CLICdp-Note-2017-006
03 December 2017

User Manual for the Allpix^2 Simulation Framework

K. Wolters[1)*], S. Spannagel[*], D. Hynds[*]

[*] *CERN, Switzerland*

Abstract

Several simulation tools exist for the detailed study of position sensitive silicon detectors, covering aspects ranging from the electrical properties of sensors to the behaviour of charged particles traversing a given detector setup. Each of these toolkits performs a very specialised task, and for the complete description of a silicon detector several such software packages must typically be used. Allpix^2 builds upon this work by providing a complete and easy-to-use C++ software package for simulating detector performance, from the interaction of particles to the digitisation of propagated carriers by the front-end electronics. A modular framework is used to flexibly add or remove modules from the simulation chain, each performing specific tasks such as interfacing to Geant4 to deposit energy in the detector and provide an accurate description of material effects, or the propagation of deposited charges through the sensor bulk. This document presents the user manual of the software as of release version 1.0.2.

This work was carried out in the framework of the CLICdp Collaboration

[1] koen.wolters@cern.ch

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 ionising radiation through the sensor and finishing with the digitisation 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 digitisation of hits in the frontend electronics.

This document presents the user manual of the software as of version 1.0.2. The software will be maintained and extended in the future, and new releases along with their respective user manuals will be announced on the project website [3].

Allpix^2 is designed as a modular framework, allowing for an easy extension to more complex and specialised detector simulations. The modular setup allows the separation of the framework core from the implementation of the algorithms in 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 [4, 5] 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 [6] 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 [7].

1.3. Contributing Code

Allpix^2 is a community project that benefits from active participation in the development and code contributions from users. We encourage users to discuss their needs either via the issue tracker of the repository [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.

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 sections 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 shell commands listed below. More detailed installation instructions can be found in Section 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 etc/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 executes the whole simulation 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 digitised 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 sections of this user manual. For quickly resolving common issues, the Frequently Asked Questions in Section 9 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 Sierra (OS X 10.12, 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 Section 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_BUILD_MULTITHREADED=OFF
  -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 behaviour.

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 (**bin/allpix -c etc/example.conf**) should pass the test without problems if a standard installation is used.

3.7. Testing

The build system of the framework is configured to provide a set of automated tests which can be executed to ensure the correct compilation and functionality of the framework.

The tests use the example configuration shipped with the source code and can be started from the build directory of Allpix^2 by invoking:

```
$ make test
```

All tests are expected to pass.

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 has 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 `etc/example.conf`. More details and a more thorough example are found in Section 4.1.2.
- 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 `etc/example_detector.conf`. Introduced in Section 4.1.3.
- The detector **models** 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 8.2 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 Section 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.

Table 1: List of units supported by Allpix[2]

Quantity	Default unit	Auxiliary units
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)	C (coulomb)
Voltage	MV (megavolt)	V (volt) kV (kilovolt)
Angle	rad (radian)	deg (degree) mrad (milliradian)

- If the value is an **arithmetic** type, it may have a suffix indicating the unit. The list of base units is shown in Table 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 $\star$ 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 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"
# Integer and floats can be specified in standard formats
int_value = 42
float_value = 123.456e9
# Units can be passed to arithmetic type
energy_value = 1.23MeV
time_value = 42ns
# Units are combined in linear order
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
# Path below will be /home/user/test if the config file is in
 ↪  /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]` 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 5.3.3.

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
#   this 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.

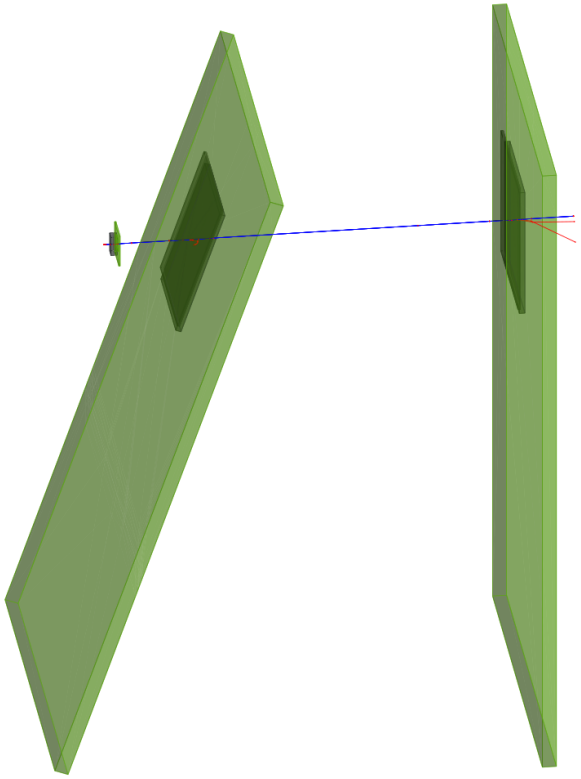

Figure 1: Visualisation of a particle passing through the telescope setup defined in the detector configuration file

- 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 an X-Z-X extrinsic Euler angles, refer to Section 5.4 for details.

