Allpix^2 User Manual

Koen Wolters (koen.wolters@cern.ch)
Simon Spannagel (simon.spannagel@cern.ch)
Daniel Hynds (daniel.hynds@cern.ch)

July 6, 2022

Version v1.4.3

Contents

1 Introduction

Allpix^2 is a generic simulation framework for silicon tracker and vertex detectors written in modern C++, following the C++11 and C++14 standards. The goal of the Allpix^2 framework is to provide an easy-to-use package for simulating the performance of Silicon 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^2 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 Allpix^2 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^2 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^2. 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 very much appreciated. These should preferably be brought up on the issues tracker of the project which can be found in the repository [6].

1.3 Contributing Code

Allpix^2 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 Allpix^2 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.

The repository contains a few tools to facilitate contributions and to ensure code quality as detailed in Chapter 10.

2 Quick Start

This chapter serves as a swift introduction to Allpix^2 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 chapters instead.

Allpix^2 is a generic simulation framework for pixel detectors. It provides a modular, flexible and user-friendly structure for the simulation of independent detectors in arbitrary configurations. The framework currently relies on the Geant4 [1], ROOT [2] and Eigen3 [8] libraries which need to be installed and loaded before using Allpix^2.

The minimal, default installation can be obtained by executing the commands listed below. More detailed installation instructions can be found in Chapter 3.

```
$ 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 for later analysis.

After this quick start it is very much recommended to proceed to the other chapters of this user manual. For quickly resolving common issues, the Frequently Asked Questions in Chapter 11 may be particularly useful.

3 Installation

This section aims to provide details and instructions on how to build and install Allpix^2. 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^2. This chapter contains details on the standard installation process and information about custom build configurations.

3.1 Supported Operating Systems

Allpix^2 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), SLC 6 (with GCC and LLVM) and Mac OS Mojave (OS X 10.14, with AppleClang).

3.2 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 as described in Section 3.4.

The core framework is compiled separately from the individual modules and Allpix^2 has therefore only one required dependency: ROOT 6 (versions below 6 are not supported) [2]. Please refer to [9] for instructions on how to install ROOT. ROOT has several components of which the GenVector package is required to run Allpix^2. This package is included in the default build.

For some modules, additional dependencies exist. For details about the dependencies and their installation see the module documentation in Chapter 7. 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 [10] for details on how to install the software. All Geant4 data sets are required to run the modules successfully. It is recommended to enable the Geant4 Qt extensions to allow visualization of the detector setup and the simulated particle tracks. A useful set of CMake flags to build a functional Geant4 package would be:

```
 -DGEANT4_INSTALL_DATA=ON
-DGEANT4_USE_GDML=ON
-DGEANT4_USE_QT=ON
-DGEANT4_USE_XM=ON
-DGEANT4_USE_OPENGL_X11=ON
-DGEANT4_USE_SYSTEM_CLHEP=OFF
```

- Eigen3 [8]: Vector package used to perform Runge-Kutta integration in the generic charge propagation module. 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^2 installation without these dependencies. Details about these configuration options are given in Section 3.5.

3.3 Downloading the source code

The latest version of Allpix^2 can be downloaded from the CERN Gitlab repository [11]. 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
```

3.4 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 Geant):

```
$ 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
```

3.5 Configuration via CMake

Allpix^2 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_`*`ModuleName`*: If the specific module `ModuleName` should be installed or not. Defaults to ON for most modules, however some modules with large additional dependencies such as LCIO [12] are disabled by default. This set of parameters allows to configure the build for minimal requirements as detailed in Section 3.2.
- `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 ..
```

3.6 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 `build` folder. Running Allpix^2 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.

3.7 Docker images

Docker images are provided for the framework to allow anyone to run simulations without the need of installing Allpix^2 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^2 natively on the host system. Of course, any additional command line arguments known to the **`allpix`** executable described in Section 4.3 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:v1.1`**, and a full list of available Docker containers is provided via the project's container registry [13]. A short description of how Docker images for this project are built can be found in Section 10.5.

4 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^2 simulation, ROOT and (in most cases) Geant4 should be initialized. Refer to Section 3.4 for instructions on how to load these libraries.

4.1 Configuration Files

The framework is configured with simple human-readable configuration files. The configuration format is described in detail in Section 5.2.1, and 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 4.1.2, several advanced simulation chain configurations are presented in Chapter 8.

- The **detector** configuration passed to the framework to determine the geometry. Describes the detector setup, containing the position, orientation and model type of all detectors. An example is available in the repository at `examples/example_detector.conf`. Introduced in Section 4.1.3.

- 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 9.3 for more details about adding new models.

In the following paragraphs, the available types and the unit system are explained and an introduction to the different configuration files is given.

4.1.1 Parsing types and units

The Allpix^2 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 module documentation in Chapter 7 for the list of module parameters and their types. Parsing the value roughly follows common-sense (more details can be found in Section 5.2.2). 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 shown in Table 4.1.

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 (as given in Table 4.1) 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"
```

Table 4.1: List of units supported by Allpix[2]

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 (megaelectronvolt)	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	T (tesla)	mT (millitesla)
Angle	rad (radian)	deg (degree) mrad (milliradian)

```
# 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
```

4.1.2 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 4.2 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 5.3.1.

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 5.6.
- 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 4.1.3, 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
```

In the following paragraphs, a fully functional (albeit simple) configuration file with valid configuration is presented, as opposed to the above examples with hypothetical module names for illustrative purpose.

4.1.3 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 should contain all of the following parameters:

- A string referring to the **type** of the detector model. The model should exist in the search path described in Section 5.4.3.
- The 3D **position** in the world frame in the order x, y, z. See Section 5.4 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 X-Z-X extrinsic Euler angles, refer to Section 5.4 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 4.2. 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.

Furthermore it is possible to specify certain parameters of the detector explained in more detail in Section 5.4.3. 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 [14] models is the following:

```
# Placement of first detector, named "telescope1"
[telescope1]
# Type to the detector is the "timepix" model
type = "timepix"
```

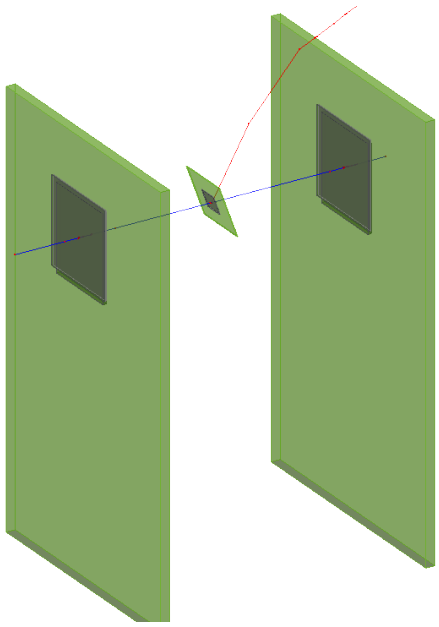

Figure 4.1: Visualization of a Pion passing through the telescope setup defined in the detector configuration file. A secondary particle is produced in the material of the detector in the center.

```
# 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
```

Figure 4.1 shows a visualization of the setup described in the file. This configuration is used

in the rest of this chapter for explaining concepts.

4.2 Framework parameters

The Allpix^2 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 4.1.3). 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).
- `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` and `DEBUG`, where all options are case-insensitive. Defaults to the `INFO` level. More details and information about the log levels, including how to change them for a particular module, can be found in Section 4.6. Can be overwritten by the `-v` parameter on the command line (see Section 4.3).
- `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 4.6.
- `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 4.3).
- `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 explictely, the value (**random_seed** + 1) is used.
- **library_directories**: Additional directories to search for module libraries, before searching the default paths. See Section 5.3.1 for details.
- **model_paths**: Additional files or directories from which detector models should be read besides the standard search locations. Refer to Section 5.4.3 for more information.
- **experimental_multithreading**: Enable **experimental** multi-threading for the framework. This can speed up simulations of multiple detectors significantly. More information about multi-threading can be found in Section 5.3.2.
- **workers**: Specify the number of workers to use in total, should be strictly larger than zero. Only used if **experimental_multithreading** is set to true. Defaults to the number of native threads available on the system if this can be determined, otherwise one thread is used.

4.3 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 4.2.
- `-v <level>`: Sets the global log verbosity level, overwriting the value specified in the configuration file described in Section 4.2. Possible values are `FATAL`, `STATUS`, `ERROR`, `WARNING`, `INFO` and `DEBUG`, where all options are case-insensitive. The module specific logging level introduced in Section 4.6 is not overwritten.
- `--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 5.2.1 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 5.4.3. 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 current simulated event is finished, while all other events requested in the configuration file are ignored. After finishing the event, the finalization stage is run for every module to ensure all modules finish properly. This signal can be very 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 recommmended 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.

4.4 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 these parameters, the respective module documentation in Chapter 7 should be consulted. A typical simulation in Allpix^2 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^2 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^2 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 in Section 4.1.3. 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^2 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_EMY
# 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
```

```
47 # Noise added by the digitisation
48 adc_smearing = 100e
49
50 # Save histograms to the ROOT output file
51 [DetectorHistogrammer]
52 # Save histograms for the "dut" detector only
53 name = "dut"
54
55 # Store all simulated objects to a ROOT file with TTrees
56 [ROOTObjectWriter]
57 # File name of the output file
58 file_name = "allpix-squared-output"
59 # Ignore initially deposited charges and propagated carriers:
60 exclude = DepositedCharge, PropagatedCharge
```

This configuration is available in the repository at `etc/manual.conf`. The detector configuration file from Section 4.1.3 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^2 is used (see the installation instructions in Chapter 3). 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 Chapter 5.

