.3 Runge-Kutta integrator

A fast Eigen-powered [9] Runge-Kutta integrator is provided as a tool to numerically solve differential equations [22]. The Runge-Kutta integrator is designed in a generic way and supports multiple methods using different tableaus. It allows to integrate a system of equations in several steps with customizable step size. The step size can also be updated during the integration depending on the error of the Runge-Kutta method (if a tableau with error estimation is used).

The `GenericPropagation module` uses Runge-Kutta integrator with the Runge-Kutta-Fehlberg method (RK5 tableau). After the integrator has been created with the initial position of the charge carrier to be transported, the `step()` function allows to advance the simulation to the next step.

```
// Define lambda functions to compute the charge carrier velocity at
 ↪  each step
std::function<Eigen::Vector3d(double, Eigen::Vector3d)> carrier_velocity
 ↪  =
    [&](double, Eigen::Vector3d cur_pos) -> Eigen::Vector3d {...};

// Create the Runge-Kutta solver with a RK5 tableau, the carrier
 ↪  velocity function to be used
// as well as the initial timestep and position of the charge carrier
auto runge_kutta = make_runge_kutta(tableau::RK5, carrier_velocity,
 ↪  initial_timestep, position);

// Advance one step with the solver:
auto step = runge_kutta.step();
```

The `getValue()` and `setValue()` methods allow to retrieve, alter and update the position, e.g. to include additional displacements from diffusion processes.

14.1.4 Field Data Parser

A field parser tool is provided, which parses files stored in the INIT or APF file formats and returns field data on a three-dimensional grid. The number of field components per grid point is configurable via the constructor argument, e.g. `FieldQuantity::VECTOR` for a vector field or `FieldQuantity::SCALAR` for a scalar field map. The parsed field data is cached internally by the class, and if a file is requested a second time, the cached field is returned. In conjunction with a static instance of the field parser class in a module, this allows to share field data across multiple module instances.

```
class MyVectorFieldModule(...) : Module(...) {
private:
    void some_function(std::string canonical_path);
    // Define static field parser instance
    static FieldParser<double> field_parser_;
}

// Create static instance of field parser in the translation unit:
FieldParser<double>
 ↪  MyVectorFieldModule::field_parser_(FieldQuantity::VECTOR);

void MyVectorFieldModule::some_function(std::string canonical_path) {
    // Get vector field from file:
    auto field_data = field_parser_.getByFileName(canonical_path,
 ↪  "V/cm");
}
```

For the INIT format, the `getByFileName()` function of the parser takes the units in which the field data should be interpreted, and they are automatically converted to the framework base units described in Section 3.1. Fields in the APF format are always stored in framework base units and do not require conversion. The file path provided to the field parser should always be canonical, if the file is not found or cannot be parsed, a `std::runtime_error` exception is thrown.

The type of field data to be parsed is automatically deduced from the file content by checking for binary or ASCII text The field parser determines whether a file is text or binary by checking the first few bytes in the file. If every byte in that part of the file is non-null, the parser considers the file to be text and reads it as INIT file; otherwise it considers the file to be binary and parses the field as APF data.

14.2 Mesh Converter

This code takes adaptive meshes from finite-element simulations and transforms them into a regularly spaced grid for faster field value lookup as required by Monte Carlo simulations tools such as Allpix Squared. The input consists of two files, one containing the vertex coordinates of each input mesh node, the other providing the relevant field values associated to each of these vertices. One output file containing the regular mesh is produced. This tool can work in two different modes, the closest-neighbor mode and interpolation mode:

Simple Closest-Neighbor Search

In this mode, selected by setting the parameter `interpolate = false`, no interpolation of field values is performed, but for every output mesh point, the values from the closest neighbor of the input mesh are taken. In most cases this approach should produces reasonably precise results with a granularity similar to the respective adaptive mesh granularity in the respective region. The tool uses the Octree `findNeighbor` algorithm [100] to find the closest neighbor to the query point.

Barycentric Interpolation Method

In this mode, the regular mesh is created by scanning the model volume in regular X, Y and Z steps and using a barycentric interpolation method to calculate the respective electric field vector on the new point. The interpolation uses the four closest, no-coplanar, neighbor vertex nodes such, that the respective tetrahedron encloses the query point. For the neighbors search, the tool uses the Octree `radiusNeighbors` neighbor search algorithm [100].

14.2.1 File Formats

Input Data

Currently, this tool supports the TCAD DF-ISE data format and requires the `.grd` and `.dat` files as input. Here, the `.grd` file contains the vertex coordinates (3D or 2D) of each mesh node and the `.dat` file contains the value of each electric field vector component for each mesh node, grouped by model regions (such as silicon bulk or metal contacts). The regions are defined in the `.grd` file by grouping vertices into edges, faces and, consecutively, volumes or elements.

Output Data

This tools can produce output in two different formats, with the file extensions `.init` and `.apf`. Both file formats can be imported into Allpix Squared.

The **APF** (Allpix Squared Field) data format contains the field data in binary form and is therefore a bit more compact and can be read much faster. Whenever possible, this format should be preferred.

The **INIT** file is an ASCII text file with a format used by other tools such as PixelAV. Its header therefore contains several fields which are not used by Allpix Squared but need to be present nevertheless. The following example shows such a file header, important variables are marked with <...> while other fields are not interpreted and can be left untouched:

```
<first line: some descriptive text to identify the field or field
 ↪ source>
##SEED##  ##EVENTS##
##TURN## ##TILT## 1.0
0.00 0.0 0.00
```

```
<thickness in um> <size x in um> <size y in um> 0 0
0 0 <number of bins x> <number of bins y> <number of bins z> 0
```

After the header part, the data follows as list of individual nodes with three indices for x, y, and z coordinates at the beginning and the scalar or vector field components afterwards. For a vector field, this looks like:

```
<node.x> <node.y> <node.z> <observable.x> <observable.y> <observable.z>
```

whereas for a scalar field such as a weighting potential, only one field component is present:

```
<node.x> <node.y> <node.z> <observable>
```

14.2.2 Compilation

When compiling the Allpix Squared framework, the Mesh Converter is automatically compiled and installed in the Allpix Squared installation directory.

It is also possible to compile the converter separately as stand-alone tool within this directory:

```
mkdir build
cd build
cmake ..
make
```

It should be noted that the Mesh Converter depends on the core utilities of the Allpix Squared framework found in the directory `src/core/utils`. Thus, it is discouraged to move the converter code outside the repository as this directory would have to be copied and included in the code as well. Furthermore, updates are only distributed through the repository and new release versions of the Allpix Squared framework.

14.2.3 Features

- TCAD DF-ISE file format parser.
- Automatic determination of the input mesh dimensionality (2D/3D).
- Fast radius neighbor search for three-dimensional point clouds.
- Barycentric interpolation between non-regular mesh points.
- Several cuts available on the interpolation algorithm variables.
- Interpolated data visualization tool.

Parameters

- `model`: Field file format to use, can be **INIT** or **APF**, defaults to **APF** (binary format).
- `parser`: Parser class to interpret input data in. Currently, only **DF-ISE** is supported and used as default.
- `region`: Region name or list of region names to be meshed, such as `bulk` or `"bulk","epi"` (No default value; required parameter).

- `observable`: Observable to be interpolated, such as `ElectricField` (No default value; required parameter).
- `observable_units`: Units in which the observable is stored in the input file (No default value; required parameter).
- `interpolate`: Boolean switch to select either the barycentric interpolation method or the closest-neighbor method. Defaults to `true`, i.e. using the interpolation method.
- `initial_radius`: Initial node neighbors search radius in micro meters. Defaults to the minimal cell dimension of the final interpolated mesh.
- `radius_step`: Radius step if no neighbor is found (defaults to `0.5um`). Only used for barycentric interpolation.
- `max_radius`: Maximum search radius (default is `50um`). Only used for barycentric interpolation.
- `allow_coplanar_interpolation`: Allow the interpolation to use coplanar/colinear vertices if no full interpolation volume can be found after increasing the search radius and if more than 100 neighbors are found. Defaults to `false`. It should be noted that this feature is experimental and that it can produce `NaN` results for the interpolated field.
- `allow_failure`: Allow the interpolation of a single mesh point to fail, i.e. when no neighbors could be found. If set to `true`, the respective mesh element will be set to zero and the interpolation will continue, if `false` the interpolation will be aborted. Defaults to `false`. Only used for barycentric interpolation.
- `volume_cut`: Minimum volume for tetrahedron for non-coplanar vertices (defaults to minimum double value). Only used for barycentric interpolation.
- `divisions`: Number of divisions of the new regular mesh for each dimension, 2D or 3D vector depending on the `dimension` setting. Defaults to 100 bins in each dimension.
- `xyz`: Array to replace the system coordinates of the mesh. A detailed description of how to use this parameter is given below.
- `workers`: Number of worker threads to be used for the interpolation. Defaults to the available number of cores on the machine (hardware concurrency).
- `vector_field`: Select if the observable is a vector field or scalar field (Defaults to `true` matching the default observable `ElectricField`).
- `log_level`: Specifies the lowest log level which should be reported. Possible values are the same as for the Allpix Squared framework.

