Allpix^2 User Manual

Koen Wolters (koen.wolters@cern.ch)
Simon Spannagel (simon.spannagel@cern.ch)

July 6, 2022

Version v0.3

Contents

1. Quick Start **5**

2. Introduction **6**
2.1. Historical Summary 7
2.2. Scope of this Manual 7
2.3. Support and Reporting Issues 8
2.4. Contributing Code 8

3. Installation **9**
3.1. Supported Operating Systems 9
3.2. Prerequisites 9
3.3. Downloading the source code 10
3.4. Initializing the dependencies 10
3.5. Configuration via CMake 11
3.6. Compilation and installation 12
3.7. Testing 12

4. Getting Started **13**
4.1. Configuration Files 13
4.1.1. Parsing types and units 13
4.1.2. Main configuration 16
4.1.3. Detector configuration 17
4.2. Framework parameters 19
4.3. Setting up the Simulation Chain 20
4.4. Adding More Modules 23
4.5. Redirect Module Inputs and Outputs 25
4.6. Logging and Verbosity Levels 26
4.7. Storing Output Data 28

5. The Allpix² Framework **30**
5.1. Architecture of the Core 31
5.2. Configuration and Parameters 31
5.2.1. File format 32
5.2.2. Accessing parameters 33
5.3. Modules and the Module Manager 34
5.3.1. Files of a Module 34
5.3.2. Module structure 37
5.3.3. Module instantiation 38
5.4. Geometry and Detectors 39
5.4.1. Changing and accessing the geometry 39
5.4.2. Coordinate systems 40

5.4.3. Detector models 40
5.5. Passing Objects using Messages 44
5.5.1. Methods to process messages 45
5.5.2. Message flags 47
5.6. Logging and other Utilities 47
5.6.1. Logging system 48
5.6.2. Unit system 48
5.6.3. Internal utilities 49
5.7. Error Reporting and Exceptions 49

6. Objects **52**

7. Modules **54**
7.1. DefaultDigitizer 54
7.2. DepositionGeant4 55
7.3. DetectorHistogrammer 56
7.4. ElectricFieldReader 57
7.5. GenericPropagation 58
7.6. GeometryBuilderGeant4 61
7.7. GeometryBuilderTGeo 62
7.8. LCIOWriter 63
7.9. RCEWriter 63
7.10. ROOTObjectReader 64
7.11. ROOTObjectWriter 65
7.12. SimpleTransfer 66
7.13. VisualizationGeant4 66

8. Module & Detector Development **69**
8.1. Implementing a New Module 69
8.2. Adding a New Detector Model 70

9. Frequently Asked Questions **71**

10. Additional Tools & Resources **73**
10.1. Framework Tools 73
10.1.1. ROOT and Geant4 utilities 73
10.1.2. Runge-Kutta integrator 73
10.2. TCAD DF-ISE mesh converter 73
10.3. ROOT Analysis Macros 75
10.3.1. Remake project 75

11. Acknowledgments **76**

A. Output of Example Simulation **77**

References **81**

1. 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 Section 2 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 the geometry. The framework currently relies on the Geant4 [1], ROOT [2] and Eigen3 [3] libraries which need to be installed and loaded before using Allpix^2.

The minimal, default installation can be installed by executing the commands below. More detailed installation instructions can be found in Section 3.

```
$ git clone https://gitlab.cern.ch/simonspa/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 simulates the whole process, starting from the passage of the beam, the deposition of charges in the detectors, the particle propagation and the conversion of the collected charges to digitized pixel hits. All generated data is finally stored on disk on 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.

2. Introduction

Allpix^2 is a generic simulation framework for silicon tracker and vertex detectors written in modern C++. It is the successor of a previously developed simulation framework called AllPix [4, 5]. The goal of the Allpix^2 framework is to provide and easy-to-use package for simulating the performance of Silicon detectors, starting with the passage of ionizing radiation through the sensor and finishing with the the digitization of hits in the readout chip.

The framework builds upon other packages to perform tasks in the simulation chain, most notably Geant4 [1] for the deposition of charge carriers in the sensor and ROOT [2] for producing histograms and storing the produced data. The core of the framework focuses on the simulation of charge transport in semiconductor detectors and the digitization to hits in the frontend electronics. The framework does not perform a reconstruction of the particle tracks.

Allpix^2 is designed as a modular framework, allowing for an easy extension to more complex and specialized detector simulations. The modular setup also allows to separate the core of the framework from the implementation of the algorithms in the modules, leading to a framework which is both easier to understand and to maintain. Besides modularity, the Allpix^2 framework was designed with the following main design goals in mind (listed from most to least important):

1. Reflects 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 very 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

2.1. Historical Summary

Development of AllPix (the original version) started around 2012 as a generic simulation framework for pixel detectors. It has been succesfully used for simulating a variety of different detector setups through the years. Originally written as a Geant4 user application, the framework has grown ‘organically‘ as new features continued to be added. Around 2016, discussions between collaborators started, favoring a rewrite of the software from scratch. The envisaged improvements included better modularity, more extensive configuration options and an easier geometry setup.

Early development of Allpix^2 started in end of 2016, but most of the initial rework in modern C++ has been carried out in the framework of a technical student project between January and July 2017. The core of the framework, its utilities and functions as well as an intial set of simulation modules have been implemented and are available in the first published release.

2.2. 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 inteface and configuration possiblities and a detailed indtroduction to the code base for potential developers. This manual is designed to:

- Guide all 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 [**doxygen**] 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.

2.3. Support and Reporting Issues

As for most of the software used within the high-energy particle physics community, no professional support for this software can be offered. The authors are, however, happy to receive feedback on potential improvements or problem arising. Reports on issues, questions concerning the software as well as the documentation and suggestions for improvements are very much appreciated. These should preferably be brought up on the issues tracker of the project which can be found in the repository [6].

2.4. Contributing Code

Allpix^2 is a community project that lives 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 [6] 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 prevents from spending time on features which already exist or are currently developed by someone else.

The repository contains a few tools to facilitate contributions.

3. Installation

This section aims at providing 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. 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). Microsoft Windows is not supported.

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]. Otherwise all required dependencies need to be installed before building Allpix^2. Please refer to [7] 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 are necessary. For details about the dependencies and their installation visit the module documentation in Section 7. The following dependencies are needed to compile the standard installation:

- Geant4 [1]: Simulates the particle beam, depositing charges in the detectors with the help of the constructed geometry. See the instructions in [8] 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=ON
-DGEANT4_USE_GDML=ON
```