4.5 Extending the Simulation Chain

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

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 [15]. 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 transparency of the detector models (in percent)
transparency = 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 **`G4VRMLFILE_MAX_FILE_NUM`** should 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 module documentation in Chapter 7.

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 module description in Chapter 7.

```
# 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^2 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^2 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 12.2. An example electric field (with the file name used in the example below) can be found in the *etc* directory of the Allpix^2 repository.

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"
```

Magnetic Fields

For simulating the detector response in the presence of a magnetic field with Allpix^2, 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 Chapter 7 for more information on the available propagation modules.

Currently, only constant magnetic fields can be applied.

4.6 Logging and Verbosity Levels

Allpix^2 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 5.8). 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.

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 Sections 5.7.1 and 5.8.

4.7 Storing Output Data

Storing the simulation output to persistent storage is of primary importance for subsequent reprocessing and analysis. Allpix^2 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 [16]. 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 5.6. 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 [17], 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 6.2 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 12.3.

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^2 framework comes with a few more output modules which allow data storage in different formats, such as the LCIO persistency event data model [12], the native RCE file format [18], or the Corryvreckan reconstruction framework data format. Detailed descriptions of these modules can be found in Chapter 7.

5 Structure & Components of the Framework

This chapter details the technical implementation of the Allpix^2 framework and is mostly intended to provide insight into the gearbox to potential developers and interested users. The framework consists of the following four main components that together form Allpix^2:

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 7.

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 5.5.

4. **Tools**: Allpix^2 provides a set of header-only 'tools' that allow access to common logic shared by various modules. Examples are the Runge-Kutta solver [19] 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 12. 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 5.7.

Finally, Allpix^2 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 5.1 provides an overview of the architectural design of the core and describes its interaction with the rest of the Allpix^2 framework. The different subcomponents such as configuration, modules and messages are discussed in Sections 5.2–5.5. The chapter closes with a description of the available framework tools in Section 5.7. Some C++ code will be provided in the text, but readers not interested may skip the technical details.

5.1 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 [20] parser to read configuration files. It also contains the Allpix^2 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 5.2.

2. **Module**: The module subsystem contains the base class of all Allpix^2 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 5.3.

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 Section 5.4.

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 5.5.

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 5.7. 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 5.8.

5.2 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 4.1.

5.2.1 File format

Throughout the framework, a simplified version of TOML [20] 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.

5.2.2 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:

```
1  // Returns true if the key exists and false otherwise
2  config.has("key")
3  // Returns the number of keys found from the provided initializer list:
4  config.count({"key1", "key2", "key3"});
5  // Returns the value in the given type, throws an exception if not existing
   ↪  or a conversion to TYPE is not possible
6  config.get<TYPE>("key")
7  // Returns the value in the given type or the provided default value if it
   ↪  does not exist
8  config.get<TYPE>("key", default_value)
9  // Returns an array of elements of the given type
10 config.getArray<TYPE>("key")
11 // Returns a matrix: an array of arrays of elements of the given type
12 config.getMatrix<TYPE>("key")
13 // Returns an absolute (canonical if it should exist) path to a file
14 config.getPath("key", true /* check if path exists */)
15 // Return an array of absolute paths
16 config.getPathArray("key", false /* do not check if paths exists */)
17 // Returns the value as literal text including possible quotation marks
18 config.getText("key")
19 // Set the value of key to the default value if the key is not defined
20 config.setDefault("key", default_value)
21 // Set the value of the key to the default array if key is not defined
22 config.setDefaultArray<TYPE>("key", vector_of_default_values)
23 // Create an alias named new_key for the already existing old_key or throws
   ↪  an exception if the old_key does not exist
24 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 5.7.3. These conversions largely follow standard C++ 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.

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.

5.3 Modules and the Module Manager

Allpix^2 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. For more details about the instantiation logic for the different types of modules, see Section 5.3.1.

5.3.1 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 5.2 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 **libAllpix-ModuleModuleName**, 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.
4. 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.
5. The other standard locations to search for libraries depending on the operating system. Details about the procedure Linux follows can be found in [21].

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.

5.3.2 Parallel execution of modules

The framework has experimental support for running several modules in parallel. This feature is disabled for new modules by default, and has to be both supported by the module and enabled by the user as described in Section 4.2. A significant speed improvement can be achieved if the simulation contains multiple detectors or simulates the same module using different parameters.

The framework allows to parallelize the execution of the same type of module, if these would otherwise be executed directly after each other in a linear order. Thus, as long as the name of the module remains the same, while going through the execution order of all `run()` methods, all instances are added to a work queue. The instances are then distributed to a set of worker threads as specified in the configuration or determined from system parameters, which will execute the individual modules. The module manager will wait for all jobs to finish before continuing to process the next type of module.

To enable parallelization for a module, the following line of code has to be added to the constructor of a module:

```
1 // Enable parallelization of this module if multithreading is enabled
2 enable_parallelization();
```

By adding this, the module promises that it will work correctly if the run-method is executed multiple times in parallel, in separate instantiations. This means in particular that the module will safely handle access to shared (for example static) variables and it will properly bind

ROOT histograms to their directory before the `run()`-method. 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.

5.4 Geometry and Detectors

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^2 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 as defined in Section 5.4.1 (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 [22].

  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

```
1 orientation_mode = "zyx"
2 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.

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 optional parameter `alignment_precision_position` to specify the alignment precision along the three global axes as described in Section 4.1.3.
- An optional parameter `alignment_precision_orientation` for the alignment precision in the three rotation angles as described in Section 4.1.3.
- An optional **electric field** in the sensitive device. An electric field can be added to a detector by a special module as demonstrated in Section 4.5.

The detector configuration is provided in the detector configuration file as explained in Section 4.1.3.

5.4.1 Coordinate systems

Local coordinate systems for each detector and a global frame of reference 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 geometrical 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 provided in Figure 5.1. 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.

5.4.2 Changing and accessing the geometry

The geometry is needed at a very early stage because it determines the number of detector module instantiations as explained in Section 5.3.1. The procedure of finding and loading the appropriate detector models is explained in more detail in Section 5.4.3.

The geometry is directly added from the detector configuration file described in Section 4.1.3. The geometry manager parses this file on construction, and the detector models are loaded and

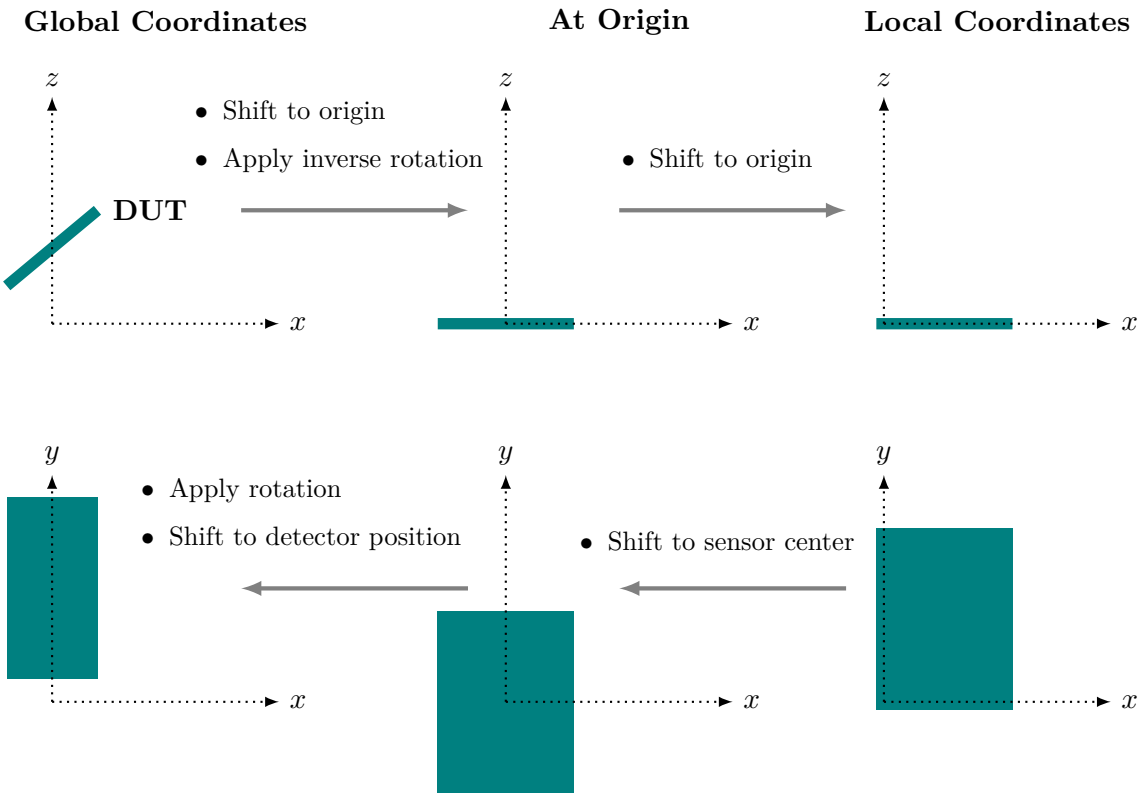

Figure 5.1: Coordinate transoformations from global to local and revers. The first row shows the detector positions in the repective coordinate systems in top view, the second row in side view.