Usage

To run the program, the following command should be executed from the installation folder:

```
mesh_converter -f <file_prefix> [<options>] [<arguments>]
```

The converter will look for a configuration file with `<file_prefix>` and `.conf` extension. This default configuration file name can be replaced with the `-c` option. The list with options can be accessed using the `-h` option. Possible options and their default values are:

```
-f <file_prefix>       common prefix of DF-ISE grid (.grd) and data
 ↪  (.dat) files
-c <config_file>       configuration file setting mesh conversion
 ↪  parameters
-h                     display this help text
-l <file>              file to log to besides standard output (disabled
 ↪  by default)
-o <init_file_prefix>  output file prefix without .init (defaults to
 ↪  file name of <file_prefix>)
-v <level>             verbosity level (default reporting level is INFO)
```

Observables currently implemented for interpolation are: `ElectrostaticPotential`, `ElectricField`, `DopingConcentration`, `DonorConcentration` and `AcceptorConcentration`. The output INIT/APF file will be saved with the same file_prefix as the `.grd` and `.dat` files and the additional name suffix `_<observable>_interpolated` and the appropriate file extension, where `<observable>` is replaced with the selected quantity.

The new coordinate system of the mesh can be changed by providing an array for the *xyz* keyword in the configuration file. The first entry of the array, representing the new mesh x coordinate, should indicate the TCAD original mesh coordinate (x, y or z), and so on for the second (y) and third (z) array entry. For example, if one wants to have the TCAD x, y and z mesh coordinates mapped into the y, z and x coordinates of the new mesh, respectively, the configuration file should have `xyz = z x y`. If one wants to flip one of the coordinates, the minus symbol (-) can be used in front of one of the coordinates (such as `xyz = z x -y`).

The program can be used to convert 3D and 2D TCAD mesh files. Note that when converting 2D meshes, the x coordinate will be fixed to 1 and the interpolation will happen over the yz plane. The keyword mesh_tree can be used as a switch to enable or disable the creation of a root file with the original TCAD mesh points stored as a `ROOT::TTree` for later, fast, inspection.

14.2.4 Mesh Plotter

In addition to the Mesh Converter, the `mesh_plotter` tool can be used to visualize the new mesh interpolation results, from the installation folder as follows:

```
mesh_plotter -f <file_name> [<options>] [<arguments>]
```

The following command-line options are supported:

```
-f <file_name>         name of the interpolated file in APF or INIT
 ↪  format
-c <cut>               projection height index (default is mesh_pitch /
 ↪  2)
-h                     display this help text
-l                     plot with logarithmic scale if set
-o <output_file_name>  name of the file to output (default is
 ↪  efield.png)
-p <plane>             plane to be plotted. xy, yz or zx (default is yz)
```

```
-u <units>              units to interpret the field data in
-s                      parsed observable is a scalar field
```

The list with options and defaults is displayed with the `-h` option. In a 3D mesh, the plane to be plotted must be identified by using the option `-p` with argument *xy*, *yz* or *zx*, defaulting to *yz*. By default, the data is interpreted as a vector field, where graphs for all three components are created. Using the option `-s` enables the interpretation of a scalar field. The units for the field to interpreted in can be defined via the option `-u`. The number of mesh divisions in each dimension is automatically read from the `init/apf` file, by default the cut in the third dimension is done in the center but can be shifted using the `-c` option described above.

14.3 ROOT Analysis & Helper Macros

Collection of macros demonstrating how to analyze data generated by the framework. Currently contains a C++ macro to convert the TTree of objects to a tree containing standard data written by the framework. This is useful for analysis and comparisons with other frameworks. A simple example of how to read the output objects TTree using a Python macro is also included.

14.3.1 Comparison tree

Reads all required trees from the given file and binds their content to the objects defined by the framework. Then creates an output tree and binds every branch to a simple arithmetic type. Continues to loop over all events in the tree and converting the stored data from the various trees to the output tree. The final output tree contains branches for the cluster sizes, aspect ratios, accumulated charge per event, the track position from the Monte Carlo truth and the reconstructed track obtained from a center of gravity calculation using the charge values without additional corrections.

To construct a comparison tree using this macro, follow these steps:

- Open root with the data file attached like `root -l /path/to/data.root`
- Load the current library of objects with `.L path/to/libAllpixObjects.so`
- Build the macro with `.L path/to/constructComparisonTree.C++`
- Open a new file with `auto file = new TFile("output.root", "RECREATE")`
- Run the macro with `auto tree = constructComparisonTree(_file0, "name_of_dut")`
- Write the tree with `tree->Write()`

14.3.2 Analysis example

Analysis example demonstrating how to read data from ROOT TTrees, access attributes and access object history. The macro for this reads TTrees of `PixelHit` and `MCParticle` objects from an Allpix Squared data file created using the `ROOTObjectWriter`. Iterating over individual events, the position of every `PixelHit` is compared to the center of gravity position of all `MCParticles` and then only to those that are retrieved from the history of the `PixelHit`. Produces graphs for a 2D hitmap, the mentioned residuals and the signal

spectrum. As this macro does not perform a clustering, it is only a starting point for a data analysis.

Usage: * Open root with the data file attached like `root -l /path/to/data.root` * Load the current library of objects with `.L path/to/libAllpixObjects.so` * Build the macro with `.L path/to/analysisExample.C++` * Run the macro with `analysisExample(_file0, "name_of_detector")`

14.3.3 Remake project

Simple macro to show the possibility to recreate source files for legacy objects stored in ROOT data files from older versions of the framework. Can be used if the corresponding dynamic library for that particular version is not accessible anymore. It is however not possible to recreate methods of the objects and it is therefore not easily possible to reconstruct the stored history.

To recreate the project source files, the following commands should be executed:

- Open root with the data file attached like `root -l /path/to/data.root`
- Build the macro with `.L path/to/remakeProject.C++`
- Recreate the source files using `remakeProject(_file0, "output_dir")`

14.3.4 Recover Configuration Files

This macro allows to recover the full configuration of a simulation from a data file written by the ROOTObjectWriter module. It retrieves the stored key-value pairs and writes them into new files, including the framework and module configuration, the detector setup and the individual detector models with possibly overwritten parameters.

The simulation configuration can be recreated using the following command:

```
root -x 'recoverConfiguration.C("path/to/output/data.root",
                                "configuration.conf")'
```

Here, the first argument is the input data file produced by the ROOTObjectWriter, while the second argument is the output file name and path for the framework configuration. The detector setup and model files will be named as defined in the main configuration and are placed in the same folder.

14.3.5 Display Monte Carlo hits (Python)

Simple macro that reads the required trees to plot Monte Carlo hits in pixel versus the pixel charge. Loops over all events of the root file. A few relevant histograms are displayed at the end of the event loop. Requires PyROOT, numpy, matplotlib. To execute the script, run:

```
python3 display_mc_hits.py -l path/to/libAllpixObjects.so -f
 ↪  path/to/data.root -d <detector_name>
```

15 Appendix

The appendix contains supplementary information to the user manual.

15.1 Authors and Acknowledgments

Allpix Squared is developed and maintained by

- Paul Schütze, DESY, pschutze
- Simon Spannagel, DESY, simonspa

The following authors, in alphabetical order, have developed or contributed to Allpix Squared:

- Mohamed Moanis Ali, GSOC2019 Student, mmoanis
- Mathieu Benoit, BNL, mbenoit
- Thomas Billoud, Université de Montréal, tbilloud
- Tobias Bisanz, CERN, tbisanz
- Marco Bomben, Université de Paris, mbomben
- Koen van den Brandt, Nikhef, kvandenb
- Carsten Daniel Burgard, DESY, cburgard
- Maximilian Felix Caspar, DESY, mcaspar
- Liejian Chen, Institute of High Energy Physics Beijing, chenlj
- Manuel Alejandro Del Rio Viera, DESY, mdelriov
- Katharina Dort, University of Gießen, kdort
- Neal Gauvin, Université de Genève, ngauvin
- Lennart Huth, DESY, lhuth
- Daniel Hynds, University of Oxford, dhynds
- Francisco-Jose Iguaz-Gutierrez, Synchrotron SOLEIL, iguaz-gutierrez
- Maoqiang Jing, Institute of High Energy Physics Beijing, mjing
- Moritz Kiehn, Université de Genève, msmk
- Rafaella Eleni Kotitsa, CERN, rkotitsa
- Stephan Lachnit, DESY, slachnit
- Salman Maqbool, CERN Summer Student, smaqbool
- Stefano Mersi, CERN, mersi
- Ryuji Moriya, CERN Summer Student, University of Glasgow, rmoriya
- Sebastien Murphy, ETHZ, smurphy
- Andreas Matthias Nürnberg, KIT, nurnberg
- Sebastian Pape, TU Dortmund University, spape
- Marko Petric, CERN, mpetric
- Florian Michael Pitters, HEPHY, fpipper
- Radek Privara, Palacky University Olomouc, rprivara
- Nashad Rahman, The Ohio State University, nashadroid