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 of one test detector and two Timepix [13] models is:

```
# name the first detector `telescope1`
[telescope1]
# set the type to the "test" detector model
type = "test"
# place it at the origin of the world frame
position = 0 0 0mm
# use the default orientation
orientation = 0 0 0

# name the second detector `dut` (device under test)
[dut]
# set the type to the "timepix" detector model
type = "timepix"
# set the position in the world frame
```

```
15 position = 100um 100um 10mm
16 # rotate 20 degrees around the world x-axis
17 orientation = 20deg 0 0
18
19 # name the third detector `telescope2`
20 [telescope2]
21 # set the type to the "timepix" detector model
22 type = "timepix"
23 # place it 50 mm  away from the origin in z-direction
24 position = 0 0 50mm
25 # use the default orientation
26 orientation = 0 0 0
```

Figure 1 shows a visualisation 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 behaviour:

- **`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 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. 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 4.5. Can be overwritten by the `-v` parameter on the command line.
- **`log_format`**: Determines the log messages format to display. Possible options are `SHORT`, `DEFAULT` and `LONG`, where all options are case-insensitive. More information can be found in Section 4.5.
- **`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.
- **`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.
- **`random_seed`**: Seed for the global random seed generator used to initialise 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.

- **`library_directories`**: Additional directories to search for module libraries, before searching the default paths. See Section 5.3.3 for details.
- **`model_path`**: 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** multithreading for the framework. This can speed up simulations of multiple detectors significantly. More information about multithreading can be found in Section 5.3.4.
- **`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. 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 Section 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 visualisation).
- 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 4.6.

In this example, charge carriers will be deposited in the three sensors defined in the detector configuration file in Section 4.1.3. Only the charge carriers deposited in the sensors of the Timepix detector models will be propagated and digitised. Finally, some detector histograms for the device under test (DUT) will be recorded as ROOT histograms and all simulated objects, including the entry and exit positions of the simulated Monte Carlo particles (Monte Carlo truth), will be stored in the Allpix^2 ROOT file. An example configuration file implementing this would look like:

```
# Initialize the global configuration
[Allpix]
# Run 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, setup the particle source and deposit the
 ↪  charges
[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
beam_energy = 120GeV
# The position of the beam
beam_position = 0 0 -1mm
# The direction of the beam
beam_direction = 0 0 1
# Use a single particle in a single 'event'
number_of_particles = 1

# Propagate the charges in the sensor
[GenericPropagation]
# Only propagate charges in the Timepix sensors
type = "timepix"
# Set the temperature of the sensor
temperature = 293K
# Propagate multiple charges together in one step for faster
 ↪  simulation
charge_per_step = 50

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

# Digitize the propagated charges
[DefaultDigitizer]
# Noise added by the readout electronics
electronics_noise = 110e
# Threshold for a hit to be detected
threshold = 600e
# Threshold dispersion
threshold_smearing = 30e
# Noise added by the digitisation
adc_smearing = 100e

```

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

# Store all simulated objects to a ROOT file containing TTrees
[ROOTObjectWriter]
# File name of the output file
file_name = "allpix-squared_output"
```

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 can be executed by passing the main configuration to the **`allpix`** binary as follows:

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

The output should look similar to the sample log provided in Appendix A. 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 Section 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 Section 5.

4.4. Extending the Simulation Chain

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

Visualisation Displaying the geometry and the particle tracks helps both in checking and interpreting the results of a simulation. Visualisation is fully supported through Geant4, supporting all the options provided by Geant4 [14]. 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 visualisation, 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 visualisation 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 visualisation 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 Section 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 Section 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 utilise 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 10.2. 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 in the `init` electric field
 ↪  map format converted from TCAD
model = "init"
# Name of the file containing the electric field
file_name = "example_electric_field.init"
```

4.5. 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.

4.6. 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 [15]. 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 [16], 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 (without the ".root" suffix) to specify the file location of the created ROOT file in the global output directory. 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 analysed using ROOT macros. A simple macro for converting the results to a tree with standard branches for comparison is described in Section 10.3.

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] or the native RCE file format [17]. Detailed descriptions of these modules can be found in Section 7.

5. Structure & Components of the Framework

This section 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 initialise 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 Section 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 implemented using the Eigen3 library and the set of template specialisations for ROOT and Geant4 configurations. More information about the tools can be found in Section 10. 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.

This section 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 to 5.5. The section 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 centred 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 [18] 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 [18] 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 should be stripped by the parser. Empty lines should be ignored.

2. Every non-empty line should start with either `#`, `[` or an alphanumeric character. Every other character should lead to an immediate parsing error.

3. If the line starts with a hash character (`#`), it is interpreted as comment and all other content on the same line is ignored.