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:

```
1 void run(unsigned int event_id) {
2     // Returns the linked detector
3     std::shared_ptr<Detector> detector = this->getDetector();
4 }
```

5.4.3 Detector models

Different types of detector models are available and distributed together with the framework: these models use the configuration format introduced in Section 5.2.1 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 more detailed geometries. Currently implemented derived models are the `MonolithicPixelDetectorModel`, which describes a monolithic detector with all electronics directly implemented in the same silicon wafer as the sensor, and the `HybridPixelDetectorModel`, 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.

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 9.3 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.

- **type**: A required parameter describing the type of the model. At the moment either **monolithic** or **hybrid**. This value determines 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.
- **implant_size**: The size of the collection diode implant in each pixel of the matrix. Provided as 2D parameter in the x-y-plane. This parameter is optional, the implant size defaults to the pixel pitch if not specified otherwise.
- **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** model. For this model, the **chip_thickness** parameter represents the first few micrometers of silicon which contain the chip circuitry and are shielded from the bias voltage and thus do not contribute to the signal formation.

The **hybrid** model 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** model are as follows:

- **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 the **bump_height** 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.

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 silicon layer). The support material can be chosen from a set of predefined materials, including PCB and Kapton.

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 silicon 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 parameter **location** 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_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^2 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 7 provides details about the materials supported by the geometry builder module (**GeometryBuilderGeant4**).

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
}
```

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 4.1.3. All model parameters, except the type parameter and the support layers, can be changed by adding a parameter with the exact 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.

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).

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 3.5).

3. The standard data paths on the system as given by the environmental variable **`$XDG_DATA_DIRS`** with "Allpix/models" appended. The **`$XDG_DATA_DIRS`** variable defaults to */usr/local/share/* (thus effectively */usr/local/share/Allpix/models*) followed by */usr/share/* (effectively */usr/share/Allpix/models*).

5.5 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^2 objects library are discussed in Chapter 6. The messaging system has a dispatching mechanism to send messages and a receiving part that fetches incoming messages.

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 exact 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(unsigned int event_id) {
    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
    messenger->dispatchMessage(this, message);
}
```

5.5.1 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 a variable. 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>) {
    messenger->bindSingle(this,
                          /* Pointer to the message variable */
                          &TestModule::message,
                          /* No special messenger flags */
                          MsgFlags::NONE);
}
std::shared_ptr<Message<Object>> message;
```

2. Bind a **set of messages** to a `std::vector` variable. 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) {
    messenger->bindMulti(this,
                          /* Pointer to the message vector */
                          &TestModule::messages,
                          /* No special messenger flags */
                          MsgFlags::NONE);
}
std::vector<std::shared_ptr<Message<Object>>> messages;
```

3. Listen to a particular message type and execute a **listener 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 listening function as this is supposed to take place in the module **run** method instead. Using a listener function can lead to unexpected behaviour 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 listener 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->registerListener(this,
                                /* Pointer to the listener method */
                                &TestModule::listener,
                                /* No special message flags */
```

```
6                                       MsgFlags::NONE);
7   }
8   void listener(std::shared_ptr<Message<Object>> message) {
9       // Do something with the received message ...
10  }
```

5.5.2 Message flags

Flags can be added to the bind and listening methods which enable a particular behaviour 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 behaviour. 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 `*`.

5.5.3 Persistency

As objects may contain information relating to other objects, in particular for storing their corresponding Monte Carlo history (see Section 6.2), objects are by default persistent until the end of each event. All messages are stored as shared pointers by the modules which send them, 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.

5.6 Redirect Module Inputs and Outputs

In the Allpix^2 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 digitisation 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"
```

5.7 Logging and other Utilities

The Allpix^2 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, introduced in Section 5.7.1 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, and which will be discussed in more detail in Section 5.7.2. The latter comprise tools which provide functionality the C++14 standard does not contain. These utilities are used internally in

the framework and are only shortly discussed in Section 5.7.3 (file system support) and Section 5.7.3 (string utilities).

5.7.1 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 very flexible and easy to read. The system is globally configured, thus only one logger instance exists. In order to send a message to the logging system at a level of **`LEVEL`**, the following can be used:

```
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 also be used: the logging system will automatically align every new line under the previous message and will leave the header space empty on new lines.

The system also allows messages to be updated on the same line, for simple progressbar-like functionality. It is enabled using the **`LOG_PROCESS(LEVEL, IDENTIFIER)`** command. 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 4.6.

5.7.2 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 very 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
```

```
9  Units::convert(1, "m/ns");
10 // Return the unit in the best type (lowest number larger than one) as
   ↪  string.
11 // The input is in default units 2000mm/ns and the 'best' output is 2m/ns
   ↪  (string)
12 Units::display(2e3, {"mm/ns", "m/ns"});
```

A description of the use of units in config files within Allpix^2 was presented in Section 4.1.1.

5.7.3 Internal utilities

The **filesystem** utilities provide functions to convert relative to absolute canonical paths, to iterate through all files in a directory and to create new directories. These functions should be replaced by the C++17 file system API [23] as soon as the framework minimum standard is updated to C++17.

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 Allpix^2 `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^2 tool system contains extensions to allow automatic conversions for ROOT and Geant4 types as explained in Section 12.1.1.

5.8 Error Reporting and Exceptions

Allpix^2 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 4.6.

The base exceptions in Allpix^2 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(unsigned int event_id) {
    // 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(unsigned int event_id) {
    // 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.

6 Objects

6.1 Object Types

Allpix^2 provides a set of objects which can be used to transfer data between modules. These objects can be sent with the messaging system as explained in Section 5.5. A `typedef` is added to every object in order to provide an alternative name for the message which is directly indicating the carried object.

The list of currently supported objects comprises:

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 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 is stored.

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. The MCParticle also stores an identifier of the particle type, using the PDG particle codes [24], 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.

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.

PropagatedCharge

The set of charge carriers propagated through the silicon 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.

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.

PixelHit

The digitised pixel hits after processing 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).

6.2 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 Chapter 5.5.3.

Object history is implemented using the ROOT TRef class [17], 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 [16]. 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.

7 Modules

This section describes all currently available Allpix^2 modules in detail. This includes a description of the physics implemented as well as possible configuration parameters along with their defaults. For inquiries about certain modules or its documentation, the respective maintainers should be contacted directly. The modules are listed in alphabetical order.

7.1 CapacitiveTransfer

Maintainer: Mateus Vicente (mvicente@cern.ch)
Status: Functional
Input: *PropagatedCharge*
Output: *PixelCharge*

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 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 transfered while neighbor pixels, with lower coupling capacitance, gets a fraction of the charged transfered 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 transfered) 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 transfered 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*.

Dependencies

This module requires an installation of Eigen3.

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 1 (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.