- Sabita Rao, GSDocs2020 Student, srao
- Daniil Rastorguev, DESY, drastorg
- Edoardo Rossi, DESY, edrossi
- Jihad Saidi, Université de Genève, jisaidi
- Andre Sailer, CERN, sailer
- Tasneem Saleem, Synchrotron SOLEIL, TasneemSaleem
- Arka Santra, Weizman Institute, asantra
- Enrico Jr. Schioppa, Unisalento and INFN Lecce, schioppa
- Sebastian Schmidt, FAU Erlangen, schmidtseb
- Sanchit Sharma, Kansas State University, SanchitKratos
- Xin Shi, Institute of High Energy Physics Beijing, xshi
- Petr Smolyanskiy, Czech Technical Univbersity Prague, psmolyan
- Viktor Sonesten, GSOC2018 Student, tmplt
- Reem Taibah, Université de Paris, retaibah
- Ondrej Theiner, Charles University, otheiner
- Annika Vauth, University of Hamburg, avauth
- Mateus Vicente Barreto Pinto, CERN, mvicente
- Håkan Wennlöf, DESY, hwennlof
- Andy Wharton, Lancaster University, awharton
- Morag Williams, University of Glasgow, williamm
- Koen Wolters, kwolters

15.2 List of Tests

15.2.1 Framework Functionality Tests

Currently implemented framework functionality tests comprise:

- `core/test_01-10_globalconfig_log_prng`: tests the possibility of logging individual random numbers used during the simulation
- `core/test_01-1_globalconfig_detectors`: test the framework behavior in case of a non-existent detector setup description file
- `core/test_01-2_globalconfig_modelpaths`: tests the correct parsing of additional model paths and the loading of the detector model.
- `core/test_01-3_globalconfig_log_format`: switches the logging format.
- `core/test_01-4_globalconfig_log_level`: sets a different logging verbosity level.
- `core/test_01-5_globalconfig_log_file`: configures the framework to write log messages into a file.
- `core/test_01-6_globalconfig_missing_model`: tests the behavior of the framework in case of a missing detector model file.
- `core/test_01-7_globalconfig_random_seed`: sets a defined random seed to start the simulation with.
- `core/test_01-8_globalconfig_random_seed_core`: sets a defined seed for the core component seed generator, e.g. used for misalignment.
- `core/test_01-9_globalconfig_librarydirectory`: tests the correct parsing and usage of additional library loading paths.

- core/test_02-1_specialization_unique_name: tests the framework behavior for an invalid module configuration: attempt to specialize a unique module for one detector instance.
- core/test_02-2_specialization_unique_type: tests the framework behavior for an invalid module configuration: attempt to specialize a unique module for one detector type.
- core/test_02-3_specialization_name: tests module instance specialization by name
- core/test_02-4_specialization_type: tests module instance specialization by type
- core/test_03-10-geometry_unique_detectors: tests if the case of multiple detectors with the same name is correctly caught
- core/test_03-11-geometry_unique_passive: tests if the case of multiple passive volumes with the same name is correctly caught
- core/test_03-12_geometry_invalid_name: tests if invalid detector names are correctly caught
- core/test_03-1_geometry_g4_coordinate_system: ensures that the Allpix Squared and Geant4 coordinate systems and transformations are identical.
- core/test_03-2_geometry_rotations: tests the correct interpretation of rotation angles in the detector setup file.
- core/test_03-3_geometry_misaligned: tests the correct calculation of misalignments from alignment precisions given in the detector setup file.
- core/test_03-4_geometry_overwrite: checks that detector model parameters are overwritten correctly
- core/test_03-6_geometry_overlap: checks for correct detection of volume overlaps in the geometry
- core/test_03-7_geometry_wrapper: checks for correct treatment of geometry wrappers and overlap calculations
- core/test_03-8_geometry_noposition: test the framework behavior with a detector with no position provided in the geometry
- core/test_03-9_geometry_nodetector: tests if missing detectors requested in individual module instances are correctly detected
- core/test_04-10_configuration_invalid_combination: tests if invalid configuration key combinations are correctly detected and reported
- core/test_04-11_configuration_missing_key: tests if missing mandatory configuration keys are correctly detected and reported
- core/test_04-12_configuration_matrix: tests if matrix values in configuration files are correctly parsed and interpreted
- core/test_04-13_configuration_matrix_brackets: tests if invalid or missing brackets of matrix values in configurations files are detected and reported
- core/test_04-1_module_config_cli_change: tests whether single configuration values can be overwritten by options supplied via the command line.
- core/test_04-2_module_config_cli_nochange: tests whether command line options are correctly assigned to module instances and do not alter other values.
- core/test_04-3_configuration_imbalanced_brackets: tests whether imbalanced brackets in configuration values are properly detected.
- core/test_04-4_detector_config_cli_change: tests whether detector options can be overwritten from the command line.
- core/test_04-5_module_config_cli_detectors: tests whether framework pa-

rameters are properly parsed from the command line.
- `core/test_04-6_module_config_double_unique`: tests whether a double definition of a unique module is detected.
- `core/test_04-7_module_config_empty_filter`: tests the framework behavior with an empty filter.
- `core/test_04-8_configuration_unused_key`: tests the detection of unused configuration keys in the global configuration section.
- `core/test_04-9_configuration_unused_key_module`: tests the detection of unused configuration keys in a module configuration section.
- `core/test_05-1_overwrite_same_denied`: tests whether two modules writing to the same file is disallowed if overwriting is denied.
- `core/test_05-2_overwrite_module_allowed`: tests whether two modules writing to the same file is allowed if the last one re-enables overwriting locally.
- `core/test_05-3_overwrite_detector_module`: tests whether two detector modules with different priorities are handled correctly.
- `core/test_05-4_overwrite_detector_module_reverse`: tests whether two detector modules with different priorities are handled correctly (reverse order).
- `core/test_05-5_overwrite_module_allow_io`: tests different input / output configurations with module overwriting.
- `core/test_06-10_multithreading_physics_singlethr`: tests the reproducibility in case of multithreading disabled.
- `core/test_06-11_multithreading_oneworkers`: tests the framework response in case too few workers are enabled.
- `core/test_06-1_multithreading`: checks if multithreading can be enabled.
- `core/test_06-2_multithreading_cli`: checks if multithreading can be enabled from the command line.
- `core/test_06-3_multithreading_concurrency`: tests if the number of workers can be configured.
- `core/test_06-4_multithreading_zeroworkers`: tests the framework response in case too few workers are enabled.
- `core/test_06-5_multithreading_buffers`: tests if the module buffer depth can be configured properly.
- `core/test_06-6_multithreading_impossible`: tests the framework response in case a module without multithreading capabilities has been enabled.
- `core/test_06-7_multithreading_disabled`: tests the framework response to explicitly disabling multithreading.
- `core/test_06-8_multithreading_buffered`: tests the reproducibility in case of a sequential module.
- `core/test_06-9_multithreading_physics`: tests the reproducibility in case of multithreading enabled.
- `core/test_07-1_catch_exception`: checks the correct propagation of exceptions with multithreading enabled.
- `core/test_07-2_catch_exception_nomt`: checks the correct propagation of exceptions with multithreading disabled.
- `core/test_08-10_physics_recombination_srh`: tests selection of recombination model “srh”
- `core/test_08-11_physics_recombination_auger`: tests selection of recombination model “auger”
- `core/test_08-12_physics_recombination_combined`: tests selection of recombi-

nation model “combined”
- core/test_08-13_physics_mobility_custom: tests selection of a custom mobility model
- core/test_08-1_physics_mobility_canali: tests selection of mobility model “canali”
- core/test_08-2_physics_mobility_hamburg: tests selection of mobility model “hamburg”
- core/test_08-3_physics_mobility_hamburg_highfield: tests selection of mobility model “hamburg_highfield”
- core/test_08-4_physics_mobility_masetti: tests selection of mobility model “masetti”
- core/test_08-5_physics_mobility_masetti_canali: tests selection of mobility model “masetti_canali”
- core/test_08-6_physics_mobility_arora: tests selection of mobility model “arora”
- core/test_08-7_physics_mobility_invalid: tests if a selection of an invalid or non-existing mobility model is correctly detected and reported
- core/test_08-8_physics_mobility_doping: tests if the requirement of a doping profile for some mobility models is correctly detected and reported as error
- core/test_08-9_physics_mobility_jacoboni: tests selection of mobility model “jacoboni”
- core/test_9-1_executable_version: tests if the allpix executable correctly reports its version if requested
- core/test_9-2_executable_help: tests if the allpix executable correctly prints the help if requested
- core/test_9-3_executable_loglevel_invalid: tests if the allpix executable correctly reports invalid log levels set from the command line
- core/test_9-4_executable_unrecognized_argument: tests if the allpix executable correctly reports unrecognized command line arguments

15.2.2 Module Functionality Tests

The following module functionality tests are currently performed:

- modules/CSADigitizer/01-pseudopulse: checks the outcome of a digitization by the CSA of a pseudo-pulse generated from arrival times.
- modules/CSADigitizer/02-tot_pseudopulse: checks the outcome of a digitization by the CSA of a pseudo-pulse generated from arrival times and conversion to time-over-threshold units.
- modules/CSADigitizer/03-custom: tests initialization of a custom response function and its parameters
- modules/CSADigitizer/04-custom_mupix: tests the digitization with a custom response function
- modules/CapacitiveTransfer/01-transfer: tests the coupling of charge into neighbor pixels using a coupling matrix
- modules/CorryvreckanWriter/01-corry: ensures proper functionality of the Corryvreckan file writer module. The monitored output comprises the coordinates of the pixel produced in the simulation.