4. If the line starts with an open square bracket (`[`), it indicates a section header (also known as configuration header). The line should contain an alphanumeric string indicating the header name followed by a closing square bracket (`]`) to end the header (a missing `]` should raise an exception). 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. After any number of ignored whitespace characters there may be a `#` character. If this is the case, the rest of the line is handled as specified in point 3.

5. 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 and underscores, but it may not start with an underscore. 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 double quotes (`"`) 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.

6. The value can either be accessed as a single value or an array. If the value is accessed as an array, the string is split at every whitespace or comma character (`,`) not enclosed in a pair of `"` characters. All empty entities are ignored. All other entities are treated as single values in the array.

7. All single values are stored as a string containing at least one character. The conversion to the actual type is performed when accessing the value.

8. 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 value in the given type, throws an exception if not
    ↪  existing or a conversion to TYPE is not possible
4  config.get<TYPE>("key")
5  // Returns the value in the given type or the provided default value
    ↪  if it does not exist
6  config.get<TYPE>("key", default_value)
7  // Returns an array of single values of the given type; throws an
    ↪  exception if the key does not exist or a conversion is not
    ↪  possible
8  config.getArray<TYPE>("key")
9  // Returns an absolute (canonical if it should exist) path to a file
10 config.getPath("key", true /* check if path exists */)
11 // Return an array of absolute paths
12 config.getPathArray("key", false /* do not check if paths exists */)
13 // Returns the value as literal text including possible quotation
    ↪  marks
14 config.getText("key")
15 // Set the value of key to the default value if the key is not
    ↪  defined
16 config.setDefault("key", default_value)
17 // Set the value of the key to the default array if key is not
    ↪  defined
18 config.setDefaultArray<TYPE>("key", vector_of_default_values)
19 // Create an alias named new_key for the already existing old_key.
    ↪  Throws an exception if the old_key does not exist
20 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. More details about the instantiation logic for the different types of modules, see Section 5.3.3.

5.3.1. Files of a Module

Every module directory should at minimum contain the following documents (with `ModuleName` replaced by the name of the module):

- **CMakeLists.txt**: The build script to load the dependencies and define the source files of the library.
- **README.md**: Full documentation of the module.
- **`ModuleNameModule.hpp`**: The header file of the module.
- **`ModuleNameModule.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 8.1.

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 [19]. It is automatically converted to LaTeXusing Pandoc [20] and included in the user manual in Section 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 H2-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 H4-size section named **Description**, containing a short description of the module.

- An H4-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 H4-size section with the title **Usage** which should contain at least one simple example of a valid configuration for the module.

`ModuleNameModule.hpp` and `ModuleNameModule.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.

5.3.2. Module structure

All modules must inherit from the **`Module`** base class, which can be found in `src/core/module/Module.hpp`. The module base class provides two base constructors, a few convenient methods and several methods which the user is required to override. Each module should provide a constructor using the fixed set of arguments defined by the framework; this particular constructor is always called during by the module instantiation logic. These arguments for the constructor differ for unique and detector modules. For unique modules, the constructor for a **`TestModule`** should be:

```
1 TestModule(Configuration config, Messenger* messenger,
  ↪  GeometryManager* geo_manager): Module(config) {}
```

It should be noted that the configuration object must be forwarded to the base module.

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, detector) {}
```

The pointer to a Messenger can be used to bind variables to either receive or dispatch messages as explained in 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 initialise 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.

5.3.3. Module instantiation

Modules are dynamically loaded and instantiated by the Module Manager. They are constructed, initialised, executed and finalised 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 library are always named following the scheme `libAllpixModule`**`ModuleName`**, reflecting the **`ModuleName`** configured via CMake. The module search order is as follows:

1. Modules already loaded before from an earlier section header
2. All directories in the global configuration parameter **`library_directories`** in the provided order, if this parameter exists
3. The internal library paths of the executable, that should automatically point to the libraries that are built and installed together with the executable. These library paths are stored in RPATH on Linux, see the next point for more information.
4. The other standard locations to search for libraries depending on the operating system. Details about the procedure Linux follows can be found in [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.

5.3.4. 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 parallelise 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 parallelisation 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 centre 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 centre).
- An **`orientation_mode`** that determines the way that the orientation is applied. This can be either `xyz` or `zxz`, where `xyz` is used if the parameter is not specified. The `xyz` option uses extrinsic Euler angles to apply a rotation around the X-axis, followed by a rotation around the original global Y-axis and finally a rotation around the global Z-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").
- The **`orientation`** to specify the Euler angles in logical order, interpreted using the method above (or with the `xyz` method if the **`orientation_mode`** is not specified).
- A **`type`** for the detector model, for example *hybrid* or *monolithic*. The model defines 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 **electric field** in the sensitive device. An electric field can be added to a detector by a special module as demonstrated in Section 4.4.

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 centre 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 centre 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

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 5.3.3. 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 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 centre 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.

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.

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 8.2 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.
- **`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 specialised 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.

In addition, multiple layers of support can be added to the detector model. 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 centred around the middle of the pixel matrix (the rotation centre 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 centre of the model (thus the centre 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 centre 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. Section 7 provides details about the materials supported by the geometry builder module (**GeometryBuilderGeant4**).

The base parameters described above are the only set of parameters supported by the **monolithic** model.

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 centre of each single pixel in the grid.

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
}
```

Specialising 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 specialised 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 specialisation. 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 `models_path` 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 `$XDG_DATA_DIRS` with `Allpix` appended. The `$XDG_DATA_DIRS` variable defaults to `/usr/local/share/` (thus effectively `/usr/local/share/Allpix`) followed by `/usr/share/` (effectively `/usr/share/Allpix`).