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

- Eigen3 [3]: Vector package to do 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 since it is a header-only library.

Extra flags needs 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 [9]. For production environments it is recommended to only download and use tagged software versions since the versions available from the git branches are considered development versions and might exhibit unexpected behavior.

However, 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/simonspa/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 run 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 its 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 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 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 [10] 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) 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 such, that it provides a set of automated tests which can be executed to ensure a correct compilation and functionality of the framework.

The tests use the example configuration delivered 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 your installation path. Remember that before running any Allpix^2 simulation, ROOT and likely Geant4 should be initialized. Refer to Section 3.4 on instructions 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. The configuration consists of several section headers within [and] brackets and a section without header at the start. Every section contain 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. Contain 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.
- 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 `*` 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
```

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)

```
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-sensitive). 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 being discarded by the parser. These section headers are useful for quickly enabling and disabling individual modules by replaing their actual name by an ignore section header.

All other section headers are used to instantiate modules. 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, but it depends on their type and configuration if this allowed. 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 amount of detectors. These kind of modules should only appear once in the whole configuration file unless a different inputs and outputs are used as explained in Section 4.5.
- 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 type 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 could be:

```
# 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
```

This configuration can however not be executed in practice because MyUniqueModule and MyDetectorModule do not exist. In the following paragraphs, a fully functional albeit simple configuration file with valid configuration including a detector configuration is presented.

4.1.3. Detector configuration

The detector configuration consist 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 have to be unique, thus using the same detector name multiple times is not possible. Every detector should contain all of the following parameters:

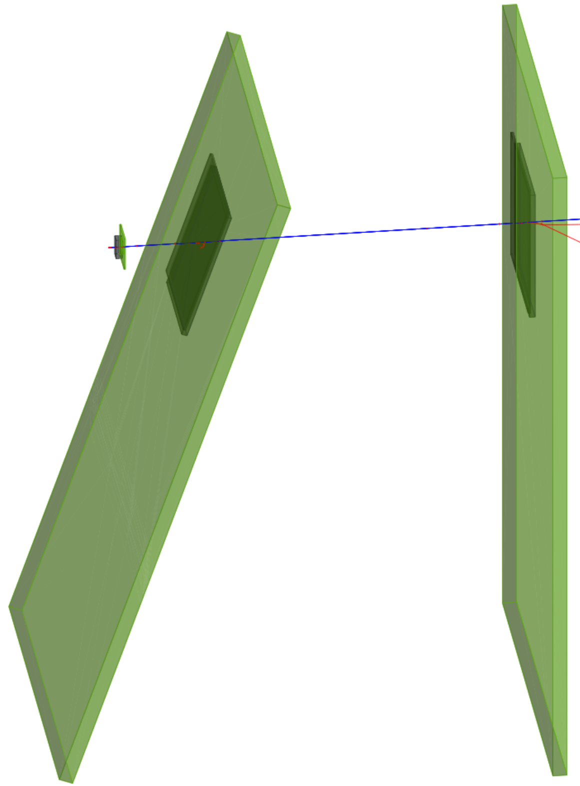

Figure 1: Visualization of a particle passing through the telescope setup defined in the detector configuration file

- A string referring to the **type** of the detector model. The model should exist in the search path described in Section 5.4.3.
- The 3D **position** in the world frame in the order x, y, z. See Section 5.4 for details.
- The **orientation** specified as Z-X-Z extrinsic Euler angle. This means the detector is rotated first around the world's Z-axis, then around the world's X-axis and then again around the global Z-axis. See Section 5.4 for details.

Furthermore it is possible to specialize certain parameters of the detector models, which is 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 [11] 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
position = 100um 100um 10mm
# rotate 20 degrees around the world x-axis
orientation = 0 20deg 0