- `modules/CorryvreckanWriter/02-mc`: ensures the correct storage of Monte Carlo truth particle information in the Corryvreckan file writer module by monitoring the local coordinates of the MC particle associated to the pixel hit.
- `modules/DefaultDigitizer/01-charge`: digitizes the transferred charges to simulate the front-end electronics. The monitored output of this test comprises the total charge for one pixel including noise contributions and the smeared threshold it is compared to.
- `modules/DefaultDigitizer/02-qdc`: digitizes the transferred charges and tests the conversion into QDC units. The monitored output comprises the converted charge value in units of QDC counts.
- `modules/DefaultDigitizer/03-gain`: digitizes the transferred charges and tests the amplification process by monitoring the total charge after signal amplification and smearing.
- `modules/DefaultDigitizer/04-toa`: digitizes the signal and calculates the time-of-arrival of the particle by checking when the threshold was crossed.
- `modules/DefaultDigitizer/05-tdc`: digitizes the signal and test the conversion of time-of-arrival to TDC units.
- `modules/DefaultDigitizer/06-saturation`: tests the front-end saturation functionality
- `modules/DefaultDigitizer/07-gainfunction`: digitizes the transferred charges and tests the amplification process using a custom (surrogate) gain function.
- `modules/DefaultDigitizer/08-gain-gainfunction`: tests the correct detection of a simultaneous configuration of a default gain and a custom gain function.
- `modules/DefaultDigitizer/09-gainfunction-param`: tests the correct detection of an incorrect number of parameters provided for a custom gain function.
- `modules/DepositionCosmics/01-sealevel`: run basic simulation of cosmics shower
- `modules/DepositionCosmics/02-altitude`: check if the target simulation altitude can be configured
- `modules/DepositionCosmics/03-subbox`: test if the correct subbox letngth for the simulated shower is calculated from the detector model and world volume
- `modules/DepositionCosmics/04-noneutrons`: check if the emission of neutrons can be switched off effectively
- `modules/DepositionCosmics/05-latitude`: check if the latitude can be changed effectively
- `modules/DepositionCosmics/06-date`: check if the simulated date of the shower observation can be changed effectively
- `modules/DepositionCosmics/07-resettime`: test if the particle emission time can correctly be reset to zero when configured
- `modules/DepositionCosmics/08-maxparticles`: test if the maximum number of shower particles can be limited correctly
- `modules/DepositionGeant4/01-deposit`: executes the charge carrier deposition module. This will invoke Geant4 to deposit energy in the sensitive volume. The monitored output comprises the exact number of charge carriers deposited in the detector.
- `modules/DepositionGeant4/02-mc`: executes the charge carrier deposition module as the previous tests, but monitors the type, entry and exit point of the Monte Carlo particle associated to the deposited charge carriers.
- `modules/DepositionGeant4/03-track`: executes the charge carrier deposition

module as the previous tests, but monitors the start and end point of one of the Monte Carlo tracks in the event.

- `modules/DepositionGeant4/04-source_point`: tests the point source in the charge carrier deposition module by monitoring the deposited charges.
- `modules/DepositionGeant4/05-source_square`: tests the square source in the charge carrier deposition module by monitoring the deposited charges.
- `modules/DepositionGeant4/06-source_sphere`: tests the sphere source in the charge carrier deposition module by monitoring the deposited charges.
- `modules/DepositionGeant4/07-source_macro`: tests the G4 macro source in the charge carrier deposition module using the macro file `source_macro_test.txt`, monitoring the deposited charges.
- `modules/DepositionGeant4/08-fano`: tests the simulation of fluctuations in charge carrier generation by monitoring the total number of generated carrier pairs when altering the Fano factor.
- `modules/DepositionGeant4/09-ions`: tests if custom ions can be fored to decay immediately
- `modules/DepositionGeant4/10-all_tracks`: runs a single Geant4 event and retrieves all tracks from the event, including those without connection to the sensor volume.
- `modules/DepositionGeant4/11-tracking_verbosity`: executes the charge carrier deposition module with high tracking verbosity level. The individual tracking steps are observed as output.
- `modules/DepositionGenerator/01-hepmcascii`: tests reading in a HepMC3 ASCII file generated using the HepMC3 Python tools
- `modules/DepositionGenerator/02-hepmcttree`: tests reading in a HepMC3 ROOTIO TTree file generated using the HepMC3 Python tools
- `modules/DepositionLaser/01_data_read`: tests reading of physical constants from the datafile
- `modules/DepositionLaser/02_multi_detectors`: tests deposition in multiple detectors
- `modules/DepositionLaser/03_passive`: tests termination of tracks in passive objects
- `modules/DepositionLaser/04_1_refraction`: tests refraction on silicon-air interface
- `modules/DepositionLaser/04_2_refraction`: tests refraction on silicon-air interface
- `modules/DepositionLaser/04_3_refraction`: tests refraction on silicon-air interface
- `modules/DepositionLaser/04_4_refraction`: tests refraction on silicon-air interface
- `modules/DepositionLaser/04_5_refraction`: tests refraction on silicon-air interface
- `modules/DepositionLaser/04_6_refraction`: tests refraction on silicon-air interface
- `modules/DepositionLaser/05_user_optics`: tests reading of physical constants from the datafile
- `modules/DepositionLaser/06_groups`: tests bucketing of photons
- `modules/DepositionPointCharge/01-point`: tests the deposition of a point charge at a specified position, checks the position of the deposited charge carrier in

global coordinates.

- `modules/DepositionPointCharge/02-scan`: tests the scan of a pixel volume by depositing charges for a given number of events, check for the calculated voxel size.
- `modules/DepositionPointCharge/03-scan_cube`: tests the calculation of the scanning points by monitoring the warning of the number of events is not a perfect cube.
- `modules/DepositionPointCharge/04-mip`: tests the deposition of charges along a line by monitoring the calculated step size and number of charge carriers deposited per step.
- `modules/DepositionPointCharge/05-mip_position`: tests the generation of the Monte Carlo particle when depositing charges along a line by monitoring the start and end positions of the particle.
- `modules/DepositionPointCharge/05-spot`: tests the deposition of charge carriers around a fixed position with a Gaussian distribution.
- `modules/DepositionReader/01-csv`: tests reading in a CSV file generated according to the specifications
- `modules/DepositionReader/02-root`: tests reading in a ROOT file generated according to the specifications
- `modules/DepositionReader/03-no_time_csv`: tests reading in a CSV file generated according to the specifications and without timing information
- `modules/DepositionReader/04-no_time_root`: tests reading in a ROOT file generated according to the specifications and without timing information
- `modules/DepositionReader/05-no_mcp_csv`: tests reading in a CSV file generated according to the specifications and without Monte Carlo particle information
- `modules/DepositionReader/06-no_mcp_root`: tests reading in a ROOT file generated according to the specifications and without Monte Carlo particle information
- `modules/DepositionReader/07-root_notree`: tests if a missing ROOT tree is detected correctly
- `modules/DepositionReader/08-root_branches`: tests if the number of branch names configured is correctly calculated
- `modules/DepositionReader/09-root_branch_wrong`: tests if missing or wrongly named branches are properly detected and reported
- `modules/DepositionReader/10-wrong_detector`: tests if depositions from a detector not present in the current simulation are ignored correctly
- `modules/DepositionReader/11-outside_sensor`: tests if deposited energies outside the active sensor volume of the detector are ignored correctly
- `modules/DepositionReader/12-end_of_file`: tests if a premature end of the CSV input file is correctly reported and the simulation terminated properly
- `modules/DepositionReader/13-end_of_tree`: tests if a premature end of the input ROOT tree is correctly reported and the simulation terminated pr
- `modules/DepositionReader/14-truncate_csv`: tests if detector name truncation works as expected in CSV files
- `modules/DepositionReader/15-truncate_root`: tests if detector name truncation works as expected in ROOT trees
- `modules/DepositionReader/16-mcp_ordered`: tests if parent relations of Monte Carlo particles are correctly determined and recorded
- `modules/DepositionReader/17-mcp_unordered`: tests if parent relations of Monte Carlo particles are correctly determined and recorded even if they appear unordered in the input data