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 Section 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 $\star$ 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 three are the most common methods and the fourth 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. Add a dependency to keep messages in memory for retrieving objects from the history. Several objects have related objects which can be retrieved as explained in Section 6.2. Linked objects can only be retrieved if the respective source objects remain in memory. Modules should explicitly mark objects that are accessed indirectly through the history of the messages, to ensure that these objects are not deleted. This method can be used as follows in a detector-specific **`TestModule`**, to keep **`PixelHitMessage`**s in memory without having direct access to them (**`PixelHitMessage`** can be replaced here with any other type of message):

```
TestModule(Configuration, Messenger* messenger,
 ↪  std::shared_ptr<Detector>) {
    messenger->addDependency<PixelHit>(this,
                                       /* No special message
                                        ↪  flags */
                                       MsgFlags::NONE);
}
```

4. 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 parallelised. 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 */
                                MsgFlags::NONE);
}
void listener(std::shared_ptr<Message<Object>> message) {
```

```
    // Do something with the received message ...
}
```

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.
- **NO_RESET**: Messages are by default automatically reset after the **run** function has been executed, to prevent older messages from previous events from reappearing. This behaviour can be disabled by setting this flag (this does not have any effect for listening functions). Setting this flag for single bound messages (without **ALLOW_OVERWRITE**, see below) would cause an exception to be raised if the message is overwritten in a later event.
- **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 $\star$.

5.6. Redirecting 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 $\star$ 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 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 4.5.

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

A description of the use of units in config files within Allpix^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 [22] 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 10.1.1.

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 4.5.

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.

Outside the core framework, exceptions can also be used directly by the modules. There are only two exceptions which should be used by typical modules to indicate errors:

- **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:

```
1  void run(unsigned int event_id) {
2      // Fetch a key from the configuration
3      std::string value = config.get("key");
4
5      // Check if it is a 'valid' value
6      if(value != 'A' && value != "B") {
7          // Raise an error if it the value is not valid
8          //   provide the configuration object, key and an
           ↪  explanation
9          throw InvalidValueError(config, "key", "A and B are the
           ↪  only allowed values");
10     }
11 }
```

- **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.

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:

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 [23]. The **`MCParticle`** additionally stores a parent **`MCParticle`** object, if available.

DepositedCharge

The set of charge carriers deposited by an ionising 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 digitiser 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. Modules are not required, but recommended, to bind the history of their created objects.

Object history is implemented using the ROOT TRef class [16], 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 [15]. 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. Inside the framework an explicit dependency should be added for all modules that use the object history, as explained in section 5.5. Whenever the related object is not in memory (either because it is not available or not fetched) a **`MissingReferenceException`** will be thrown.

If no explicit dependency is added to a linked object in history, the module could misbehave depending on other modules. It should also be noted that *all* intermediate objects need an explicit dependency to fetch deeper history (like the **`MCParticle`**s from **`PixelHit`**). Finally, the **`MissingReferenceException`** should be properly caught by modules to handle instances where the history is not available or incomplete.

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. CorryvreckanWriter

Maintainer: Daniel Hynds (daniel.hynds@cern.ch)
Status: Functional
Input: PixelHit

Description

Takes all digitised pixel hits and converts them into Corryvrekan pixel format. These are then written to an output file in the expected format to be read in by the reconstruction software.

Parameters

- `file_name` : Output filename (appended with .root)

Usage

Typical usage is:

```
[CorryvreckanWriter]
file_name = corryvreckan
```

7.2. 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.
- 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. 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 applying the threshold). Defaults to 110 electrons.
- `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.
- `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.

Usage

The default configuration is equal to the following:

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

7.3. DepositionGeant4

Maintainer: Koen Wolters (koen.wolters@cern.ch)
Status: Functional
Output: DepositedCharge, MCParticle

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 beam that will deposit charges in every event.

The particle type can be set via a string (particle_type) or by the respective PDG code (particle_code). Refer to the Geant4 webpage [24] for information about the available types of particles and the PDG particle code definition [23] for a list of the available particles and PDG codes.

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. The information about the truth particle passage is also fully available, with every deposit linked to a MCParticle. The parental hierarchy of the MCParticles is not always available in the current implementation.

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 [25].
- `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).
- `max_step_length` : Maximum length of a simulation step in every sensitive device. Defaults to 1um.
- `particle_type` : Type of the Geant4 particle to use in the source (string). Refer to the Geant4 documentation [24] for information about the available types of particles.
- `particle_code` : PDG code of the Geant4 particle to use in the source.
- `beam_energy` : Mean energy of the generated particles.
- `beam_energy_spread` : Energy spread of the generated particle beam.
- `beam_position` : Position of the particle beam/source in the world geometry.
- `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 particle as a unit vector.
- `number_of_particles` : Number of particles to generate in a single event. Defaults to one particle.

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+"
beam_energy = 120GeV
beam_position = 0 0 -1mm
```

```
beam_direction = 0 0 1
beam_divergence = 3mrad 0mrad
number_of_particles = 1
```

7.4. DetectorHistogrammer

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

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.