# name the third detector 'telescope2'
[telescope2]
# set the type to the "timepix" detector model
type = "timepix"
# place it 50 mm up in the z-axis direction from the origin
position = 0 0 50mm
# use the default orientation
orientation = 0 0 0
```

Figure 1 shows a visualization of the setup described in the file. This configuration is used in the rest of this chapter for explaining concepts.

4.2. Framework parameters

The Allpix^2 framework provides a set of global parameters which control and alter its behavior:

- **`detectors_file`**: Location of the file describing the detector configuration (introduced in Section 4.1.3). The only required global parameter: the framework will fail if it is not specified.
- **`number_of_events`**: Determines the total number of events the framework should simulate. Equivalent to the amount of times the modules are run. 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 and how to change them for a particular module can be found in Section 4.6. Can be overwritten by the `-v` parameter on the command line.
- **`log_format`**: Determines the format to display. Possible options are `SHORT`, `DEFAULT` and `LONG`, where all options are case-insensitive. More information can be found in Section 4.6.
- **`log_file`**: File where the log output should be written to in addition to printing to the standard output (usually the terminal). Only writes to standard output if this option is not provided. Another (additional) location to write to can be specified on the command line using the `-l` parameter.
- **`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 initialize seeds for module instantiations. 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.

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 contains at least the following components.

- The **geometry builder**, responsible for creating the external Geant4 geometry from the internal geometry. In this document, *internal geometry* refers to the detector parameters used by Allpix^2 for coordinate transformations and conversions throughout the simulation, while *external geometry* refers to the constructed Geant4 geometry used for charge carrier deposition (and possibly visualization) only.
- The **deposition** module that simulates the particle beam that deposits 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 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 as will be detailed in Section 4.7.

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

```
# 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 one of the standard Geant4 physics lists
physics_list = QGSP_BERT
# Use a charged pion as particle
particle_type = "pi+"
# Set the energy of the particle
particle_energy = 120GeV
# The position of the point source
particle_position = 0 0 -1mm
# The direction of the source
particle_direction = 0 0 1
# Use a single particle in a single 'event'
number_of_particles = 1
```

```
27
28 # Specify a linear electric field for all detectors
29 # NOTE: This will be explained in more detail later in the manual
30 [ElectricFieldReader]
31 # Use a linear field
32 model = "linear"
33 # Applied bias voltage to calculate the electric field from
34 bias_voltage = -100V
35 # Depletion voltage at which the given sensor is fully depleted
36 depletion_voltage = -50V
37
38 # Propagate the charges in the sensor
39 [GenericPropagation]
40 # Only propagate charges in the Timepix sensors
41 type = "timepix"
42 # Set the temperature of the sensor
43 temperature = 293K
44 # Propagate multiple charges together in one step for faster simulation
45 charge_per_step = 50
46
47 # Transfer the propagated charges to the pixels
48 [SimpleTransfer]
49 max_depth_distance = 10um
50
51 # Digitize the propagated charges
52 [DefaultDigitizer]
53 # Input noise added by the electronics
54 electronics_noise = 110e
55 # Threshold for a hit to be detected
56 threshold = 600e
57 # Noise of the threshold level
58 threshold_smearing = 30e
59 # Uncertainty added by the digitization
60 adc_smearing = 100e
61
62 # Save histograms to the ROOT output file
63 [DetectorHistogrammer]
64 # Save histograms only for the dut
65 name = "dut"
66
67 # Store all simulated objects to a ROOT file containing TTrees
68 [ROOTObjectWriter]
69 # 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 Allpix² is used (see the installation instructions in Section 3). If modules and 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. Adding More Modules

In the following, a few basic modules are discussed which might be of use for a very first simulation.

Visualization Displaying the geometry and the particle tracks helps both in checking and interpreting the results of a simulation. Visualization is fully supported through Geant4, supporting all the options provided by Geant4 [12]. Using the Qt viewer with the OpenGL driver is the recommended option as long as the installed version of Geant4 is built with Qt support enabled.

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

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

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

```

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

If Qt is not available, a VRML viewer can be used as an alternative, however it is recommended to reinstall Geant4 with the Qt viewer included. 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 for example FreeWRL or OpenVRML.
- Subsequently, two environmental parameters have to be exported to the shell environment to inform Geant4 about the configuration: `G4VRMLFILE_VIEWER` should point to the location of the viewer executable and `G4VRMLFILE_MAX_FILE_NUM` should typically be set to 1 to prevent too many files from being created.
- Finally, the configuration section of the visualization module should be altered as follows:

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

More information about all possible configuration parameters can be found in the module documentation in Section 7.

Electric Fields The example configuration before already contained a module for adding a linear electric field to the detectors. By default, detectors do not have any electric field and no bias voltage is applied.

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 actually 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
```

Allpix² also provides the possibility to utilize a full electrostatic TCAD simulation for the description of the electric field. In order to speed up the lookup of the electric field values at different positions in the sensor, the adaptive TCAD mesh has to be interpolated and transformed into a regular grid with configurable feature size before using it. Allpix² comes with a converter tool which reads TCAD DF-ISE files from the sensor simulation, interpolates the field and writes it out in the appropriate format. A more detailed description of the tool can be found in Section 10.2. An example electric field (which the file name used in the example above) can be found in the *etc* directory of the Allpix² repository.

Electric fields can be attached to a specific detectors 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. Redirect Module Inputs and Outputs

In the Allpix² framework, modules by default exchange messages based on their in- and output message types and the detector type. It is, however, possible to specify a name for the incoming and outgoing message of every module in the simulation. This module will then only receive messages dispatched with the name provided and send named messages out to other modules listening for messages with a 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 the default input 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 allowed to set the **input** parameter to the special value * to indicate that the module should listen to all messages irrespective of their name.

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

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

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

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

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

4.6. Logging and Verbosity Levels

Allpix^2 is designed to identify mistakes and implementation errors as early as possible and tries to provide the user with clear indications about the problem. The amount of feedback can be controlled using different log 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 5.7. An example for a fatal error is an invalid configuration parameter.

- **STATUS**: Important informational messages about the status of the simulation. Is only used for informational messages which have to be logged in every run such as the global seed for pseudo-random number generators and the cuurent 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 reason of 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 a output message is not used and that a module may therefore do unnecessary work.
- **INFO**: Informational messages about the physics process of the simulation. Contains summaries about the simulation details of 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 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 could 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 does also support a few different formats to display the log messages. The following formats are supported via the global parameter **log_format** or the individual module parameter with the same name:

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

More details about the logging system and the procedure for reporting errors in the code can be found in Section 5.6.1 and 5.7.