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 = 0
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]]
```

7.2 CorryvreckanWriter

Maintainer: Simon Spannagel (simon.spannagel@cern.ch), Daniel Hynds (daniel.hynds@cern.ch)
Status: Functional
Input: PixelHit

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.

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.

Usage

Typical usage is:

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

7.3 DefaultDigitizer

Maintainer: Simon Spannagel (simon.spannagel@cern.ch)
Status: Functional
Input: PixelCharge
Output: PixelHit

Description

Very 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 ADC with configurable resolution can be simulated.

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 multiplying the input charge with a defined gain factor. The actually applied gain is smeared with a Gaussian distribution on an event-by-event basis.
- 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.
- An ADC with configurable resolution, given in bit, can be simulated. For this, first an inaccuracy of the ADC is simulated using an additional Gaussian smearing which allows to take ADC noise into account. Then, the charge is converted into ADC units using the `adc_slope` and `adc_offset` parameters provided. Finally, the calculated value is clamped to be contained within the ADC resolution, over- and underflows are treated as saturation.

The ADC implementation also allows to simulate ToT (time-over-threshold) devices by setting the `adc_offset` parameter to the negative `threshold`. Then, the ADC only converts charge above threshold.

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 ADC units is provided if ADC simulation is enabled by setting `adc_resolution` to a value different from zero. In addition, the distribution of the actually applied threshold is provided as histogram.

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).
- `gain_smearing` : Standard deviation of the Gaussian uncertainty in the gain factor. Defaults to 0.
- `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.
- `adc_resolution` : Resolution of the ADC in units of bits. Thus, a value of 8 would translate to an ADC range of 0 – 255. A value of 0bit switches off the ADC simulation and returns the actual charge in electrons. Defaults to 0.
- `adc_smearing` : Standard deviation of the Gaussian noise in the ADC conversion (after applying the threshold). Defaults to 300 electrons.
- `adc_slope` : Slope of the ADC calibration in electrons per ADC unit (unit: “e”). Defaults to 10e.
- `adc_offset` : Offset of the ADC 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_adc`: Allows the ADC 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.
- `output_plots` : Enables output histograms to be 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 30ke.
- `output_plots_bins` : Set the number of bins for the output plot histograms, defaults to 100.

Usage

The default configuration is equal to the following:

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

7.4 DepositionGeant4

Maintainer: Koen Wolters (koen.wolters@cern.ch), Tobias Bisanz (tobias.bisanz@phys.uni-goettingen.de), Thomas Billoud (thomas.billoud@cern.ch)
Status: Functional
Output: DepositedCharge, MCParticle, MCTrack

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 is calculated using the mean pair creation energy `charge_creation_energy`, fluctuations are modeled using a Fano factor `fano_factor` assuming Gaussian statistics.

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 [25] 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 [25]. 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 [26] for information about the available types of particles and the PDG particle code definition [24] 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 [27] 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. 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`.

Ions can be used as particles by setting their atomic properties, i.e. the atomic number Z, the atomic mass A, their charge Q and the excitation energy E via the following syntax:

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

where `Z` and `A` are unsigned integers, `Q` is a signed integer and `E` a floating point value with units, e.g. `13eV`.

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 very thin sensors [28]. 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 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.

Dependencies

This module requires an installation Geant4.

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 [29].
- `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 silicon (3.64 eV, [30]).
- `fano_factor`: Fano factor to calculate fluctuations in the number of electron/hole pairs produced by a given energy deposition. Defaults to 0.115 [31].
- `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 [26] for information about the available types of particles.
- `particle_code` : PDG code of the Geant4 particle to use in the source.
- `source_energy` : Mean 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**.
- `file_name` : Name of the macro file (if source_type=**macro**).
- `number_of_particles` : Number of particles to generate in a single event. Defaults to one particle.
- `output_plots` : Enables output histograms to be 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.

Please note that the old source parameters from version v1.1.2 and before (`beam_energy`, `beam_energy_spread` and `beam_position`) are still supported but it is recommended to use the new corresponding ones.

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.

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"
source_energy = 10MeV
source_position = 0 0 -1mm
source_type = "beam"
beam_direction = 0 0 1
number_of_particles = 1
```

7.5 DepositionPointCharge

Maintainer: Simon Spannagel (simon.spannagel@cern.ch)
Status: Functional
Output: DepositedCharge, MCParticle

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 80 electron-hole pairs per micrometer. 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.

This module supports three different deposition models:

- In the `fixed` model, charge carriers are always deposited at the exact same position, specified via the `position` parameter, in every event of the simulation. This model ist 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.

Parameters

- `number_of_charges`: Number of charges to deposit inside the sensor. Defaults to 1. Only used for `point` source type.
- `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.
- `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. Defaults to `fixed`.
- `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.
- `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.

Usage

```
[DepositionPointCharge]
number_of_charges = 100
position = -10um 10um 0um
model = "fixed"
source_type = "mip"
```

7.6 DetectorHistogrammer

Maintainer: Koen Wolters (koen.wolters@cern.ch), Paul Schuetze (paul.schuetze@desy.de), Simon Spannagel (simon.spannagel@cern.ch)
Status: Functional
Input: PixelHit, MCParticle

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 convoled. For technical reasons, this offset is drawn randomly from a Gauss distribution independently for the resolution and the efficiency measurement.

- 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.
- 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 charge distribution.
- 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.

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.
- `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 `2um 2um`.
- `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.

Usage

This module is normally bound to a specific detector to plot, for example to the 'dut':

```
[DetectorHistogrammer]
name = "dut"
granularity = 100, 100
```

7.7 ElectricFieldReader

Maintainer: Koen Wolters (koen.wolters@cern.ch), Simon Spannagel (simon.spannagel@cern.ch)
Status: Functional

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 very 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). If the sensor is depleted from the implant side, 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 electric field is calculated as '$E(z) = \frac{U_{bias}-U_{depl}}{d} + 2\frac{U_{depl}}{d}\left(\frac{z}{d}\right)$'.
- For electric fields 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 [32]. 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 assumes the field represents a single pixel unit cell. If the field size and pixel pitch do not match, a warning is printed and the field is scaled to the pixel pitch.

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.

Furthermore the module can produce a plot the electric field profile on an projection axis normal to the x,y or z-axis at a particular plane in the sensor.

Parameters

- `model` : Type of the electric field model, either **linear**, **constant** or **mesh**.
- `bias_voltage` : Voltage over the whole sensor thickness. Used to calculate the electric field if the *model* parameter is equal to **constant** or **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**.
- `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.
- `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).
- `file_name` : Location of file containing the meshed electric field data. Only used if the *model* parameter has the value **mesh**.
- `field_scale` : Scale of the electric field in x- and y-direction. This parameter allows to use electric fields for fractions or multiple pixels. For example, an electric field calculated for a quarter pixel cell can be used by setting this parameter to `0.5 0.5` (half pitch in both directions) while a field calculated for four pixel cells in y and a single cell in x could be mapped to the pixel grid using `1 4`. Defaults to `1.0 1.0`. Only used if the *model* parameter has the value **mesh**.
- `field_offset`: Offset of the field from the pixel edge in x- and y-direction. By default, the framework assumes that the provided electric field starts at the edge of the pixel, i.e. with an offset of `0.0`. With this parameter, 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. In this case, a parameter of `0.5 0.5` should be used. The shift is applied in positive direction of the respective coordinate. Only used if the *model* parameter has the value **mesh**.
- `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).

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"
```

7.8 GDMLOutputWriter

Maintainer: Koen van den Brandt (kbrandt@nikhef.nl) **Status**: Functional

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.

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`

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`

Usage

Creating a GDML output file with the name myOutputfile.gdml

GDMLOutputWriter file_name = myOutputfile “‘

7.9 GenericPropagation

Maintainer: Koen Wolters (koen.wolters@cern.ch), Simon Spannagel (simon.spannagel@cern.ch)
Status: Functional
Input: DepositedCharge
Output: PropagatedCharge

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.

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 parameterization by C. Jacoboni et al. [33] 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 is calculated as

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

with v_m, E_c, β defined for electrons and holes separately as detailed in [33].

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 [19] 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 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 very time-consuming and should be switched off even when investigating drift behavior.

Dependencies

This module requires an installation of Eigen3.

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).
- `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.
- `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.

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 linegraphs 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_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_plots_lines_at_implants` : Determine whether to plot all charge carrier drift lines (`false`) or to just plot lines from charge carriers which reached the implant side within the allotted integration time (`true`). Defaults to `false`, i.e. all charge carrier drift lines are drawn.
- `output_animations` : In addition to the other output plots, also write a GIF animation of the charges drifting towards the electrodes. This is very 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.

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
```

7.10 GeometryBuilderGeant4

Maintainer: Koen Wolters (koen.wolters@cern.ch)
Status: Functional

Description

Constructs the Geant4 geometry from the internal geometry description. First constructs the world frame with a configurable margin and material. Then continues to create all the detectors using their internal detector models and to place them within the world frame.

All available detector models are fully supported. This builder can create extra support layers of the following materials:

- Air
- Aluminum
- Carbonfiber (a mixture of carbon and epoxy)
- Copper
- Epoxy
- G10 (PCB material)
- Kapton (using the `G4_KAPTON` definition)
- Lead
- Plexiglass (using the `G4_PLEXIGLASS` definition)
- Silicon
- Solder (a mixture of tin and lead)
- Tungsten

Dependencies

This module requires an installation Geant4.

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.

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
```

7.11 InducedTransfer

Maintainer: Simon Spannagel (simon.spannagel@cern.ch)
Status: Functional
Input: PropagatedCharge
Output: PixelCharge

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 [34] [35] 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 `induction_matrix` parameter.

Parameters

- `induction_matrix`: Size of the pixel sub-matrix for which the induced charge is calculated, provided as number of pixels in x and y. The numbers have to be odd and default to `3, 3`. Usually, a 3x3 grid (9 pixels) should suffice since the weighting potential at a distance of more than one pixel pitch normally is small enough to be neglected.

Usage

```
[InducedTransfer]
induction_matrix = 3 3
```

7.12 LCIOWriter

Maintainer: Andreas Nurnberg (andreas.nurnberg@cern.ch), Simon Spannagel (simon.spannagel@cern.ch), Tobias Bisanz(tobias.bisanz@phys.uni-goettingen.de)
Status: Functional
Input: PixelHit

Description

Writes pixel hit data to LCIO file, compatible with the EUTelescope analysis framework [36].

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.

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", "`
  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"]]`.

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"], ["telescope
```

7.13 MagneticFieldReader

Maintainer: Paul Schuetze (paul.schuetze@desy.de)
Status: Functional

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.