The module creates the following histograms:

- 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.
- Total number of pixel hits (event size) per event (an event can have multiple particles).
- Cluster sizes in x, y and total per cluster.

Parameters

No parameters

Usage

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

```
[DetectorHistogrammer]
name = "dut"
```

7.5. ElectricFieldReader

Maintainer: Koen Wolters (koen.wolters@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 always depleted from the implant side, the direction of the electric field depends on the sign of the bias voltage (with negative bias voltage the electric field vector points towards the backplane and vice versa). The 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.
- For electric fields in the *INIT* format it parses a file containing an electric field map in the INIT format also used by the PixelAV software [26]. 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.

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 **init**.
- `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**.
- `file_name` : Location of file containing the electric field in the INIT format. Only used if the *model* parameter has the value **init**.
- `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 INIT field is added, is given below

```
[ElectricFieldReader]
model = "linear"
bias_voltage = -150V
depletion_voltage = -50V

[ElectricFieldReader]
name = "dut"
model = "init"
# Should point to the example electric field in the repositories etc directory
file_name = "example_electric_field.init"
```

7.6. 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. [27]. 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 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 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 with fifth-order error estimation is used to integrate the electric field. 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 0.1nm.
- `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 0.5ps.
- `timestep_max` : Maximum step in time to use for the Runge-Kutta integration regardless of the spatial precision. Defaults to 0.1ns.
- `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.
- `output_plots` : Determines if output plots 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_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_plots` 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
```

7.7. 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:

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

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.
- `GDML_output_file` : Optional file to write the geometry to in GDML format. Can only be used if this Geant4 version is built with GDML support enabled and will throw an error otherwise. This feature is to be considered experimental as the GDML implementation of Geant4 is incomplete.

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
```

7.8. LCIOWriter

Maintainer: Andreas Nurnberg (andreas.nurnberg@cern.ch)
Status: Functional
Input: PixelHit

Description

Writes pixel hit data to LCIO file, compatible with the EUTelescope analysis framework [28].

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`.
- `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"
- `output_collection_name`: Name of the LCIO collection containing the pixel data. Default: "zsdata_m26"

Usage

```
[LCIOWriter]
file_name = "run000123-converter.slcio"
```

7.9. 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 (holes are ignored) 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, $`\beta`$ 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.

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.
- `output_plots`: Determines if plots should be generated.

Usage

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

7.10. RCEWriter

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

Description

Reads in the PixelHit messages and saves track data in the RCE format, appropriate for the Proteus telescope reconstruction software [29]. 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 (without the .root suffix) to create, relative to the output directory of the framework. The default filename is *rce_data.root*

Usage

To create the default file (with the name *rce_data.root*) an instantiation without arguments can be placed at the end of the main configuration:

```
[RCEWriter]
```

7.11. 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).

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"
```

7.12. 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 (without the .root suffix) to create, relative to the output directory of the framework.
- `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"
```

7.13. 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.

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.

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
```

7.14. 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 visualisation 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 [30] 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 [24] 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.

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
```

8. Module & Detector Development

This section 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.

8.1. Implementing a New Module

It is essential to carefully read the framework module manager documentation in Section 5.3, the information about the directory structure in Section 5.3.1 and the details of the module structure in Section 5.3.2 before creating a new module. Thereafter, the steps below should provide enough details for starting a new module, hereafter called **`ModuleName`**:

1. Run the module initialisation script at `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 [19] 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.

After this, it is up to the developer to implement all required functionality.

It should be kept in mind that writing more generic modules, which are not tied to a specific detector type or simulation, will allow other users to benefit from the development. Furthermore, it may be beneficial to split up modules to support the modular design of Allpix^2. Additional sources of documentation which may be useful during the development of a module include:

- The framework documentation in Section 5 for an introduction to the different parts of the framework.
- The module documentation in Section 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 [6].
- 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].

8.2. Adding a New Detector Model

Custom detector models based on the detector classes provided with Allpix^2 can easily be added to the framework. Crucial information to read before writing the model is provided in Section 5.2.1, which

describes the file format, Section 4.1.1 for information about the units used in Allpix^2 and Section 5.4, which describes the geometry and detector models. 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_path`** 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].

9. Frequently Asked Questions

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).

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). Please refer to Section 5.3.3 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. By default, both the input and the output of module are messages with an empty name. 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.