4.7. Storing Output Data

Saving the output to persistent storage is of primary importance for later review and analysis. Allpix^2 primarily uses ROOT for storing output data, because it supports writing arbitrary objects and is a standard tool in High-Energy Physics. The `ROOTObjectWriter` automatically saves all objects created by the modules to a TTree [13]. It stores separate trees for all object types and creates branches for every unique message name, a combination of the detector, the module and the message output name as described in Section 4.5. For each event, values are added to the leafs 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 TRef allowing to fetch related objects as long as these are loaded in memory. An exception is thrown when trying to fetch an object which is not loaded.

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:

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

The generated output file can be analyzed using ROOT macros. A simple macro for converting the results to a tree with standard branches for comparisons is described in Section 10.3.1.

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 could be:

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

The Allpix^2 framework comes with a few more output modules which allow storing data in different formats, such as the LCIO persistency event data model [10] or the native RCE file format [14]. Detailed descriptions of these modules can be found in Section 7.

5. The Allpix^2 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 initiate the modules, to provide the geometry, to facilitate module communication and to 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 execute a part of the simulation chain. Modules are build 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 around 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 they can also be named to allow redirecting data to specific modules facilitating more sophisticated simulation setups. Messages are meant 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' providing access to common logic shared by various modules. Examples are the Runge-Kutta solver implemented using the Eigen3 library and the set of template specializations for ROOT and Geant4 configurations. More information about the tools can be found in Section 10. This set of tools is different from the set of core utilities the framework provides itself, which is part of the core and explained in Section 5.6

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. Finally, the section closes with a description of the available framework tools in Section 5.6. 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 also contains the part responsible for instantiating and running the modules from the supplied configuration file. The core is structured around five subsystems of which four are centered around a manager and the fifth contain a set of simple 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 [15] 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. The detector model contains 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 5.6. 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 informational messages and reporting errors, make the framework easier to use and debug. A few notes about the use and structure of exceptions are provided in Section 5.7.

5.2. Configuration and Parameters

Individual modules as well as the framework itself are configured through configuration files. Explanations on how to use the various configuration files together with several examples

have been provided in Section 4.1. All configuration files follow the same format, but the way their input is interpreted differs per configuration file.

5.2.1. File format

Throughout the framework, a simplified version of TOML [15] 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 header 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 an string of alphabetic characters, numbers and underscores, but it may not start with an underscore. This string indicates the 'key'. After a 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 not considered. 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:

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

Conversions to the requested type are using the `from_string` and `to_string` methods provided by the string utility library described in Section 5.6.3. These conversions largely follow the standard C++ parsing, with one important exception. If (and only if) the value is retrieved as any C/C++ string type and the string is fully enclosed by a pair of `"` characters, they 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 its core ideas is to partition functionality in independent modules. The modules are defined in the subdirectory *src/modules/* in the repository. The name of the directory is the unique name of the module. The suggested naming scheme is CamelCase, thus an exemplary module name would be *GenericPropagation*. There are two different kind of modules which can be defined:

- **Unique**: Modules for which always only one single instance runs irrespective of the number of detectors.
- **Detector**: Modules which are specific to a single detector. They are 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 can be found in 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.
- **`ModuleName`Module.hpp**: The header file of the module (note that another name can be used for this source file, but that is deprecated).
- **`ModuleName`Module.cpp**: The implementation file of the module.

The files are discussed in more detail below. By default, all modules are added to the *src/modules/* directory will be build automatically by CMake. This means that all subdirectories should feature a module with a *CMakeLists.txt* containing instructions on how to build the respective module.

If a module depends on additional packages which not every user might have installed, one can consider adding the following line to the top of the module's *CMakeLists.txt* (see below):

```
ALLPIX_ENABLE_DEFAULT(OFF)
```

Whether or not this is necessary for a given module will be decided on a case-by-case basis.

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 the module defined. The internal name of the module is automatically saved in the variable ${MODULE_NAME} which should be used as argument to other functions. Another name can be used by overwriting the variable content, but in the examples below, ${MODULE_NAME} is used exclusively.
2. The following lines should contain the logic to load possible dependencies of the module (below is an example to load Geant4). Only ROOT is automatically included and linked to the module.
3. A line with ALLPIX_MODULE_SOURCES(${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. Possibly 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)
```