Parameters

- `model` : Type of the magnetic field model, currently only **constant** possible.
- `magnetic_field` : Vector describing the magnetic field.

Usage

An example is given below

```
[MagneticFieldReader]
model = "constant"
magnetic_field = 500mT 3.8T 0T
```

7.14 ProjectionPropagation

Maintainer: Simon Spannagel (simon.spannagel@cern.ch), Paul Schuetze (paul.schuetze@desy.de)
Status: Functional
Input: DepositedCharge
Output: PropagatedCharge

Description

The module projects the deposited electrons (or holes) to the sensor surface and applies a randomized diffusion. It can be used as a replacement for a charge propagation (e.g. the GenericPropagation module) for saving 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. The integral is calculated as follows, with '$\mu_0 = V_m/E_c$':

$$`t = \int \frac{1}{v} ds = \int \frac{1}{\mu(s)E(s)} ds = \int \frac{\left(1+\left(\frac{E(S)}{E_c}\right)^{\beta}\right)^{1/\beta}}{\mu_0 E(s)} ds`$$

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

$$`t = \frac{1}{\mu_0} \int \left(\frac{1}{E(s)} + \frac{1}{E_c}\right) ds = \frac{1}{\mu_0} \int \left(\frac{1}{ks+E_0} + \frac{1}{E_c}\right) ds = \frac{1}{\mu_0} \left[\frac{\ln(ks+E_0)}{k} + \frac{s}{E_c}\right]_a^b = \frac{1}{\mu_0} \left[\frac{\ln(E(s))}{k} + \frac{s}{E_c}\right]_a^b `.$$

Since the approximation of the drift time assumes a linear electric field, this module cannot be used with any other electric field configuration.

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.

Parameters

- `temperature`: Temperature in the sensitive device, used to estimate the diffusion constant and therefore the width of the diffusion distribution.
- `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.
- `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.
- `output_plots`: Determines if plots should be generated.

Usage

```
[ProjectionPropagation]
temperature = 293K
charge_per_step = 10
output_plots = 1
```

7.15 PulseTransfer

Maintainer: Simon Spannagel (simon.spannagel@cern.ch)
Status: Functional
Input: PropagatedCharge
Output: PixelCharge

Description

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. 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.

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.

Usage

The default configuration is equal to the following:

```
[PulseTransfer]
```

7.16 RCEWriter

Author: Salman Maqbool (salman.maqbool@cern.ch)
Maintainer: Moritz Kiehn (msmk@cern.ch)
Status: Functional
Input: PixelHit

Description

Reads in the PixelHit messages and saves them in the RCE format, appropriate for the Proteus telescope reconstruction software [37]. An event tree and a sensor tree and their branches are initialized in the module's `init()` 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.

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][37] 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][37] toml format. The file extension `.toml` will be appended if not present. Defaults to `geometry.toml`.

Usage

To create the default file an instantiation without arguments can be placed at the end of the main configuration:

```
[RCEWriter]
```

7.17 ROOTObjectReader

Maintainer: Koen Wolters (koen.wolters@cern.ch)
Status: Functional
Output: *all objects in input file*

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.

Parameters

- `file_name` : Location of the ROOT file containing the trees with the object data.
- `include` : Array of object names (without `allpix::` prefix) to be read from the ROOT trees, all other object names are ignored (cannot be used simulateneously with the *exclude* parameter).
- `exclude`: Array of object names (without `allpix::` prefix) not to be read from the ROOT trees (cannot be used simulateneously 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.

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"
```

7.18 ROOTObjectWriter

Maintainer: Koen Wolters (koen.wolters@cern.ch)
Status: Functional
Input: *all objects in simulation*

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.

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).

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 simulateneously with the *exclude* parameter).
- `exclude`: Array of object names (without `allpix::` prefix) that are not written to the ROOT trees (cannot be used together simulateneously with the *include* parameter).

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"
```

7.19 SimpleTransfer

Maintainer: Koen Wolters (koen.wolters@cern.ch)
Status: Functional
Input: PropagatedCharge
Output: PixelCharge

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.

When a collection diode size is specified for the respective detector via its `implant_size` parameter, the `collect_from_implant` option can be turned on in order to only pick charge carriers from the implant region and ignore everything outside this region. 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).

A histogram of charge carrier arrival times is generated if `output_plots` is enabled. The range and granularity of this plot can be configured.

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. 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`.

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
```

7.20 TextWriter

Maintainer: Simon Spannagel (simon.spannagel@cern.ch)
Status: Functional
Input: *all objects in simulation*

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"
```

7.20.1 TransientPropagation

Maintainer: Simon Spannagel (simon.spannagel@cern.ch)
Status: Immature
Input: DepositedCharge
Output: PixelCharge

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.

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 parameterization by C. Jacoboni et al. [33] and the magnetic field via a calculation of the Lorentz drift.

A fourth-order Runge-Kutta-Fehlberg method [19] 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 [34] [35] 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 pixel individually.

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.

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).
- `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.
- `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.
- `induction_matrix`: Size of the pixel sub-matrix for which the induced charge is calculated, provided as number of pixels in x and y. The numbers have to be odd and default to `3, 3`. 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, a 3x3 grid (9 pixels) should suffice since the weighting potential at a distance of more than one pixel pitch normally is small enough to be neglected while time simulation time is almost tripled.
- `ignore_magnetic_field`: The magnetic field, if present, is ignored for this module. Defaults to false.
- `output_plots` : Determines if simple output plots should be generated for a monitoring of the simulation flow. Disabled by default.

Usage

```
[TransientPropagation]
temperature = 293K
charge_per_step = 10
output_plots = true
timestep = 0.02ns
```

7.21 VisualizationGeant4

Maintainer: Koen Wolters (koen.wolters@cern.ch)
Status: Functional

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.

Dependencies

This module requires an installation of Geant4.

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 [38] 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.

- `transparency` : Default transparency 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 transparency.
- `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 [26] 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.

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
```

7.22 WeightingPotentialReader

Maintainer: Simon Spannagel (simon.spannagel@cern.ch)
Status: Functional

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.

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 [39]. 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 [39] 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 '$x_{1,2} = x \pm \frac{w_x}{2} \qquad y_{1,2} = y \pm \frac{w_y}{2}$'. 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.

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**.
- `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). Only used if `output_plots` is enabled.

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"
```

8 Examples

This section provides brief descriptions of the example configurations currently provided in the Allpix^2 repository. The examples are listed in alphabetical order.

8.1 CapacitiveTransfer example files

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* 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.

8.2 Fast Simulation Example

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.

8.3 Magnetic Field Example

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.

8.4 Precise DUT Simulation Example

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.

8.5 Example for Replaying a 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.

8.6 Source Measurement with Shielding

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.

8.7 TCAD Field Simulation Example

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.

Because the charge carrier propagation using the `GenericPropagation` module contributes the lion's share of the total simulation time, the simulation can profit from multi-threading, i.e. running the propagation for different detectors in parallel on different threads. An exemplary run on an Intel i7 machine with four cores sees a speedup of a factor two. It should be noted that currently multi-threading is still considered experimental.

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 final analysis.

9 Module & Detector Development

This chapter provides a few brief recipes for developing new simulation modules and detector models for the Allpix^2 framework. Before starting the development, the `CONTRIBUTING.md` file in the repository should be consulted for further information on the development process, code contributions and the preferred coding style for Allpix^2.

9.1 Coding and Naming Conventions

The code base of the Allpix^2 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.

9.1.1 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.

9.1.2 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

```
1 ./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 10.3.

9.2 Implementing a New Module

Owing to its modular structure, the functionality of the Allpix^2 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^2.

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 ti 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 in LaTeX has to be provided, as this Markdown-formatted file [40] is automatically converted and included in the user manual. Formulae can be included by enclosure in Dollar-backtick markers, i.e. ``$E(z) = 0$``. 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 5 for an introduction to the different parts of the framework.

- The module documentation in Chapter 7 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^2 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 [11].

9.2.1 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*:

```
1 ALLPIX_ENABLE_DEFAULT(OFF)
```

General guidelines and instructions for implementing new modules are provided in Section 9.2.

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(${MODULE_NAME} sources)` defines the module source files. Here, `sources` should be replaced by a list of all source files relevant to this module.
4. Possible lines to include additional directories and to link libraries for dependencies loaded earlier.
5. A line containing `ALLPIX_MODULE_INSTALL(${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)
IF(NOT Geant4_FOUND)
    MESSAGE(FATAL_ERROR "Could not find Geant4, make sure to source the Geant4 environment\n$ source YOUR_GEANT4_DIR/bin/geant4.sh")
ENDIF()

# 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})

# Link the Geant4 libraries to the module library
TARGET_LINK_LIBRARIES(${MODULE_NAME} ${Geant4_LIBRARIES})

# 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 Markdown format [40]. It is automatically converted to LaTeXusing Pandoc [41] and included in the user manual in Chapter 7. By documenting the module functionality in Markdown, the information is also viewable with a web browser in the repository within the module sub-folder.

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:

- The H1-size header with the name of the module and at least the following required elements: the **Maintainer** and the **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:

  ```
  # ModuleName
  Maintainer: Example Author (<example@example.org>)
  Status: Functional
  ```

In addition, the **Input** and **Output** objects to be received and dispatched by the module should be mentioned.

- An H3-size section named **Description**, containing a short description of the module.
- An H3-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 H3-size section with the title **Usage** which should contain at least one simple example of a valid configuration for the module.

`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.

9.2.2 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:

```
1 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:

```
1 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 5.5. 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:

- `init()`: Called after loading and constructing all modules and before starting the event loop. This method can for example be used to initialize histograms.
- `run(unsigned int event_number)`: Called for every event in the simulation, with the event number (starting from one). An exception should be thrown for serious errors, otherwise a warning should be logged.
- `finalize()`: Called 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.

9.3 Adding a New Detector Model

Custom detector models based on the detector classes provided with Allpix^2 can easily be added to the framework. In particular Section 5.4.3 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 5.4.3.
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 4.2), 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 `models` folder 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 [11].

10 Development Tools & Continuous Integration

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.

The chapter is structured as follows. Section 10.1 describes the available **make** targets for code quality and formatting checks, Section 10.3 briefly introduces the CI, and Section 10.6 provides an overview of the currently implemented framework, module, and performance test scenarios.

10.1 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 XML2.0 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.

10.2 Packaging

Allpix^2 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^2 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.

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>
```

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"
```

10.3 Continuous Integration

Quality and compatibility of the Allpix^2 framework is ensured by an elaborate continuous integration (CI) which builds and tests the software on all supported platforms. The Allpix^2 CI uses the GitLab Continuous Integration features and consists of seven distinct stages as depicted in Figure 10.1. It is configured via the `.gitlab-ci.yml` file in the repository's root

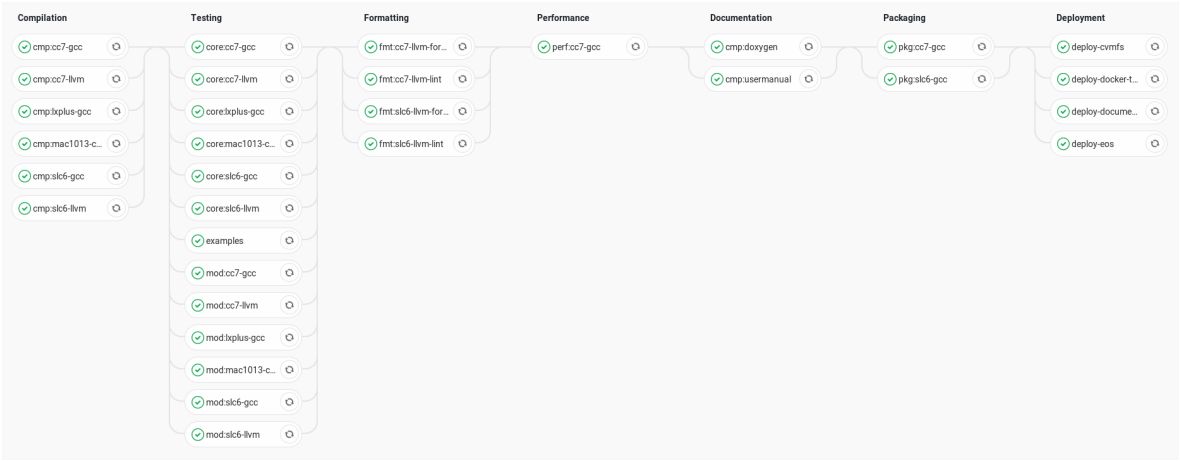

Figure 10.1: Typical Allpix^2 continous integration pipeline with 32 jobs distributed over seven distinct stages. In this example, all jobs passed.

directory, while additional setup scripts for the GitLab Ci Runner machines and the Docker instances can be found in the `.gitlab/ci` directory.

The **compilation** stage builds the framework from the source on different platforms. Currently, builds are performed on Scientific Linux 6, CentOS7, and Mac OS X. On Linux type platforms, the framework is compiled with recent versions of GCC and Clang, while the latest AppleClang is used on Mac OS X. 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 Section 10.6. 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.

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 as described in Section 10.6. 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 LaTeXdocuments or additional files which accidentally have not been committed to the repository.

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^2 project website through the EOS file system [42]. More detailed information on deployments is provided in the following.

10.4 Automatic Deployment

The CI is configured to automatically deploy new versions of Allpix^2 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.

10.4.1 Software deployment to CVMFS

The software is automatically deployed to CERN's VM file system (CVMFS) [43] 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 SLC 6 and CentOS 7 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*.

10.4.2 Documentation deployment to EOS

The project documentation is deployed to the project's EOS space at `/eos/project/a/allpix-squared/www/` for publication on the project website. This comprises both the PDF and HTML versions of the user manual (subdirectory `usermanual`) as well as the Doxygen code reference (subdirectory `reference/`). The documentation is only published only for new tagged versions of the framework.

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*.

10.4.3 Release tarball deployment to EOS

Binary release tarballs are deployed to EOS to serve as downloads from the website to the directory `/eos/project/a/allpix-squared/www/releases`. New tarballs are produced for every tag as well as for nightly builds of the `master` branch, which are deployed with the name `allpix-squared-latest-<system-tag>-opt.tar.gz`.

The files are taken from the packaging jobs and published via the `ci-web-deployer` Docker image from the CERN GitLab CI tools. 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*.

10.5 Building Docker images

New Allpix^2 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 [13]. 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 three consecutive images with different layers of software added. The 'base' image contains all build dependencies such as compilers, CMake, and git. It derives from a CentOS7 Docker image and can be build using the `etc/docker/Dockerfile.base` file via the following commands:

```
# Log into the CERN GitLab Docker registry:
$ docker login gitlab-registry.cern.ch
# Compile the new image version:
$ docker build --file etc/docker/Dockerfile.base            \
               --tag gitlab-registry.cern.ch/allpix-squared/\
                     allpix-squared/allpix-squared-base     \
               .
# Upload the image to the registry:
$ docker push gitlab-registry.cern.ch/allpix-squared/\
              allpix-squared/allpix-squared-base
```

The two main dependencies of the framework are ROOT6 and Geant4, which are added to the base image via the **deps** Docker image created from the file `etc/docker/Dockerfile.deps` via:

```
$ docker build --file etc/docker/Dockerfile.deps            \
               --tag gitlab-registry.cern.ch/allpix-squared/\
               allpix-squared/allpix-squared-deps           \
              .
$ docker push gitlab-registry.cern.ch/allpix-squared/\
              allpix-squared/allpix-squared-deps
```

These images are created manually and only updated when necessary, i.e. if major new version of the underlying dependencies are available.

The dependencies Docker images should not be flattend 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^2 image.

Finally, the latest revision of Allpix^2 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 3.7.

10.6 Tests

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. The tests can also be manually invoked from the build directory of Allpix^2 with

```
$ ctest
```

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
```

The configuration of the tests can be found in the `etc/unittests/test_*` directories of the repository and are automatically discovered by CMake. CMake automatically searches for Allpix^2 configuration files in the different directories and passes them to the Allpix^2 executable (cf. Section 4.3).

Adding a new test is as simple as adding a new configuration file to one of the different subdirectories and specifying the pass or fail conditions based on the tags described in the following paragraph.

Pass and Fail Conditions

The output of any test is compared to a search string in order to determine whether it passed or failed. These expressions are simply placed in the configuration file of the corresponding tests, a tag at the beginning of the line indicates whether it should be used for passing or failing the test. Each test can only contain one passing and one failing expression. If different functionality and thus outputs need to be tested, a second test should be added to cover the corresponding expression.

Different tags are provided for Mac OS X since the C++ standard does not define the exact implementation of random number distributions such as `std::normal_distribution`. Thus, the distributions produce different results on different platforms even when used with the same random number as input.

Passing a test
: The expression marked with the tag `#PASS`/`#PASSOSX` 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`/`#FAILOSX` 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, the module test 09 described below implements such a dependency as it uses the output of module test 08-1 to read data from a previously produced Allpix^2 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 4.3. 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.

Framework Functionality Tests

The framework functionality tests aim at reproducing basic features such as correct parsing of configuration keys or resolution of module instantiations. Currently implemented tests comprise:

`test_01-1_globalconfig_detectors.conf`
: tests the framework behavior in case of a non-existent detector setup description file.

`test_01-2_globalconfig_modelpaths.conf`
: tests the correct parsing of additional model paths and the loading of the detector model.

`test_01-3_globalconfig_log_format.conf`
: switches the logging format.

`test_01-4_globalconfig_log_level.conf`
: sets a different logging verbosity level.

`test_01-5_globalconfig_log_file.conf`
: configures the framework to write log messages into a file.

`test_01-6_globalconfig_missing_model.conf`
: tests the behavior of the framework in case of a missing detector model file.

`test_01-7_globalconfig_random_seed.conf`
: sets a defined random seed to start the simulation with.