- `modules/DetectorHistogrammer/01-histogramming`: tests the detector histogramming module and its clustering algorithm. The monitored output comprises the total number of clusters.
- `modules/DopingProfileReader/01-regions`: tests if a doping profile can be configured by means of different concentration regions in depth
- `modules/DopingProfileReader/02-constant`: tests if a constant doping profile can be configured
- `modules/ElectricFieldReader/01-linear`: creates a linear electric field in the constructed detector by specifying the bias and depletion voltages. The monitored output comprises the calculated effective thickness of the depleted detector volume.
- `modules/ElectricFieldReader/02-mesh`: loads an INIT file containing a TCAD-simulated electric field and applies the field to the detector model. The monitored output comprises the number of field cells for each pixel as read and parsed from the input file.
- `modules/ElectricFieldReader/03-linear_depth`: creates a linear electric field in the constructed detector by specifying the applied bias voltage and a depletion depth. The monitored output comprises the calculated effective thickness of the depleted detector volume.
- `modules/ElectricFieldReader/04-linear_depletion_side`: checks that depleting from the sensor backside is possible.
- `modules/ElectricFieldReader/05-constant`: tests the possibility of setting a constant electric field
- `modules/ElectricFieldReader/06-mutually_exclusive`: tests that the mutually exclusive parameters depletion_depth and depletion_voltage cannot be used together
- `modules/ElectricFieldReader/07-sensor_thickness`: tests that the depltion thickness cannot be larger than the sensor thickness
- `modules/ElectricFieldReader/08-mesh_mapping_quadrant`: tests the possibility of configuring an offset for the mesh
- `modules/ElectricFieldReader/09-mesh_mapping_half`: tests mapping of a half-field onto the pixel plane
- `modules/ElectricFieldReader/10-mesh_mapping_inverse`: tests mapping of a field entered around a pixel corner onto the pixel plane
- `modules/ElectricFieldReader/11-mesh_scale`: tests the possibility to scale the mesh in x and y
- `modules/ElectricFieldReader/12-parabolic`: tests the parabolic electric field
- `modules/ElectricFieldReader/13-parabolic_minimum_pos`: tests if the minimum position is required to be within the defined electric field region
- `modules/ElectricFieldReader/14-custom_1d`: tests the possibility of setting a one-dimensional custom electric field function
- `modules/ElectricFieldReader/15-custom_3d`: tests the possibility of setting a three-dimensional custom electric field function
- `modules/ElectricFieldReader/16-custom_functions`: tests that the custom function either requires one or three components
- `modules/ElectricFieldReader/17-custom_parameters_1d`: tests that the number of parameters provided to custom one-dimensional field functions needs to match
- `modules/ElectricFieldReader/18-custom_parameters_3d`: tests that the number of parameters provided to custom three-dimensional field functions needs to

match

- `modules/ElectricFieldReader/19-linear-largefield`: tests if very high bias voltages are correctly detected and reported as warning
- `modules/ElectricFieldReader/20-mesh_offset_negative`: tests that the mesh offset cannot be negative
- `modules/ElectricFieldReader/21-mesh_offset_large`: tests that the mesh offset cannot be larger than one pixel pitch
- `modules/GenericPropagation/01-propagation`: uses the Runge-Kutta-Fehlberg integration of the equations of motion implemented in the drift-diffusion model to propagate the charge carriers to the implants. The monitored output comprises the total number of charges moved, the number of integration steps taken and the simulated propagation time.
- `modules/GenericPropagation/02-magnetic`: uses the Runge-Kutta-Fehlberg integration of the equations of motion implemented in the drift-diffusion model to propagate the charge carriers to the implants under the influence of a constant magnetic field. The monitored output comprises the total number of charges moved, the number of integration steps taken and the simulated propagation time.
- `modules/GenericPropagation/03-lifetime`: test recombination of charge carriers during drift
- `modules/GenericPropagation/04-mobility_unsuitable`: tests if the selection of doping-dependent mobility models without doping information is caught correctly
- `modules/GenericPropagation/05-mobility_nomodel`: tests if non-existing mobility models selected in the configuration file are detected
- `modules/GenericPropagation/06-no_lifetime`: tests the fallback of infinite charge carrier lifetime in case no recombination model is chosen
- `modules/GenericPropagation/07-lifetime_unsuitable`: tests if the selection of doping-dependent recombination models without doping information is caught correctly
- `modules/GenericPropagation/08-lifetime_nomodel`: tests if non-existing recombination models selected in the configuration file are detected
- `modules/GenericPropagation/09-no_trapping`: tests the fallback of infinite charge carrier lifetime in case no trapping model is chosen
- `modules/GenericPropagation/10-trapping_nomodel`: tests if non-existing trapping models selected in the configuration file are detected
- `modules/GenericPropagation/11-trapping_custom`: tests functionality of custom trapping model
- `modules/GenericPropagation/12-max_charge_groups`: test the automatic scaling of charge per step when transport of a deposit would exceed the set max charge groups
- `modules/GenericPropagation/13-impact-ionization`: tests functionality of impact ionizations in high-field regions
- `modules/GenericPropagation/14-impact-missing-holes`: checks that a warning is printed in cases not all impact ionization carriers would be propagated
- `modules/GenericPropagation/15-impact-secondary`: tests generation of impact ionization charge carriers of opposite type
- `modules/GenericPropagation/16-lifetime-custom`: tests if custom recombination models work properly
- `modules/GeometryBuilderGeant4/01-build`: takes the provided detector setup and builds the Geant4 geometry from the internal detector description. The

monitored output comprises the calculated wrapper dimensions of the detector model.

- `modules/GeometryBuilderGeant4/02-addpoint`: ensures the module adds corner points of the passive material in a correct way.
- `modules/GeometryBuilderGeant4/03-addpoint_rotate`: ensures proper rotation of the position of the corner points of the passive material.
- `modules/GeometryBuilderGeant4/04-mothervolume`: ensures placing a detector inside a passive material will not cause overlapping materials.
- `modules/GeometryBuilderGeant4/05-worldvolume`: ensures the added corner points of the passive material increase the world volume accordingly.
- `modules/GeometryBuilderGeant4/06-same_materials`: tests if a warning will be thrown if the material of the passive material is the same as the material of the world volume.
- `modules/GeometryBuilderGeant4/07-radial_build`: builds the Geant4 geometry of a radial strip detector. The monitored output is the world size based on the wrapper dimensions and rotation
- `modules/GeometryBuilderGeant4/08-build_sensor`: takes the provided detector setup and builds the Geant4 geometry from the internal detector description. The monitored output comprises the calculated sensor position.
- `modules/GeometryBuilderGeant4/09-build_support`: takes the provided detector setup and builds the Geant4 geometry from the internal detector description. The monitored output comprises the calculated position of the support layer.
- `modules/GeometryBuilderGeant4/10-build_bumps`: takes the provided detector setup and builds the Geant4 geometry from the internal detector description. The monitored output comprises the calculated position of the bump bonding layer.
- `modules/GeometryBuilderGeant4/11-build_chip`: takes the provided detector setup and builds the Geant4 geometry from the internal detector description. The monitored output comprises the calculated position of the ASIC.
- `modules/GeometryBuilderGeant4/12-multithreading-oneworker`: tests the use the MTRunManager with workers=1
- `modules/LCIOWriter/01-lcio`: ensures proper functionality of the LCIO file writer module. Similar to the above test, the correct conversion of PixelHits (coordinates and charge) is monitored.
- `modules/LCIOWriter/02-detector_assignment`: exercises the assignment of detector IDs to Allpix Squared detectors in the LCIO output file. A fixed ID and collection name is assigned to the simulated detector.
- `modules/LCIOWriter/03-no_mc_truth`: ensures that simulation results are properly converted to LCIO and stored even without the Monte Carlo truth information available.
- `modules/MagneticFieldReader/01-constant`: creates a constant magnetic field for the full volume and applies it to the geometryManager. The monitored output comprises the message for successful application of the magnetic field.
- `modules/MagneticFieldReader/02-local`: checks that the local magnetic field including the detector rotation is correct.
- `modules/ProjectionPropagation/01-project`: projects deposited charges to the implant side of the sensor. The monitored output comprises the total number of charge carriers propagated to the sensor implants.
- `modules/ProjectionPropagation/02-lifetime`: projects deposited charges to the implant side of the sensor with a reduced integration time to ignore some charge

carriers. The monitored output comprises the total number of charge carriers propagated to the sensor implants.

- `modules/PulseTransfer/01-pseudopulse`: tests the calculation of induced signals based on the arrival time of charge carriers at the sensor surface.
- `modules/PulseTransfer/02-pseudopulse-ancestors`: tests the calculation of induced signals based on the arrival time of charge carriers at the sensor surface.
- `modules/RCEWriter/02-write`: ensures proper functionality of the RCE file writer module. The correct conversion of the PixelHit position and value is monitored by the test's regular expressions.
- `modules/ROOTObjectReader/01-reading`: tests the capability of the framework to read data back in and to dispatch messages for all objects found in the input tree. The monitored output comprises the total number of objects read from all branches.
- `modules/ROOTObjectReader/02-seed_mismatch`: tests the capability of the framework to detect different random seeds for misalignment set in a data file to be read back in. The monitored output comprises the error message including the two different random seed values.
- `modules/ROOTObjectReader/03-seed_ignore`: tests if core random seeds are properly ignored by the ROOTObjectReader module if requested by the configuration. The monitored output comprises the warning message emitted if a difference in seed values is discovered.
- `modules/ROOTObjectWriter/01-write`: ensures proper functionality of the ROOT file writer module. It monitors the total number of objects and branches written to the output ROOT trees.
- `modules/SimpleTransfer/01-transfer`: tests the transfer of charges from sensor implants to readout chip. The monitored output comprises the total number of charges transferred and the coordinates of the pixels the charges have been assigned to.
- `modules/SimpleTransfer/02-implant`: tests the transfer of charges from sensor implants to readout chip in case sensor implants have been defined in the detector model.
- `modules/SimpleTransfer/03-radial_transfer`: tests the transfer of charges from sensor implants to readout chip in a radial strip detector. The monitored output comprises the total number of charges transferred and the coordinates of the pixels the charges have been assigned to.
- `modules/TextWriter/01-write`: ensures proper functionality of the ASCII text writer module by monitoring the total number of objects and messages written to the text file.
- `modules/TransientPropagation/01-propagation`: uses the Runge-Kutta-Fehlberg integration of the equations of motion implemented in the drift-diffusion model to propagate the charge carriers to the implants. The total induced charge is monitored.
- `modules/TransientPropagation/02-magnetic`: uses the Runge-Kutta-Fehlberg integration of the equations of motion implemented in the drift-diffusion model to propagate the charge carriers to the implants under the influence of a constant magnetic field. The monitored output comprises the total number of charges moved, the number of integration steps taken and the simulated propagation time.
- `modules/TransientPropagation/03-lifetime`: test recombination of charge carriers during drift