```
8       MESSAGE(FATAL_ERROR "Could not find Geant4, make sure to source the
    ↪   Geant4 environment\n$ source YOUR_GEANT4_DIR/bin/geant4.sh")
9   ENDIF()
10
11  # Add the sources for this module
12  ALLPIX_MODULE_SOURCES(${MODULE_NAME}
13      TestModule.cpp
14  )
15
16  # Add Geant4 to the include directories
17  TARGET_INCLUDE_DIRECTORIES(${MODULE_NAME} SYSTEM PRIVATE
    ↪   ${Geant4_INCLUDE_DIRS})
18
19  # Link the Geant4 libraries to the module library
20  TARGET_LINK_LIBRARIES(${MODULE_NAME} ${Geant4_LIBRARIES})
21
22  # Provide standard install target
23  ALLPIX_MODULE_INSTALL(${MODULE_NAME})
```

README.md The README.md serves as the documentation for the module and should be written in the Markdown format [16]. It is automatically converted to LaTeXusing Pandoc [17] and included in this user manual in Section 7. By documenting the module functionality in Markdown, the information as also viewable with a web browser at the repository in the module subfolder.

The README.md should follow the structure indicated in the README file of the DummyModule in *src/modules/Dummy*. The documentation 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 entry should mention both the full name and email address of the module maintainer between parentheses. An example for a minimal header is therefore

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

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

- A H4-size section named **Description**, containing a short description of the module.

- A H4-size section named **Parameters** with all available configuration parameters of the module. The parameters should be briefly explained in an itemized list with the name of the parameter set as inline code block.
- A H4-size section with the title **Usage** which should contain at least one simple example of a valid configuration for the module.

`ModuleName`Module.hpp and `ModuleName`Module.cpp All modules should consist of both a header file and a source file. In the header file, the module is defined together with all its methods. Brief Doxygen documentation should be added to explain what every method does. The source file should provide the implementation of every method and also its more detailed Doxygen documentation. Method shall only be declared in the header and only defined in the source file to keep the interface clean.

5.3.2. Module structure

All modules have to 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 to override. Every module should provide a constructor consuming a fixed set of arguments defined by the framework. This particular constructor is always called during construction by the module instantiation logic. The arguments for the constructor differs for unique and detector modules. For unique modules, the constructor for a `TestModule` should be:

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

It should be noted that the configuration object has to 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 instead. It should always forward this detector to the base class together with the configuration object. Thus, the constructor of a detector module is:

```
TestModule(Configuration config, Messenger* messenger,
    std::shared_ptr<Detector> detector): Module(config, detector) {}
```

The pointer to the 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 the object of their respective detector.

In addition to the constructor, every module can override the following methods:

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

The modules are dynamically loaded and instantiated by the Module Manager. Modules are constructed, initialized, executed and finalized in the linear order they are defined in the configuration file. Thus the configuration file should follow the order of the real process. For every non-special section in the main configuration file (see 5.2 for more details), a corresponding library is searched for which contains the module. Module library are always named following the scheme **libAllpix`ModuleModuleName`** reflecting the `ModuleName` configured via CMake. The module search order is as follows:

1. Modules already loaded before from an earlier section header
2. All directories in the global configuration parameter *library_directories* in the provided order if this parameter exists
3. The internal library paths of the executable, that should automatically point to the libraries that are build 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 [18].

If the loading of the module library is successful, it is checked if the module is an unique or a detector module. 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*.

The instantiation logic only allows a single instance for every unique name. 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.

5.4. Geometry and Detectors

Simulations are frequently performed for a set of different detectors (such as a beam telescope and a device under test). All these individual detectors together is what Allpix^2 defines as the geometry. Every detector has a set of properties attached to it:

- A unique **detector name** to refer to the detector in the configuration.
- The **position** in the world frame. This is the position of the geometric center of the sensitive device (sensor) given in world coordinates as X, Y and Z (note that any additional components like the chip and possible support layers are ignored when determining the geometric center).
- The **orientation** given as Euler angles using the extrinsic Z-X-Z convention relative to the world frame (also known as the 1-3-1 or the "x-convention" and the most widely used definition of Euler angles [19]).
- A **type** of a detector model such as *hybrid* or *monolithic*. The model defines the geometry and parameters of the detector. Multiple detectors can share the same model. Several ready-to-use models are shipped 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 is explained in Section 4.1.3.

5.4.1. 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, the detector models are loaded and linked later during geometry closing as described above. It is also possible to add additional models and detectors directly using `addModel` and `addDetector` (before the geometry is

closed). Furthermore it is possible to add additional points which should be part of the world geometry using `addPoint`. This can for example be used to add the beam source to the world geometry.

The detectors and models can be accessed by name and type through the geometry manager using `getDetector` and `getModel`, respectively. All detectors can be fetched at once using the `getDetectors` method. If the module is a detector-specific module its related Detector can be accessed through the `getDetector` method of the module base class instead (returns a null pointer for unique modules) as follows:

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

5.4.2. Coordinate systems

All detectors have a fixed position in the world frame which has an arbitrary origin. Every detector also has a local coordinate system attached to it. The origin of this local coordinate system does usually not correspond with the geometric center of the sensitive device, which is the center of rotation of the detector in the global frame. The origin of the local coordinate system is instead based on the pixel grid in the sensor. The origin of the local coordinate system is fixed to the center of the first pixel in the grid, which allows for simpler calculations through the framework that are also easier to understand.

While the actual origin of the local coordinate system depends on the type of the model, there are fixed rules for the orientation of the coordinate system. The positive z-axis should point in the direction the particle beam is supposed to enter the sensor, perpendicular to the 2D pixel grid. The x-axis should be in the plane that defines the pixel grid. It should be in horizontal direction perpendicular to the direction of the beam, if the sensor is placed unrotated in a horizontal beam. The y-axis should be normal to both the x- and the z-axis in such a way that a right-handed coordinate system is constructed.

5.4.3. Detector models

Different types of detector models are already available and shipped with the framework. The configuration for these standard models use the configuration format introduced in Section 5.2.1 and can be found in the *models* directory in the repository. Every models extends from the `DetectorModel` base class which defines the minimum parameter of a detector model in the framework:

- The coordinate of the center in the local frame. This is the location of the local point which is defined as position in the global frame, and the rotation center for the specified orientation.
- The number of pixels in the sensor in both the x- and y-axis. Every pixel is an independent block replicated over the x,y-plane of the sensor.
- The size of an individual pixel. The multiplication of the pixel size and the number of pixels is known as the pixel grid and goes over the full x,y-plane.
- The sensor with a center and a size. The sensor is at least as large as the pixel grid size and has a certain thickness. It can have excess length in the x,y-plane in each direction.
- The readout chip with a center and a size. It is positioned directly after the sensor by default. The chip can also have an excess as described above for the sensor.
- Possible support layers with a center and a size. It is positioned directly after the sensor and the chip by default. The support layer can be of various materials and possibly contain a cutout.
- Total size of the box with the local frame center in the middle that fit all elements of the model.

This standard detector model can be extended to provide a more detailed geometry if required by particular modules (most notably the Geant4 geometry builder). The position and size of all elements can be changed by these extending models. A model with only the standard elements described above is the `MonolithicPixelDetectorModel`. Currently the only extended detector model is the `HybridPixelDetectorModel`, which also include bump bonds between the sensor and the readout chip.

Detector model parameters

Models are defined in configuration files which are used to instatiate the actual model classes. These files for detector models can contain various types of parameters. Some are required for all models, other optional for all models and there are also parameters only supported by certain types of models. For more details about the steps to perform to add and use your own new model, Section 8.2 should be consulted.

The set of base parameters supported by every models is provided below. These parameters should be given at the top of the file before opening any sections.

- `type`: A required parameter describing the type of the model. At the moment either **monolithic** or **hybrid**. This value determines any optional extra supported parameters discussed later.
- `number_of_pixels`: The number of pixels in the 2D pixel grid. Determines the base size of the sensor together with the `pixel_size` parameter below.

- `pixel_size`: The size of a single pixel in the pixel grid. Given in 2D as pixels do not have any direct thickness. 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`: Fallback for the excess width of the sensor in all four directions (top, bottom, left and right). Used if the specialized parameters described below are not given. Defaults to zero, thus having a sensor size equal to the number of pixels times the size of a single pixel.
- `sensor_excess_`*`direction`*: With direction either *top*, *bottom*, *left* or *right*, where the top, bottom, right and left direction are respectively the positive y-axis, the negative y-axis, the positive x-axis and the negative x-axis. It specifies the extra excess length added to the sensor in the specific direction.
- `chip_thickness`: Thickness of the readout chip, placed next to the sensor.
- `chip_excess`: Fallback for the excess width of the chip, defaults to zero thus a chip equal to the size of the pixel grid. See the `sensor_excess` parameter above.
- `chip_excess_`*`direction`*: With direction either *top*, *bottom*, *left* or *right*. The chip excess in the specific direction, see the `sensor_excess_`*`direction`* parameter above.

Besides these base parameters, several base layers of support can be added to detector models. Every layer of support should be given in its own section with the name `support`. By default there are no support layers. The support layers support 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 (on the opposite side of the chip), *chip* to add the support layer after the chip or *absolute* to specify the offset in the z-direction manually. Defaults to *chip* if not given. If the parameter is equal to *sensor* or *chip*, the support layers are stacked in there respective direction when multiple layers of support are specified.
- `offset`: When 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 centered by default to the middle of the pixel grid (the rotation center of the model). If the `location` is set to absolute, the offset is a required parameter and given as a 3D vector with respect to the center of the model (thus the center of the active sensor). Care should be taken to ensure that these support layers and the rest of the model do not overlap. Either *sensor* to stick it to the sensor (on the opposite side of the chip) or *chip* to add the support layer

after the chip. Defaults to *chip* if not given. Sensors are stacked in there respectively direction if multiple layers of support are given.

- hole_size: Adds an optional cut-out hole to the support with the 2D size provided. The hole always covers the full support thickness. No hole will be added if this parameter is not given.
- hole_offset: The hole is added by default to the center of the support layer. A 2D offset from this default center can be specified using this parameter.
- material: Material of the support to use, given as a lowercase string. There is no default set of materials and support for certain types of materials is up to the modules. Refer to Section 7 for details about the materials supported by the geometry creator module.

The base parameters are the only set of parameters supported by the **monolithic** model. The **hybrid** model add bump bonds between the chip and the sensor while automatically making sure the chip and support layers are shifted appropriately. The set of extra parameters for the **hybrid** model are the following (these should be put in the empty start section):

- 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 set the radius of the sphere to use, which should generally be smaller than the height of the bump.
- bump_cylinder_radius: The radius of the cylinder part of the bump. The height of the cylinder is determined by the bump_height parameter.
- bump_offset: A 2D offset of the grid of bumps. The individual bumps are by default positioned at the center of all the single pixels in the grid.

Fetching specific models within the framework

Some modules are specific for a particular type of detector model. To fetch a specific detector model from the base class, the model should be downcasted. An example to try fetching an HybridPixelDetectorModel is the following (the downcast return a null pointer if the class is not of the appropriate type).

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