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 specialised in the detector configuration file. To specialise 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.

How do I access the history of a particular object?

Many objects can include an internal link to related other objects (for example the function **`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. However take notice that there are multiple layers between a **`PixelHit`** and its **`MCParticles`**, which are the **`PixelCharge`**,

PropagatedCharge and **DepositedCharge**. These should all be loaded in memory to make it possible to fetch the history. Because getting the Monte Carlo truth of a **PixelHit** is quite a common thing a **getMCParticles** method is available which searches all layers of the history and returns an exception if any of the in between steps is not available or not loaded. Please refer to Section 6.2 for more information.

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 10.2, instructions to import the generated field are provided in Section 4.4.

10. Additional Tools & Resources

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

10.1. Framework Tools

10.1.1. 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.

10.1.2. Runge-Kutta integrator

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

10.2. 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 [31].

The output .init file (with the same name as the .grd and .dat files) can be imported into Allpix^2. The INIT file has a header followed by a list of columns organized as

```
node.x node.y node.z e-field.x e-field.y e-field.z
```

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.

Usage

To run the program, the following command should be executed from the installation folder:

```
bin/tcad_dfise_converter/dfise_converter -f <file_name_prefix>
                                [<options>] [<arguments>]
```

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
-o <init_file_prefix> output file prefix without .init (defaults to file
      name of <file_prefix>)
-R <region> region name to be meshed (defaults to 'bulk')
-O <observable> observable to be interpolated (defaults Electric Field)
-r <radius> initial node neighbors search radius in um (defaults to 1 um)
-t <radius_threshold> minimum distance from node to new mesh point
      (defaults to 0 um)
-s <radius_step> radius step if no neighbor is found (defaults to 0.5 um)
-m <max_radius> maximum search radius (default is 10 um)
-i <index_cut> index cut during permutation on vertex neighbours
      (disabled by default)
-c <volume_cut> minimum volume for tetrahedron for non-coplanar vertices
      (defaults to minimum double value)
-x <mesh x_pitch> new regular mesh X pitch (defaults to 100)
-y <mesh_y_pitch> new regular mesh Y pitch (defaults to 100)
-z <mesh_z_pitch> new regular mesh Z pitch (defaults to 100)
-d <mesh_dimension> specify mesh dimensionality (defaults to 3)
-l <file> file to log to besides standard output (disabled by default)
-v <level> verbosity level (default reporiting level is INFO)
```

The output INIT file will be saved with the same *file_name_prefix* as the .grd and .dat files.

The *mesh_plotter* tool can be used from the installation folder as follows:

```
bin/tcad_dfise_converter/mesh_plotter -f <file_name>
                              [<options>] [<arguments>]
```

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.

10.3. ROOT Analysis Macros

Collection of macros demonstrating how to analyze data generated by the framework. Currently contains a single 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.

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++`
- Run the macro with `auto tree = constructComparisonTree(_file0, "name_of_dut")`
- Open a new file with `auto file = new TFile("output.root", "RECREATE")`
- 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")`

11. 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:

- Neal Gauvin, Université de Genève
- Moritz Kiehn, Université de Genève
- Salman Maqbool, CERN Summer Student
- Andreas Matthias Nürnberg, CERN
- Marko Petric, CERN
- Edoardo Rossi, DESY
- Paul Schütze, DESY
- Mateus Vicente Barreto Pinto, Université de Genève

The original AllPix [4, 5] has been developed by:

- Mathieu Benoit, Université de Genève
- John Idarraga, Leiden University

A. Output of Example Simulation

A possible output for the example simulation in Section 4.3 is given below:

```
(S) Welcome to Allpix^2 v1.0beta1+110^gc065eb9
(S) Initialized PRNG with system entropy seed 11350876086373902512
(S) Loaded 8 modules
(S) Initializing 15 module instantiations
(I) [I:DepositionGeant4] Using G4 physics list "QGSP_BERT"
(I) [I:ElectricFieldReader:telescope1] Setting linear electric field from
   -100V bias voltage and -50V depletion voltage
(I) [I:ElectricFieldReader:dut] Setting linear electric field from -100V
   bias voltage and -50V depletion voltage
(I) [I:ElectricFieldReader:telescope2] Setting linear electric field from
   -100V bias voltage and -50V depletion voltage
(S) Initialized 15 module instantiations
(S) Running event 1 of 5
(I) [R:DepositionGeant4] Deposited 87334 charges in sensor of detector
   telescope1
(I) [R:DepositionGeant4] Deposited 45780 charges in sensor of detector dut
(I) [R:DepositionGeant4] Deposited 49552 charges in sensor of detector
   telescope2
(I) [R:GenericPropagation:dut] Propagated 22890 charges in 458 steps in
   average time of 3.48832ns
(I) [R:GenericPropagation:telescope2] Propagated 24776 charges in 496 steps
    in average time of 3.49675ns
(I) [R:SimpleTransfer:dut] Transferred 22890 charges to 4 pixels
(I) [R:SimpleTransfer:telescope2] Transferred 24776 charges to 4 pixels
(I) [R:DefaultDigitizer:dut] Digitized 2 pixel hits
(I) [R:DefaultDigitizer:telescope2] Digitized 4 pixel hits
(S) Running event 2 of 5
(I) [R:DepositionGeant4] Deposited 72234 charges in sensor of detector
   telescope1
(I) [R:DepositionGeant4] Deposited 56736 charges in sensor of detector dut
(I) [R:DepositionGeant4] Deposited 55882 charges in sensor of detector
   telescope2
(I) [R:GenericPropagation:dut] Propagated 28368 charges in 568 steps in
   average time of 3.49392ns
(I) [R:GenericPropagation:telescope2] Propagated 27941 charges in 559 steps
    in average time of 3.49617ns
(I) [R:SimpleTransfer:dut] Transferred 28368 charges to 4 pixels
(I) [R:SimpleTransfer:telescope2] Transferred 27941 charges to 4 pixels
(I) [R:DefaultDigitizer:dut] Digitized 2 pixel hits
(I) [R:DefaultDigitizer:telescope2] Digitized 4 pixel hits
(S) Running event 3 of 5
(I) [R:DepositionGeant4] Deposited 66248 charges in sensor of detector
   telescope1
(I) [R:DepositionGeant4] Deposited 57228 charges in sensor of detector dut
(I) [R:DepositionGeant4] Deposited 59420 charges in sensor of detector
   telescope2
(I) [R:GenericPropagation:dut] Propagated 28614 charges in 573 steps in
   average time of 3.49467ns
(I) [R:GenericPropagation:telescope2] Propagated 29710 charges in 595 steps
    in average time of 3.49148ns
(I) [R:SimpleTransfer:dut] Transferred 28614 charges to 3 pixels
(I) [R:SimpleTransfer:telescope2] Transferred 29710 charges to 4 pixels
```

```
(I) [R:DefaultDigitizer:dut] Digitized 2 pixel hits
(I) [R:DefaultDigitizer:telescope2] Digitized 4 pixel hits
(S) Running event 4 of 5
(I) [R:DepositionGeant4] Deposited 65206 charges in sensor of detector
   telescope1
(I) [R:DepositionGeant4] Deposited 70198 charges in sensor of detector dut
(I) [R:DepositionGeant4] Deposited 41846 charges in sensor of detector
   telescope2
(I) [R:GenericPropagation:dut] Propagated 35099 charges in 702 steps in
   average time of 3.49416ns
(I) [R:GenericPropagation:telescope2] Propagated 20923 charges in 419 steps
    in average time of 3.48176ns
(I) [R:SimpleTransfer:dut] Transferred 35099 charges to 4 pixels
(I) [R:SimpleTransfer:telescope2] Transferred 20923 charges to 4 pixels
(I) [R:DefaultDigitizer:dut] Digitized 2 pixel hits
(I) [R:DefaultDigitizer:telescope2] Digitized 4 pixel hits
(S) Running event 5 of 5
(I) [R:DepositionGeant4] Deposited 62782 charges in sensor of detector
   telescope1
(I) [R:DepositionGeant4] Deposited 47698 charges in sensor of detector dut
(I) [R:DepositionGeant4] Deposited 53766 charges in sensor of detector
   telescope2
(I) [R:GenericPropagation:dut] Propagated 23849 charges in 477 steps in
   average time of 3.49966ns
(I) [R:GenericPropagation:telescope2] Propagated 26883 charges in 538 steps
    in average time of 3.49897ns
(I) [R:SimpleTransfer:dut] Transferred 23849 charges to 3 pixels
(I) [R:SimpleTransfer:telescope2] Transferred 26883 charges to 4 pixels
(I) [R:DefaultDigitizer:dut] Digitized 2 pixel hits
(I) [R:DefaultDigitizer:telescope2] Digitized 4 pixel hits
(S) Finished run of 5 events
(I) [F:DepositionGeant4] Deposited total of 1783820 charges in 6 sensor(s)
   (average of 59460 per sensor for every event)
(I) [F:GenericPropagation:dut] Propagated total of 138820 charges in 2778
   steps in average time of 3.4942ns
(I) [F:GenericPropagation:telescope2] Propagated total of 130233 charges in
    2607 steps in average time of 3.49347ns
(I) [F:SimpleTransfer:telescope1] Transferred total of 0 charges to 0
   different pixels
(I) [F:SimpleTransfer:dut] Transferred total of 138820 charges to 4
   different pixels
(I) [F:SimpleTransfer:telescope2] Transferred total of 130233 charges to 4
   different pixels
(I) [F:DefaultDigitizer:telescope1] Digitized 0 pixel hits in total
(I) [F:DefaultDigitizer:dut] Digitized 10 pixel hits in total
(I) [F:DefaultDigitizer:telescope2] Digitized 20 pixel hits in total
(I) [F:DetectorHistogrammer:dut] Plotted 10 hits in total, mean position is
    (126,125.5)
(S) [F:ROOTObjectWriter] Wrote 5498 objects to 12 branches in file:
                  /tmp/output/allpix-squared_output.root
(S) Finalization completed
(S) Executed 15 instantiations in 2 seconds, spending 58% of time in
   slowest instantiation DepositionGeant4
(I) Module GeometryBuilderGeant4 took 0.0356574 seconds
(I) Module DepositionGeant4 took 0.888584 seconds
(I) Module ElectricFieldReader:telescope1 took 0.000102932 seconds
```