- `modules/TransientPropagation/04-mobility_unsuitable`: tests if the selection of doping-dependent mobility models without doping information is caught correctly
- `modules/TransientPropagation/05-mobility_nomodel`: tests if non-existing mobility models selected in the configuration file are detected
- `modules/TransientPropagation/06-no_lifetime`: tests the fallback of infinite charge carrier lifetime in case no recombination model is chosen
- `modules/TransientPropagation/07-lifetime_unsuitable`: tests if the selection of doping-dependent recombination models without doping information is caught correctly
- `modules/TransientPropagation/08-lifetime_nomodel`: tests if non-existing recombination models selected in the configuration file are detected
- `modules/TransientPropagation/09-no_trapping`: tests the fallback of infinite charge carrier lifetime in case no trapping model is chosen
- `modules/TransientPropagation/10-trapping_nomodel`: tests if non-existing trapping models selected in the configuration file are detected
- `modules/TransientPropagation/11-trapping_custom`: tests functionality of custom trapping model
- `modules/TransientPropagation/12-max_charge_groups`: test the automatic scaling of charge per step when transport of a deposit would exceed the set max charge groups
- `modules/TransientPropagation/13-impact-ionization`: tests functionality of impact ionizations in high-field regions
- `modules/TransientPropagation/14-impact-timestep`: checks that a warning is printed in cases the timestep is too coarse for impact ionization
- `modules/TransientPropagation/15-impact-secondary`: tests generation of impact ionization charge carriers of opposite type
- `modules/TransientPropagation/16-impact-gain`: tests final gain of single event
- `modules/TransientPropagation/17-linear_field`: checks that an error is printed if the module is sued with a linear electric field.
- `modules/TransientPropagation/18-no_weighting_potential`: uses the Runge-Kutta-Fehlberg integration of the equations of motion implemented in the drift-diffusion model to propagate the charge carriers to the implants. The monitored output comprises the total number of charges moved, the number of integration steps taken and the simulated propagation time.
- `modules/TransientPropagation/19-mt_linegraphs`: uses the Runge-Kutta-Fehlberg integration of the equations of motion implemented in the drift-diffusion model to propagate the charge carriers to the implants. The total induced charge is monitored.
- `modules/WeightingPotentialReader/01-pad`: tests the on-the-fly generation of the weighting potential following the plane condenser with "pad over infinite plane" approach.

15.2.3 Performance Tests

Current performance tests comprise:

- `performance/test_01_deposition`: tests the performance of charge carrier deposition in the sensitive sensor volume using Geant4. A stepping length of 1.0 um is chosen, and 10000 events are simulated. The addition of an electric field and

the subsequent projection of the charges are necessary since Allpix Squared would otherwise detect that there are no recipients for the deposited charge carriers and skip the deposition entirely.

- `performance/test_02-1_propagation_generic`: tests the very critical performance of the drift-diffusion propagation of charge carriers, as this is the most computing-intense module of the framework. Charge carriers are deposited and a propagation with 10 charge carriers per step and a fine spatial and temporal resolution is performed. The simulation comprises 500 events.
- `performance/test_02-2_propagation_project`: tests the projection of charge carriers onto the implants, taking into account the diffusion only. Since this module is less computing-intense, a total of 5000 events are simulated, and charge carriers are propagated one-by-one.
- `performance/test_02-3_propagation_generic_multithread`: tests the performance of multithreaded simulation. It utilizes the very same configuration as performance test 02-1 but in addition enables multithreading with four worker threads.
- `performance/test_03_multithreading`: tests the performance of the framework when using multithreading with 4 workers to simulate 500 events. It uses a similar configuration as the example configuration.

Bibliography

[1] S. Agostinelli et al. “Geant4 - a simulation toolkit”. In: *Nucl. Instr. Meth. A* 506.3 (2003), pp. 250–303. ISSN: 0168-9002. DOI: 10.1016/S0168-9002(03)01368-8.

[2] Rene Brun and Fons Rademakers. “ROOT - An Object Oriented Data Analysis Framework”. In: *AIHENP’96 Workshop, Lausanne*. Vol. 389. Sept. 1996, pp. 81–86.

[3] Mathieu Benoit and John Idarraga. *The AllPix Simulation Framework*. Mar. 2017. URL: https://twiki.cern.ch/twiki/bin/view/Main/AllPix.

[4] Mathieu Benoit, John Idarraga, and Samir Arfaoui. *AllPix. Generic simulation for pixel detectors*. URL: https://github.com/ALLPix/allpix.

[5] Daniel Hynds, Simon Spannagel, and Koen Wolters. *The Allpix Squared Code Documentation*. Aug. 2017. URL: https://allpix-squared.docs.cern.ch/reference/.

[6] *The Allpix Squared Project Issue Tracker*. July 2017. URL: https://gitlab.cern.ch/allpix-squared/allpix-squared/issues.

[7] *The Allpix Squared Project Forum*. Dec. 2018. URL: https://cern.ch/allpix-squared-forum.

[8] Jens Maurer and Steven Watanabe. *The Boost Random Number Library*. 2000. URL: https://www.boost.org/doc/libs/1_75_0/doc/html/boost_random/reference.html.

[9] Gaël Guennebaud, Benoît Jacob, et al. *Eigen v3*. 2010. URL: http://eigen.tuxfamily.org.

[10] C++ Reference. *Compiler support for C++17*. May 2021. URL: https://en.cppreference.com/w/cpp/compiler_support/17.

[11] Rene Brun and Fons Rademakers. *Building ROOT*. URL: https://root.cern.ch/building-root.

[12] Geant4 Collaboration. *Geant4 Installation Guide. Building and Installing Geant4 for Users and Developers*. 2016. URL: http://geant4.web.cern.ch/geant4/UserDocumentation/UsersGuides/InstallationGuide/html/.

[13] *The Allpix Squared Project Repository*. Aug. 2017. URL: https://gitlab.cern.ch/allpix-squared/allpix-squared/.

[14] S. Aplin et al. “LCIO: A persistency framework and event data model for HEP”. In: *Nuclear Science Symposium and Medical Imaging Conference (NSS/MIC), IEEE*. Anaheim, CA, Oct. 2012, pp. 2075–2079. DOI: 10.1109/NSSMIC.2012.6551478.

[15] Simon Spannagel. *The Allpix Squared Docker Container Registry*. Mar. 2018. URL: https://gitlab.cern.ch/allpix-squared/allpix-squared/container_registry.

[16] C. Aguado Sanchez et al. “CVMFS - a file system for the CernVM virtual appliance”. In: *XII Advanced Computing and Analysis Techniques in Physics Research (ACAT08)*. Vol. ACAT08. 2008, p. 052.

[17] X. Llopart et al. “Timepix, a 65k programmable pixel readout chip for arrival time, energy and/or photon counting measurements”. In: *Nucl. Instr. Meth. A* 581.1 (2007). VCI 2007, pp. 485–494. ISSN: 0168-9002. DOI: 10.1016/j.nima.2007.08.079.

[18] Geant4 Collaboration. *Geant4 User's Guide for Application Developers. Visualization*. 2016. URL: https://geant4.web.cern.ch/geant4/UserDocumentation/UsersGuides/ForApplicationDeveloper/html/ch08.html.

[19] Rene Brun and Fons Rademakers. *ROOT User's Guide. Trees*. URL: https://root.cern.ch/root/htmldoc/guides/users-guide/Trees.html.

[20] Rene Brun and Fons Rademakers. *ROOT User's Guide. Input/Output*. URL: https://root.cern.ch/root/htmldoc/guides/users-guide/InputOutput.html.

[21] Rainer Bartholdus, Su Dong, et al. *ATLAS RCE Development Lab*. URL: https://twiki.cern.ch/twiki/bin/view/Atlas/RCEDevelopmentLab.

[22] Erwin Fehlberg. *Low-order classical Runge-Kutta formulas with stepsize control and their application to some heat transfer problems*. NASA Technical Report NASA-TR-R-315. http://hdl.handle.net/2060/19690021375. 1969.

[23] Tom Preston-Werner. *TOML. Tom's Obvious, Minimal Language*. URL: https://github.com/toml-lang/toml.

[24] Michael Kerrisk. *Linux Programmer's Manual. ld.so, ld-linux.so - dynamic linker/loader*. URL: http://man7.org/linux/man-pages/man8/ld.so.8.html.