`test_01-8_globalconfig_random_seed_core.conf`
: sets a defined seed for the core component seed generator, e.g. used for misalignment.

`test_02-1_specialization_unique_name.conf`
: tests the framework behavior for an invalid module configuration: attempt to specialize a unique module for one detector instance.

`test_02-2_specialization_unique_type.conf`
: tests the framework behavior for an invalid module configuration: attempt to specialize a unique module for one detector type.

`test_03-1_geometry_g4_coordinate_system.conf`
: ensures that the Allpix^2 and Geant4 coordinate systems and transformations are identical.

`test_03-2_geometry_rotations.conf`
: tests the correct interpretation of rotation angles in the detector setup file.

`test_03-3_geometry_misaligned.conf`
: tests the correct calculation of misalignments from alignment precisions given in the detector setup file.

`test_03-4_geometry_overwrite.conf`
: checks that detector model parameters are overwritten correctly as described in Section 5.4.3.

`test_04-1_configuration_cli_change.conf`
: tests whether single configuration values can be overwritten by options supplied via the command line.

`test_04-2_configuration_cli_nochange.conf`
: tests whether command line options are correctly assigned to module instances and do not alter other values.

`test_05-1_overwrite_same_denied.conf`
: tests whether two modules writing to the same file is disallowed if overwriting is denied.

`test_04-2_configuration_cli_nochange.conf`
: tests whether two modules writing to the same file is allowed if the last one reenables overwriting locally.

Module Functionality Tests

These tests ensure the proper functionality of each module covered 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] allows to reproduce the same simulation event.

One 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.

Currently implemented tests comprise:

`test_01_geobuilder.conf`
: 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.

`test_02-1_electricfield_linear.conf`
: 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.

test_02-2_electricfield_init.conf
loads an INIT file containing a TCAD-simulated electric field (cf. Section 4.5) 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.

test_02-3_electricfield_linear_depth.conf
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.

test_02-4_magneticfield_constant.conf
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.

test_03-1_deposition.conf
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.

test_03-2_deposition_mc.conf
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.

test_03-3_deposition_track.conf
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.

test_03-4_deposition_source_point.conf
tests the point source in the charge carrier deposition module by monitoring the deposited charges.

test_03-5_deposition_source_square.conf
tests the square source in the charge carrier deposition module by monitoring the deposited charges.

test_03-6_deposition_source_sphere.conf
tests the sphere source in the charge carrier deposition module by monitoring the deposited charges.

test_03-7_deposition_source_macro.conf
tests the G4 macro source in the charge carrier deposition module using the macro file `source_macro_test.txt`, monitoring the deposited charges.

test_03-8_deposition_point.conf
tests the deposition of a point charge at a specified position, checks the position of the deposited charge carrier in global coordinates.

test_03-9_deposition_scan.conf
tests the scan of a pixel volume by depositing charges for a given number of events, check for the calculated voxel size.

test_03-10_deposition_scan_cube.conf
tests the calculation of the scanning points by monitoring the warning of the number of events is not a perfect cube.

test_03-11_deposition_mip.conf
tests the deposition of charges along a line by monitoring the calculated step size and number of charge carriers deposited per step.

test_03-12_deposition_mip_position.conf
tests the generation of the Monte Carlo particle when depositing charges along a line by monitoring the start and end positions of the particle.

test_03-13_deposition_fano.conf
tests the simulation of fluctuations in charge carrier generation by monitoring the total number of generated carrier pairs when altering the Fano factor.

test_03-14_deposition_spot.conf
tests the deposition of charge carriers around a fixed position with a Gaussian distribution.

test_04-1_propagation_project.conf
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.

test_04-2_propagation_generic.conf
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.

test_04-3_propagation_generic-magnetic.conf
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.

test_04-4_propagation_project_integration.conf
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.

test_05_transfer_simple.conf
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.

test_06-1_digitization_charge.conf
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.

test_06-2_digitization_adc.conf
digitizes the transferred charges and tests the conversion into ADC units. The monitored output comprises the converted charge value in units of ADC counts.

test_06-3_digitization_gain.conf
digitizes the transferred charges and tests the amplification process by monitoring the total charge after signal amplification and smearing.

test_07_histogramming.conf
tests the detector histogramming module and its clustering algorithm. The monitored output comprises the total number of clusters and their mean position.

test_08-1_writer_root.conf
ensures proper functionality of the ROOT file writer module. It monitors the total number of objects and branches written to the output ROOT trees.

test_08-2_writer_rce.conf
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.

test_08-3_writer_lcio.conf
ensures proper functionality of the LCIO file writer module. Similar to the above test, the correct conversion of PixelHits (coordinates and charge) is monitored.

test_08-4_writer_corryvreckan.conf
ensures proper functionality of the Corryvreckan file writer module. The monitored output comprises the coordinates of the pixel produced in the simulation.

test_08-5_writer_corryvreckan_mc.conf
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.

test_08-6_writer_text.conf
ensures proper functionality of the ASCII text writer module by monitoring the total number of objects and messages written to the text file..

test_08-7_writer_lcio_detector_assignment.conf
exercises the assignment of detector IDs to Allpix^2 detectors in the LCIO output file. A fixed ID and collection name is assigned to the simulated detector.

test_08-8_writer_lcio_no_mc_truth.conf
ensures that simulation results are properly converted to LCIO and stored even without the Monte Carlo truth information available.

test_09-1_reader_root.conf
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.

`test_09-2_reader_root_seed.conf`
: 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.

`test_09-3_reader_root_ignoreseed.conf`
: 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.

Performance Tests

Similar to the module test implementation described above, performance tests use configurations prepared such, that one particular module takes most of the load (dubbed the "slowest instantiation" by Allpix^2), 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.

Current performance tests comprise:

`test_01_deposition.conf`
: tests the performance of charge carrier deposition in the sensitive sensor volume using Geant4 [1]. A stepping length of 1.0 µm is chosen, and 10 000 events are simulated. The addition of an electric field and the subsequent projection of the charges are necessary since Allpix^2 would otherwise detect that there are no recipients for the deposited charge carriers and skip the deposition entirely.

`test_02-1_propagation_generic.conf`
: 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.

`test_02-2_propagation_project.conf`
: 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.

`test_02-3_propagation_generic_multithread.conf`
: tests the performance of multi-threaded simulation. It utilizes the very same configuration as performance test 02-1 but in addition enables multi-threading with four worker threads.

11 Frequently Asked Questions

This chapter provides answers to some of the most frequently asked questions concerning usage, configuration and extension of the Allpix^2 framework.

11.1 Installation & Usage

What is the easiest way to use Allpix^2 on CERN's LXPLUS?

Central installations of Allpix^2 on LXPLUS are provided via CVMFS for both supported LXPLUS operating systems, SLC6 and CERN CentOS7. Please refer to Section 10.4.1 for the details of how to access these installations.

What is the quickest way to get a local installation of Allpix^2?

The project provides ready-to-use Docker containers which contain all dependencies such as Geant4 and ROOT. Please refer to Section 3.7 for more information on how to start and use these containers.

11.2 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 5.3.1 for more information.

How can I run the exact 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 5.5 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 3.5.

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^2 can parse. More information about this tool can be found in Section 12.2, instructions to import the generated field are provided in Section 4.5.

11.3 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 5.4.3 for more information.

11.4 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^2 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 6.2 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 6.2 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.

How do I access data stored in a file produced with the ROOTObjectWriter from an analysis script?

Allpix^2 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 C++ 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 and in Section 12.3 of this user manual. Scripts written in both C++ and in Python are provided.

11.5 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 **`listener`** function for the module as described in Section 5.5. 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 **`listener`** routine, but deferred to the **`run`** function of the respective output module.

How do I process data from multiple detectors?

When developing a new Allpix^2 module which processes data from multiple detectors,

e.g. as the simulation of a track trigger module, this module has to be of type *unqiue* as described in Section 5.3. 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 5.5. For *PixelHit* messages, an example code would be:

```
TrackTriggerModule(Configuration&, Messenger* messenger,
 ↪  GeometryManager* geo_manager) {
    messenger->bindMulti(this,
                         &TrackTriggerModule::messages,
                         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^2, 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 5.5.2 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(this, &MyModule::pixels_message_);
    messenger->bindSingle(this, &MyModule::mcparticle_message_,
 ↪  MsgFlags::REQUIRED);
}
```

11.6 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 very 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 Figure 11.1 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.