```
(I) Module ElectricFieldReader:dut took 6.3408e-05 seconds
(I) Module ElectricFieldReader:telescope2 took 5.5951e-05 seconds
(I) Module DefaultDigitizer:telescope1 took 5.6189e-05 seconds
(I) Module DefaultDigitizer:dut took 0.000543031 seconds
(I) Module DefaultDigitizer:telescope2 took 0.000312817 seconds
(I) Module DetectorHistogrammer:dut took 0.0237165 seconds
(I) Module ROOTObjectWriter took 0.180625 seconds
(I) Module SimpleTransfer:telescope1 took 6.8943e-05 seconds
(I) Module SimpleTransfer:dut took 0.00504597 seconds
(I) Module SimpleTransfer:telescope2 took 0.00274271 seconds
(I) Module GenericPropagation:dut took 0.206209 seconds
(I) Module GenericPropagation:telescope2 took 0.188115 seconds
(S) Average processing time is 306 ms/event, event generation at 3 Hz
```

References

[1] S. Agostinelli et al., *Geant4 - a simulation toolkit*, Nucl. Instr. Meth. Phys. A **506** (2003) 250, ISSN: 0168-9002, DOI: https://doi.org/10.1016/S0168-9002(03)01368-8.

[2] R. Brun, F. Rademakers, *ROOT - An Object Oriented Data Analysis Framework*, AIHENP'96 Workshop, Lausanne, vol. 389, 1996, p. 81.

[3] D. Hynds, S. Spannagel, K. Wolters, *The Allpix^2 Website*, 2017, URL: https://cern.ch/allpix-squared/.

[4] M. Benoit, J. Idarraga, *The AllPix Simulation Framework*, 2017, URL: https://twiki.cern.ch/twiki/bin/view/Main/AllPix.

[5] M. Benoit, J. Idarraga, S. Arfaoui, *AllPix, Generic simulation for pixel detectors*, URL: https://github.com/ALLPix/allpix.

[6] D. Hynds, S. Spannagel, K. Wolters, *The Allpix^2 Code Documentation*, 2017, URL: http://cern.ch/allpix-squared/reference/.

[7] *The Allpix^2 Project Issue Tracker*, 2017, URL: https://gitlab.cern.ch/allpix-squared/allpix-squared/issues.

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

[9] R. Brun, F. 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 Allpix^2 Project Repository*, 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*, Nuclear Science Symposium and Medical Imaging Conference (NSS/MIC), IEEE, Anaheim, CA, 2012, p. 2075, DOI: 10.1109/NSSMIC.2012.6551478.

[13] X. Llopart et al., *Timepix, a 65k programmable pixel readout chip for arrival time, energy and/or photon counting measurements*, Nucl. Instr. Meth. Phys. A **581** (2007), VCI 2007 485, ISSN: 0168-9002, DOI: http://dx.doi.org/10.1016/j.nima.2007.08.079.

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

[15] R. Brun, F. Rademakers, *ROOT User's Guide, Trees*, URL: https://root.cern.ch/root/htmldoc/guides/users-guide/Trees.html.

[16] R. Brun, F. Rademakers, *ROOT User's Guide, Input/Output*, URL: https://root.cern.ch/root/htmldoc/guides/users-guide/InputOutput.html.

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

[18] T. Preston-Werner, *TOML, Tom's Obvious, Minimal Language*, URL: https://github.com/toml-lang/toml.

[19] J. Gruber, A. Swartz, *Markdown*, URL: https://daringfireball.net/projects/markdown/.

[20] J. MacFarlane, *Pandoc, A universal document converter*, URL: http://pandoc.org/.

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

[23] 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.

[24] Geant4 Collaboration, *Geant4 Particles*, URL: http://geant4.cern.ch/G4UsersDocuments/UsersGuides/ForApplicationDeveloper/html/TrackingAndPhysics/particle.html.

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

[26] M. Swartz, *A detailed simulation of the CMS pixel sensor*, tech. rep., 2002.

[27] C. Jacoboni et al., *A review of some charge transport properties of silicon*,
Solid State Electronics **20** (1977) 77, DOI: 10.1016/0038-1101(77)90054-5.

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

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

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

[31] J. Behley, V. Steinhage, A. B. Cremers,
Efficient radius neighbor search in three-dimensional point clouds,
2015 IEEE International Conference on Robotics and Automation (ICRA), 2015, p. 3625,
DOI: 10.1109/ICRA.2015.7139702.