[25] W. Shockley. “Currents to Conductors Induced by a Moving Point Charge”. In: *J. Appl. Phys.* 9.10 (1938), pp. 635–636. DOI: 10.1063/1.1710367.

[26] Simon Ramo. “Currents Induced by Electron Motion”. In: *Proc. IRE* 27.9 (Sept. 1939), pp. 584–585. ISSN: 0096-8390. DOI: 10.1109/JRPROC.1939.228757.

[27] W. Riegler and G. Aglieri Rinella. “Point charge potential and weighting field of a pixel or pad in a plane condenser”. In: *Nucl. Inst. Meth.* 767 (2014), pp. 267–270. ISSN: 0168-9002. DOI: 10.1016/j.nima.2014.08.044.

[28] R. Ballabriga et al. “Transient Monte Carlo simulations for the optimisation and characterisation of monolithic silicon sensors”. In: *Nucl. Instr. Meth. A* 1031 (2022), p. 166491. ISSN: 0168-9002. DOI: 10.1016/j.nima.2022.166491.

[29] Eric W. Weisstein. *Euler Angles. From MathWorld – A Wolfram Web Resource*. URL: http://mathworld.wolfram.com/EulerAngles.html.

[30] Amit Patel. *Hexagonal Grids*. URL: https://www.redblobgames.com/grids/hexagons/.

[31] John J. Smithrick and Ira T. Myers. “Average Triton Energy Deposited in Silicon per Electron-Hole Pair Produced”. In: *Phys. Rev. B* 1 (7 Apr. 1970), pp. 2945–2948. DOI: 10.1103/PhysRevB.1.2945.

[32] R. C. Alig, S. Bloom, and C. W. Struck. “Scattering by ionization and phonon emission in semiconductors”. In: *Phys. Rev. B* 22 (12 Dec. 1980), pp. 5565–5582. DOI: 10.1103/PhysRevB.22.5565.

[33] S. Croft and D.S. Bond. “A determination of the Fano factor for germanium at 77.4 K from measurements of the energy resolution of a 113 cm3 HPGe gamma-ray spectrometer taken over the energy range from 14 to 6129 keV”. In: *International Journal of Radiation Applications and Instrumentation. Part A. Applied Radiation and Isotopes* 42.11 (1991), pp. 1009–1014. ISSN: 0883-2889. DOI: https://doi.org/10.1016/0883-2889(91)90002-I.

[34] Alan Owens et al. “High resolution x-ray spectroscopy using GaAs arrays”. In: *Journal of Applied Physics* 90 (Nov. 2001), p. 5376. DOI: 10.1063/1.1406546.

[35] Padhraic Liam Mulligan. “Fabrication and Characterization of Gallium Nitride Schottky Diode Devices for Determination of Electron-hole Pair Creation Energy and Intrinsic Neutron Sensitivity”. PhD thesis. Ohio State University. Department of Mechanical Engineering, 2015.

[36] A.J. Dabrowski, J. Iwańczyk, and R. Triboulet. “n-type cadmium telluride surface-barrier nuclear radiation detectors”. In: *Nuclear Instruments and Methods* 118.2 (1974), pp. 531–535. ISSN: 0029-554X. DOI: 10.1016/0029-554X(74)90662-4.

[37] M. Sammartini et al. “A CdTe pixel detector–CMOS preamplifier for room temperature high sensitivity and energy resolution X and γ ray spectroscopic imaging”. In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 910 (2018), pp. 168–173. ISSN: 0168-9002. DOI: 10.1016/j.nima.2018.09.025.

[38] J. Tang, F. Kislat, and H. Krawczynski. “Cadmium Zinc Telluride detectors for a next-generation hard X-ray telescope”. In: *Astroparticle Physics* 128 (Mar. 2021), p. 102563. ISSN: 0927-6505. DOI: 10.1016/j.astropartphys.2021.102563.

[39] A. Niemela and H. Sipila. “Evaluation of CdZnTe detectors for soft X-Ray applications”. In: *IEEE Transactions on Nuclear Science* 41.4 (1994), pp. 1054–1057. DOI: 10.1109/23.322857.

[40] Takehiro Shimaoka et al. “Fano factor evaluation of diamond detectors for alpha particles”. In: *physica status solidi (a)* 213.10 (2016), pp. 2629–2633. DOI: https://doi.org/10.1002/pssa.201600195. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/pssa.201600195.

[41] B.F. Phlips et al. “Silicon carbide pin diodes as radiation detectors”. In: *IEEE Nuclear Science Symposium Conference Record, 2005.* Vol. 3. 2005, pp. 1236–1239. DOI: 10.1109/NSSMIC.2005.1596542.

[42] Giuseppe Bertuccio et al. “Advances in silicon carbide X-ray detectors”. In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 652.1 (2011). Symposium on Radiation Measurements and Applications (SORMA) XII 2010, pp. 193–196. ISSN: 0168-9002. DOI: https://doi.org/10.1016/j.nima.2010.08.046.

[43] C. Jacoboni et al. “A review of some charge transport properties of silicon”. In: *Solid State Electronics* 20 (Feb. 1977), pp. 77–89. DOI: 10.1016/0038-1101(77)90054-5.

[44] C. Canali et al. “Electron and hole drift velocity measurements in silicon and their empirical relation to electric field and temperature”. In: *IEEE Trans. Elec. Dev.* 22.11 (1975), pp. 1045–1047. DOI: 10.1109/T-ED.1975.18267.

[45] C. Scharf and R. Klanner. "Measurement of the drift velocities of electrons and holes in high-ohmic ⟨100⟩ silicon". In: *Nucl. Instr. Meth. A* 799 (2015), pp. 81–89. ISSN: 0168-9002. DOI: https://doi.org/10.1016/j.nima.2015.07.057.

[46] G. Masetti, M. Severi, and S. Solmi. "Modeling of carrier mobility against carrier concentration in arsenic-, phosphorus-, and boron-doped silicon". In: *IEEE Trans. Elec. Dev.* 30.7 (1983), pp. 764–769. DOI: 10.1109/T-ED.1983.21207.

[47] N.D. Arora, J.R. Hauser, and D.J. Roulston. "Electron and hole mobilities in silicon as a function of concentration and temperature". In: *IEEE Trans. Elec. Dev.* 29.2 (1982), pp. 292–295. DOI: 10.1109/T-ED.1982.20698.

[48] J. G. Ruch and G. S. Kino. "Transport Properties of GaAs". In: *Phys. Rev.* 174 (3 Oct. 1968), pp. 921–931. DOI: 10.1103/PhysRev.174.921.

[49] B. Bergmann et al. "Detector response and performance of a 500um thick GaAs attached to Timepix3 in relativistic particle beams". In: *Journal of Instrumentation* 15.03 (Mar. 2020), pp. C03013–C03013. DOI: 10.1088/1748-0221/15/03/c03013.

[50] R Quay et al. "A temperature dependent model for the saturation velocity in semiconductor materials". In: *Materials Science in Semiconductor Processing* 3.1 (2000), pp. 149–155. ISSN: 1369-8001. DOI: https://doi.org/10.1016/S1369-8001(00)00015-9.

[51] M.Ali Omar and Lino Reggiani. "Drift velocity and diffusivity of hot carriers in germanium: Model calculations". In: *Solid-State Electronics* 30.12 (1987), pp. 1351–1354. ISSN: 0038-1101. DOI: https://doi.org/10.1016/0038-1101(87)90063-3.

[52] O. Madelung, U. Rössler, and M. Schulz, eds. *Landolt-Börnstein - Group III Condensed Matter · Volume 41A1β: "Group IV Elements, IV-IV and III-V Compounds. Part b - Electronic, Transport, Optical and Other Properties"*. accessed 2022-03-23. DOI: https://doi.org/10.1007/b80447.

[53] Tigran T Mnatsakanov et al. "Carrier mobility model for GaN". In: *Solid-State Electronics* 47.1 (2003), pp. 111–115. ISSN: 0038-1101. DOI: https://doi.org/10.1016/S0038-1101(02)00256-3.

[54] The ROOT authors. *ROOT TFormula Class Reference*. 2013. URL: https://root.cern.ch/doc/master/classTFormula.html.

[55] W. Shockley and W. T. Read. "Statistics of the Recombinations of Holes and Electrons". In: *Phys. Rev.* 87 (5 Sept. 1952), pp. 835–842. DOI: 10.1103/PhysRev.87.835.

[56] R.N. Hall. "Germanium rectifier characteristics". In: *Phys. Rev.* 81.1 (1951), p. 228.

[57] J.G. Fossum and D.S. Lee. "A physical model for the dependence of carrier lifetime on doping density in nondegenerate silicon". In: *Solid-State Electronics* 25.8 (1982), pp. 741–747. ISSN: 0038-1101. DOI: https://doi.org/10.1016/0038-1101(82)90203-9.

[58] A. Schenk. "A model for the field and temperature dependence of Shockley-Read-Hall lifetimes in silicon". In: *Solid-State Electronics* 35.11 (1992), pp. 1585–1596. ISSN: 0038-1101. DOI: https://doi.org/10.1016/0038-1101(92)90184-E.

[59] Mark Kerr and Andres Cuevas. "General parametrization of Auger Recombination in crystalline silicon". In: *Journal of Applied Physics - J APPL PHYS* 91 (Feb. 2002). DOI: 10.1063/1.1432476.