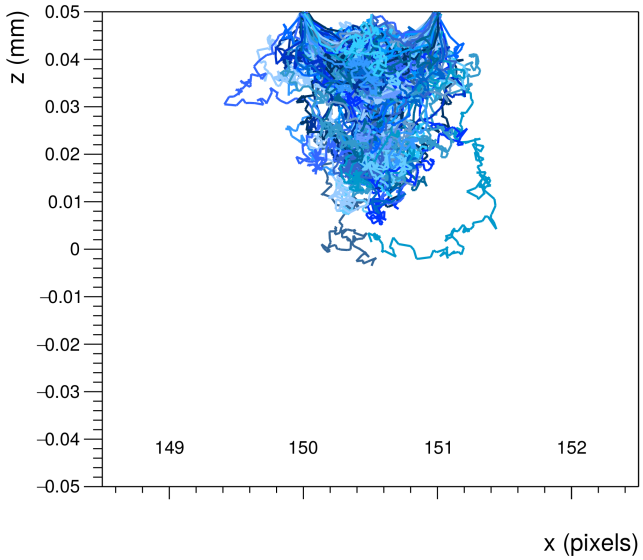

Figure 11.1: Drift and diffusion visualization of charge carrier groups being transported through a high-resistivity CMOS silicon sensor. The plot shows the situation after an integration time of 20 ns, only charge carrier groups which reached the implant side of the sensor are drawn.

12 Additional Tools & Resources

This chapter briefly describes tools provided with the Allpix^2 framework, which might be re-used in new modules or in standalone code.

12.1 Framework Tools

The following tools are part of the Allpix^2 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.

12.1.1 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 5.7.3) 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.

12.1.2 Runge-Kutta integrator

A fast Eigen-powered [8] Runge-Kutta integrator is provided as a tool to numerically solve differential equations [19]. 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.

```
1 // Define lambda functions to compute the charge carrier velocity at each
  ↪ step
2 std::function<Eigen::Vector3d(double, Eigen::Vector3d)> carrier_velocity =
3     [&](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.

12.1.3 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_.get_by_file_name(canonical_path,
↪ "V/cm");
}
```

For the INIT format, the **get_by_file_name()** 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 4.1.1. 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.

12.2 TCAD DF-ISE mesh converter

This code takes the `.grd` and `.dat` files of the DF-ISE format from TCAD simulations as input. 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.

A new regular mesh is created by scanning the model volume in regular X Y and Z steps (not necessarily coinciding with original mesh nodes) 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 software uses the Octree implementation [44].

The output `.init` or `.apf` file can be imported into Allpix Squared. The INIT file is an ASCII text file with a header followed by a list of columns organized as

```
node.x node.y node.z observable.x observable.y observable.z
```

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.

Compilation

When compiling the Allpix Squared framework, the TCAD DF-ISE 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 TCAD DF-ISE 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.

Features

- TCAD DF-ISE file format reader.
- 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).
- `dimension`: Specify mesh dimensionality (defaults to 3).
- `region`: Region name or list of region names to be meshed (defaults to `bulk`).
- `observable`: Observable to be interpolated (defaults to `ElectricField`).
- `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`).
- `max_radius`: Maximum search radius (default is `10um`).
- `volume_cut`: Minimum volume for tetrahedron for non-coplanar vertices (defaults to minimum double value).
- `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.
- `mesh_tree`: Boolean to enable creation of a root file with the TCAD mesh nodes stored in a `ROOT::TTree`. This setting is deactivated by default.
- `workers`: Number of worker threads to be used for the interpolation. Defaults to the available number of cores on the machine (hardware concurrency).

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.

In addition, the `mesh_plotter` tool can be used, in order 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)
```

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*. The data to be plotted can be selected with the `-d` option, the arguments are *ex*, *ey*, *ez* for the vector components or the default value n for the norm of the electric field. 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.

Octree

J. Behley, V. Steinhage, A.B. Cremers. *Efficient Radius Neighbor Search in Three-dimensional Point Clouds*, Proc. of the IEEE International Conference on Robotics and Automation (ICRA), 2015 [44].

12.3 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.

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()`

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")`

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.

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: * run `python display_mc_hits.py -l path/to/libAllpixObjects.so -f path/to/data.root -d <detector_name>`

Acknowledgments

Allpix^2 has been developed and is maintained by

- Koen Wolters, CERN
- Daniel Hynds, CERN
- Simon Spannagel, CERN

The following authors, in alphabetical order, have contributed to Allpix^2:

- Thomas Billoud, Université de Montréal
- Tobias Bisanz, Georg-August-Universität Göttingen
- Liejian Chen, Institute of High Energy Physics Beijing
- Katharina Dort, CERN Summer Student
- Neal Gauvin, Université de Genève
- Maoqiang Jing, University of South China, Institute of High Energy Physics Beijing
- Moritz Kiehn, Université de Genève
- Salman Maqbool, CERN Summer Student
- Andreas Matthias Nürnberg, CERN
- Marko Petric, CERN
- Edoardo Rossi, DESY
- Andre Sailer, CERN
- Paul Schütze, DESY
- Xin Shi, Institute of High Energy Physics Beijing
- Ondrej Theiner, Charles University
- Mateus Vicente Barreto Pinto, Université de Genève

The authors would also like to express their thanks to the developers of AllPix [3, 4].

Bibliography

[1] S. Agostinelli et al. "Geant4 - a simulation toolkit". In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 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. 21, 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 Allpix² Code Documentation*. Aug. 22, 2017. URL: https://cern.ch/allpix-squared/reference/.

[6] *The Allpix² Project Issue Tracker*. July 27, 2017. URL: https://gitlab.cern.ch/allpix-squared/allpix-squared/issues.

[7] *The Allpix² Project Forum*. Dec. 12, 2018. URL: https://cern.ch/allpix-squared-forum.

[8] Gaël Guennebaud, Benoît Jacob, et al. *Eigen v3*. 2010. URL: http://eigen.tuxfamily.org.

[9] Rene Brun and Fons Rademakers. *Building ROOT*. URL: https://root.cern.ch/building-root.

[10] 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/.

[11] *The Allpix² Project Repository*. Aug. 2, 2017. URL: https://gitlab.cern.ch/allpix-squared/allpix-squared/.

[12] 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.

[13] Simon Spannagel. *The Allpix² Docker Container Registry*. Mar. 12, 2018. URL: https://gitlab.cern.ch/allpix-squared/allpix-squared/container_registry.

[14] X. Llopart et al. "Timepix, a 65k programmable pixel readout chip for arrival time, energy and/or photon counting measurements". In: *Nucl. Inst. Meth.* 581.1 (2007). VCI 2007, pp. 485–494. ISSN: 0168-9002. DOI: 10.1016/j.nima.2007.08.079.

[15] Geant4 Collaboration. *Geant4 User's Guide for Application Developers. Visualization*. 2016. URL: https://geant4.web.cern.ch/geant4/UserDocumentation/UsersGuides/ForApplicationDeveloper/html/ch08.html.

[16] Rene Brun and Fons Rademakers. *ROOT User's Guide. Trees.* URL: https://root.cern.ch/root/htmldoc/guides/users-guide/Trees.html.

[17] Rene Brun and Fons Rademakers. *ROOT User's Guide. Input/Output.* URL: https://root.cern.ch/root/htmldoc/guides/users-guide/InputOutput.html.

[18] Rainer Bartholdus, Su Dong, et al. *ATLAS RCE Development Lab.* URL: https://twiki.cern.ch/twiki/bin/view/Atlas/RCEDevelopmentLab.

[19] 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.

[20] Tom Preston-Werner. *TOML. Tom's Obvious, Minimal Language.* URL: https://github.com/toml-lang/toml.

[21] 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.

[22] Eric W. Weisstein. *Euler Angles. From MathWorld – A Wolfram Web Resource.* URL: http://mathworld.wolfram.com/EulerAngles.html.

[23] Beman Dawes. *Adopt the File System TS for C++17.* Feb. 2016. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0218r0.html.

[24] 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.

[25] Geant4 Collaboration. *Geant4 GPS.* URL: http://geant4-userdoc.web.cern.ch/geant4-userdoc/UsersGuides/ForApplicationDeveloper/html/GettingStarted/generalParticleSource.html.

[26] Geant4 Collaboration. *Geant4 Particles.* URL: http://geant4-userdoc.web.cern.ch/geant4-userdoc/UsersGuides/ForApplicationDeveloper/html/TrackingAndPhysics/particle.html.

[27] 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.

[28] 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.

[29] Geant4 Collaboration. *Geant4 Physics Lists.* URL: http://geant4.cern.ch/support/proc_mod_catalog/physics_lists/referencePL.shtml.

[30] 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.

[31] 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.

[32] Morris Swartz. *A detailed simulation of the CMS pixel sensor.* Tech. rep. 2002.

[33] 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.

[34] 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.

[35] 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.

[36] The EUTelescope Developers. *The EUTelescope Analysis Framework*. URL: http://eutelescope.web.cern.ch/.

[37] The Proteus Developers. *The Proteus Testbeam Reconstruction Framework*. URL: https://gitlab.cern.ch/unige-fei4tel/proteus/.

[38] Geant4 Collaboration. *Geant4 Visualization Drivers*. URL: https://geant4.web.cern.ch/geant4/UserDocumentation/UsersGuides/ForApplicationDeveloper/html/ch08s03.html.

[39] 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.

[40] John Gruber and Aaron Swartz. *Markdown*. URL: https://daringfireball.net/projects/markdown/.

[41] John MacFarlane. *Pandoc. A universal document converter*. URL: http://pandoc.org/.

[42] 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.

[43] 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.

[44] 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.