[60] J. Dziewior and W. Schmid. “Auger coefficients for highly doped and highly excited silicon”. In: *Applied Physics Letters* 31.5 (1977), pp. 346–348. DOI: 10.1063/1.89694.

[61] Angela Vasilescu. *The NIEL scaling hypothesis applied to neutron spectra of irradiation facilities and in the ATLAS and CMS SCT*. ROSE Technical Note: ROSE/TN/97-2. 1997.

[62] G Kramberger et al. “Effective trapping time of electrons and holes in different silicon materials irradiated with neutrons, protons and pions”. In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 481.1 (2002), pp. 297–305. ISSN: 0168-9002. DOI: https://doi.org/10.1016/S0168-9002(01)01263-3.

[63] Olaf Krasel et al. “Measurement of trapping time constants in proton-irradiated silicon pad detectors”. In: *Nuclear Science, IEEE Transactions on* 1 (Jan. 2005), pp. 3055–3062. DOI: 10.1109/TNS.2004.839096.

[64] et al W. Adam. “Trapping in proton irradiated p+-n-n+ silicon sensors at fluences anticipated at the HL-LHC outer tracker”. In: *Journal of Instrumentation* 11.04 (Apr. 2016), P04023–P04023. DOI: 10.1088/1748-0221/11/04/p04023.

[65] I. Mandić et al. “Measurements with silicon detectors at extreme neutron fluences”. In: *Journal of Instrumentation* 15.11 (Nov. 2020), P11018–P11018. DOI: 10.1088/1748-0221/15/11/p11018.

[66] The Weightfield2 developers. *Weightfield2: a freeware 2D simulator for silicon and diamond detector*. URL: http://personalpages.to.infn.it/~cartigli/Weightfield2/index.html.

[67] 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* 53.9 (2006), pp. 2328–2334. ISSN: 1557-9646. DOI: 10.1109/TED.2006.881010.

[68] Esteban Curras Rivera and Michael Moll. “Study of impact ionization coefficients in silicon with Low Gain Avalanche Diodes”. In: (2022). arXiv: 2211.16543 [physics.ins-det].

[69] R. Van Overstraeten and H. De Man. “Measurement of the ionization rates in diffused silicon p-n junctions”. In: *Solid-State Electronics* 13.5 (1970), pp. 583–608. ISSN: 0038-1101. DOI: 10.1016/0038-1101(70)90139-5.

[70] Y. Okuto and C.R. Crowell. “Threshold energy effect on avalanche breakdown voltage in semiconductor junctions”. In: *Solid-State Electronics* 18.2 (1975), pp. 161–168. ISSN: 0038-1101. DOI: 10.1016/0038-1101(75)90099-4.

[71] M. Valdinoci et al. “Impact-ionization in silicon at large operating temperature”. In: *1999 International Conference on Simulation of Semiconductor Processes and Devices. SISPAD’99 (IEEE Cat. No.99TH8387)*. 1999, pp. 27–30. DOI: 10.1109/SISPAD.1999.799251.

[72] L. Garren et al. *Monte Carlo Particle Numbering Scheme*. 2015. URL: http://hepdata.cedar.ac.uk/lbl/2016/reviews/rpp2016-rev-monte-carlo-numbering.pdf.

[73] R. Kleczek, P. Grybos, and R. Szczygiel. "Charge sensitive amplifier for nanoseconds pulse processing time in CMOS 40 nm technology". In: *2015 22nd International Conference Mixed Design of Integrated Circuits Systems (MIXDES)*. 2015, pp. 292–296.

[74] "Index". In: *Tradeoffs and Optimization in Analog CMOS Design*. John Wiley & Sons, Ltd, 2008, pp. 583–594. ISBN: 9780470033715. DOI: 10.1002/9780470033715. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/9780470033715.

[75] Helga Holmestad. "Data analysis, simulations, and reconstruction of antiproton annihilations in a silicon pixel detector". PhD thesis. Department of Physics, University of Oslo, Oct. 2018.

[76] Chris Hagmann, David Lange, and Douglas Wright. "Cosmic-ray shower generator (CRY) for Monte Carlo transport codes". In: *2007 IEEE Nuclear Science Symposium Conference Record*. Vol. 2. 2007, pp. 1143–1146. DOI: 10.1109/NSSMIC.2007.4437209.

[77] Chris Hagmann, David Lange, and Doug Wright. *Monte Carlo Simulation of Proton-induced Cosmic-ray Cascades in the Atmosphere. UCRL-TM-229452*. URL: https://nuclear.llnl.gov/simulation/doc_cry_v1.7/cry_physics.pdf.

[78] Chris Hagmann et al. *Cosmic-ray Shower Library (CRY). UCRL-TM-229453*. URL: https://nuclear.llnl.gov/simulation/doc_cry_v1.7/cry.pdf.

[79] Geant4 Collaboration. *Geant4 Physics Lists*. URL: https://geant4-userdoc.web.cern.ch/UsersGuides/PhysicsListGuide/html/index.html.

[80] Geant4 Collaboration. *Geant4 GPS*. URL: http://geant4-userdoc.web.cern.ch/geant4-userdoc/UsersGuides/ForApplicationDeveloper/html/GettingStarted/generalParticleSource.html.

[81] Geant4 Collaboration. *Geant4 Particles*. URL: http://geant4-userdoc.web.cern.ch/geant4-userdoc/UsersGuides/ForApplicationDeveloper/html/TrackingAndPhysics/particle.html.

[82] S. Hauf et al. "Radioactive Decays in Geant4". In: *IEEE Transactions on Nuclear Science* 60.4 (Aug. 2013), pp. 2966–2983. ISSN: 0018-9499. DOI: 10.1109/TNS.2013.2270894.

[83] J. Apostolakis et al. "An implementation of ionisation energy loss in very thin absorbers for the GEANT4 simulation package". In: *Nucl. Instrum. Meth.* A453 (2000), pp. 597–605. DOI: 10.1016/S0168-9002(00)00457-5.

[84] C. Andreopoulos et al. "The GENIE Neutrino Monte Carlo Generator". In: *Nucl. Instrum. Meth. A* 614 (2010), pp. 87–104. DOI: 10.1016/j.nima.2009.12.009. arXiv: 0905.2517 [hep-ph].

[85] Andy Buckley et al. "The HepMC3 event record library for Monte Carlo event generators". In: *Comp. Phys. Comm.* 260 (2021), p. 107310. ISSN: 0010-4655. DOI: doi.org/10.1016/j.cpc.2020.107310.

[86] Martin A. Green and Mark J. Keevers. "Optical properties of intrinsic silicon at 300 K". In: *Progress in Photovoltaics: Research and Applications* 3.3 (1995), pp. 189–192. DOI: 10.1002/pip.4670030303.

[87] Morris Swartz. *A detailed simulation of the CMS pixel sensor*. Tech. rep. 2002.

[88] GDML Authors. *Geometry Description Markup Language (GDML)*. URL: https://gdml.web.cern.ch/GDML/.

[89] Geant4 Collaboration. *Geant4 Material Database*. URL: https://geant4-userdoc.web.cern.ch/UsersGuides/ForApplicationDeveloper/html/Appendix/materialNames.html.

[90] The EUTelescope Developers. *The EUTelescope Analysis Framework*. URL: https://eutelescope.github.io/.

[91] The Proteus Developers. *The Proteus Testbeam Reconstruction Framework*. URL: https://gitlab.cern.ch/proteus/proteus.

[92] Geant4 Collaboration. *Geant4 Visualization Drivers*. URL: https://geant4-userdoc.web.cern.ch/UsersGuides/ForApplicationDeveloper/html/Visualization/visdrivers.html.

[93] S. Spannagel et al. "Allpix^2: A modular simulation framework for silicon detectors". In: *Nucl. Instr. Meth. A* 901 (2018), pp. 164–172. ISSN: 0168-9002. DOI: 10.1016/j.nima.2018.06.020. arXiv: 1806.05813.

[94] *CMake for External Allpix Squared Modules Repository*. Sept. 2019. URL: https://gitlab.cern.ch/allpix-squared/external-modules/.

[95] GitLab Inc. *GitLab Flavored Markdown (GLFM)*. URL: https://docs.gitlab.com/ee/user/markdown.html.

[96] A. J. Peters and L. Janyst. "Exabyte Scale Storage at CERN". In: *Journal of Physics: Conference Series* 331.5 (2011), p. 052015. DOI: 10.1088/1742-6596/331/5/052015.

[97] John MacFarlane. *Pandoc. A universal document converter*. URL: https://pandoc.org/.

[98] Bjørn Erik Pedersen. *hugo. A Fast and Flexible Static Site Generator*. URL: https://gohugo.io/.

[99] *The Allpix Squared Website Repository*. URL: https://gitlab.cern.ch/allpix-squared/allpix-squared-website/.

[100] J. Behley, V. Steinhage, and A. B. Cremers. "Efficient radius neighbor search in three-dimensional point clouds". In: *2015 IEEE International Conference on Robotics and Automation (ICRA)*. May 2015, pp. 3625–3630. DOI: 10.1109/ICRA.2015.7139702.