```
    // The model of this Detector is a HybridPixelDetectorModel
}
```

Specializing detector models

A detector model contains default values for all the parameters. Some parameters like the sensor thickness can however vary between different detectors of the same general model. To allow for easy adjustment of these parameters, models can be specialized in the detector configuration file introduced in 4.1.3. All of the model parameters, except the type parameter, in the header at the top (thus not the support layers) can be changed by adding a parameter with the exact same key to the detector model file with the specialized value. The framework will then internally automatically create a copy of this model with the requested change.

Search order for models

To support different detector models and storage locations the framework supports model readers. The core geometry manager does also read models and will read all remaining models, not parsed earlier, before the geometry is closed. The model readers and the core geometry manager should search for model files in the following order.

1. If defined, the paths in the *models_path* parameter provided to the model reader module or the global *models_path* parameter if no module-specific one is defined (the geometry manager only uses the global one). Files are read and parsed directly. If the path is a directory, all files in the directory are added (not recursing into subdirectories).
2. The location where the models are installed to (see 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 the allpix-directory 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*).

For almost all purposes a specific model reader is not needed and all internal models can be read by the geometry manager.

5.5. Passing Objects using Messages

Communication between modules happens through messages (only some internal information is shared through external detector objects and the dependencies like Geant4). 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 part to send messages and a receiving part that fetches 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 directly 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 the message to their receivers then goes by the following rules:

1. The receiving module the will only receive a message if it has the exact same type as the message dispatched (thus carrying the exact same object). If the receiver is however listening to the `BaseMessage` type it will receive all dispatched messages instead.
2. The receiving module will only receive messages with the exact same name as it is listening for. The module uses the **input** parameter to determine to which message names the module should listen. If the **input** parameter is equal to `*` the module should listen to all messages. Every module listens by default to messages with no name specified (thus receiving the messages of default dispatching modules).
3. If the receiving module is a detector module it will only receive messages that are bound to that specific detector or messages that are not bound to any detector.

An example how to dispatch, in the `run` function of a module, a message containing an array of `Object` types bound to a detector named `dut` is provided here:

```
void run(unsigned int event_id) {
    std::vector<Object> data;
    // ..fill the data vector with objects ...

    // The message is dispatched only for 'dut' detector
    std::shared_ptr<Message<Object>> message =
    std::make_shared<Message<Object>>(data, "dut");

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

5.5.1. Methods to process messages

The message system has multiple methods to process received messages. The first two are the most common methods and the third should only be used if necessary. The options are:

1. Bind a **single message** to a variable. This should usually be the preferred method as most modules only expect one message to arrive per event (as a module should typically send only one message containing the list of all the objects it should send). An example of how to bind a message containing an array of **Object** types in the constructor of a detector `TestModule` would be:

```
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 an vector variable. This method should be used it 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 **Object** types in the constructor of a detector `TestModule` would be:

```
TestModule(Configuration, Messenger* messenger,
 ↪  std::shared_ptr<Detector>) {
    messenger->bindMulti(this,
                          /* Pointer to the message vector */
                          &TestModule::messages,
                          /* No special messenger flags */
                          MsgFlags::NONE);
}
std::vector<std::shared_ptr<Message<Object>>> messages;
```

3. Listen to a particular message type and execute a **listener function** as soon as an object is received. Can be used for more advanced strategies for fetching messages. Note that this method can lead to surprising behaviour because the listener function is executed during the run of the dispatching module (leading to log messages with incorrect section headers at the minimum). The listening module should not do any heavy work in the listening function as this is supposed to take place in their `run` method instead. An example of using this to listen to a message containing an array of `Object` types in a detector `TestModule` would be:

```
TestModule(Configuration, Messenger* messenger,
 ↪  std::shared_ptr<Detector>) {
    messenger->registerListener(this,
```

```
3                                     /* Pointer to the listener method
   ↪ */
4                                     &TestModule::listener,
5                                     /* No special message flags */
6                                     MsgFlags::NONE);
7  }
8  void listener(std::shared_ptr<Message<Object>> message) {
9      // Do something with received message ...
10 }
```

5.5.2. Message flags

Various flags can be added to the bind function and listening functions. The flags enable a particular behaviour of the framework (if the particular type of method supports the flag).

- **REQUIRED**: Specify that this message is required to be received. If the particular type of message is not received before it is time to execute the run function, the run is automatically skipped by the framework. This can be used to ignore modules that cannot do any action without received messages, for example propagation without any deposited charges.
- **NO_RESET**: Messages are by default automatically reset after the `run` function executes to prevent older messages from previous runs to appear again. 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) 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 setting it multiple times instead of once). This flag prevents this behaviour. It is only used for variables to a single message.
- **IGNORE_NAME**: If this flag is specified, the name of the dispatched message is not considered. Thus the **input** parameter is ignored and forced to the value `*`.

5.6. Logging and other Utilities

The Allpix^2 framework provides a set of utilities that can be attributed to two types:

- Two utilities to improve the usability of the framework. One of these is a flexible and easy-to-use logging system, introduced below in Section 5.6.1. The other is an easy-to-use framework for units that supports converting arbitrary combinations

of units to an independent number which can be used transparently through the framework. It will be discussed in more detail in Section 5.6.2.

- A few utilities to extend the functionality provided by the C++ Standard Template Library (STL). These are provided to provide functionality the C++14 standard lacks (like filesystem support). The utilities are used internally in the framework and are only shortly discussed here. The utilities falling in this category are the filesystem functions (see Section 5.6.3) and the string utilies (see Section 5.6.3).

5.6.1. Logging system

The logging system is build to handle input/output in the same way as `std::cin` and `std::cout`. This approach is both very flexible and easy to read. The system is globally configured, thus there exists only one logger, and no special local versions. 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 newline 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 the new lines.

The system also allows for producing a message which is updated on the same line for simple progress bar like functionality. It is enabled using the `LOG_PROCESS(LEVEL, IDENTIFIER)` macro (where the `IDENTIFIER` is a special string to determine if the output should be written to the same line or not). If the output is a terminal screen the logging output will be colored to make it prettier to read. This will be disabled automatically for all devices that are not terminals.

More details about the various logging levels can be found in Section 4.6.

5.6.2. Unit system

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

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:

```cpp
// 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 of mm/ns in m/ns
Units::convert(1, "m/ns");
// Give the unit in the best type (lowest number above one) as string
//   input is default unit 2000mm/ns and 'best' output is 2m/ns
↪ (string)
Units::display(2e3, {"mm/ns", "m/ns"});
```

More details about how the unit system is used within Allpix^2 can be found in Section 4.1.1.

5.6.3. Internal utilities

Filesystem Provides 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 [20] as soon as the framework minimum standard is updated to C++17.

String utilities The STL only provides string conversions for standard types using `std::stringstream` and `std::to_string`. It does not allow to parse strings encapsulated in pairs of `"` characters and neither does it allow to integrate 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` do allow for these flexible conversions and it it extensively used in the configuration system. Conversions of numeric types with a unit attached are automatically resolved using the unit system discussed in Section 5.6.2. The Allpix^2 tools system contain extensions to allow automatic conversions for ROOT and Geant4 types as explained in Section 10.1.1. The string utilities also include trim and split strings functions as they are missing in the STL.

5.7. Error Reporting and Exceptions

Allpix^2 generally follows the principle to throw exceptions in all cases where something is definitely wrong, it should never try to circumvent problems. Also error codes are not supposed to be returned, only exceptions should be used to report fatal errors. Exceptions

are also thrown to signal for errors in the user configuration. The asset of this method is that configuration and code is more likely to do what they are supposed to do.

For warnings and informational messages the logging should be used extensively. This helps in both following the progress of the simulation as well as for debugging problems. Care should however be taken to limit the amount of messages outside of the `DEBUG` and `TRACE` levels. More details about the log levels and their usage is given in Section 4.6.

The base exceptions in Allpix^2 are available in the utilities. The most important exception base classes are the following:

- **ConfigurationError**: All errors related to incorrect user configuration. Could be 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 running a module.
- **LogicError**: Problems related to modules that 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 a well-behaving module and should therefore not be triggerable by users. Reporting these type of errors can help developers during their development of new modules.

Outside of 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**: Available under the subset of configuration exceptions. Signals any problem with the value of a configuration parameter that is not related to either the parsing or the 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 configuration object, key and an explanation
9          throw InvalidValueError(config, "key", "A and B are the only
   ↪   allowed values");
10     }
11 }
```

- **ModuleError**: Available under the subset of module exceptions. Should be used to indicate any runtime error in a module that is not directly caused by an invalid configuration value. For example if it is not possible to write an output. A reason should be given to indicate what the problem is.

6. Objects

Allpix^2 ships a set of objects that should be used to transfer data between modules. These objects can be send with the messaging system explained in Section 5.5. A typedef is added to every object to provide an alternative name for the message directly linking to the carried object.

Currently this list of supported objects are the following:

MCParticle

The Monte-Carlo truth information about the particle passage through the sensor. Both the entry and the exit point are stored in the object, to approximate the track. The exact handling of non-linear tracks due to for example in-sensor nuclear interactions, is up to module. The MCParticle also stores an identifier of the particle type. The naming scheme is again up to the module, but it is recommended to use PDG codes [21].

DepositedCharge

Set of charges that are deposited by an ionizing particle crossing the active material of the sensor. The object stores both the local position in the sensor together with the total number of deposited charges in elementary charge units. Also the time (in *ns* the internal framework unit) of the deposition after the start of the event is stored.

PropagatedCharge

Set of charges that are propagated through the silicon sensor due to drift and/or diffusion processes. The object should store the final local position of the propagation. This is either on the pixel implant if the set of charges are ready to be collected, or on any other position in the sensor if the set of charges got stuck or lost in another process. Timing information about the total time to arrive at the final location, from the start of the event, can also be stored.

PixelCharge

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

PixelHit
Digitized hit of a pixel after digitization. The object allows to store both the time and a signal value. The time can be stored in an arbitrary unit used to timestamp the hits. The signal can also store different kind of information depending on the type of the digitizer used. Examples of the signal information is the 'true' charge, the number of ADC counts or the ToT (time-over-threshold).

7. Modules

7.1. 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 allow for a configurable threshold.

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

- A Gaussian noise is added to the input charge value in order to simulate input noise to the preamplifier circuit
- 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 the simulation of fluctuations of the threshold level.
- An inaccuracy of the ADC is simulated using an additional Gaussian smearing, this allows to take ADC noise into account.

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 readout 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_smearing` : Standard deviation of the Gaussian noise in the ADC conversion (after applying the threshold). Defaults to 300 electrons.
- `output_plots` : Enables output histograms to be be generated from the data in every step (slows down simulation considerably). Disabled by default.

Usage

The default configuration is equal to the following

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

7.2. DepositionGeant4

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

Description

Module that creates the deposits in the sensitive devices, wrapper around the Geant4 logic. Depends on a geometry construction in the GeometryBuilderGeant4 module. Initializes the physical processes to simulate and create a particle source that will generate particles in every event. For all particles passing the detectors in the geometry, the energy loss is converted into charge deposits for all steps (of customizable size) in the sensor. The information about the truth particle passage is also made available for later modules.

Parameters

- `physics_list`: Internal Geant4 list of physical processes to simulate. More information about possible physics list and recommendations for default is available here.
- `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.
- `particle_position` : Position of the particle source in the world geometry.
- `particle_type` : Type of the Geant4 particle to use in the source. Refer to this page for information about the available types of particles.
- `particle_radius_sigma` : Standard deviation of the radius from the particle source.
- `particle_direction` : Direction of the particle as a unit vector.
- `particle_energy` : Energy of the generated particle.
- `number_of_particles` : Number of particles to generate in a single event. Defaults to one particle.

Usage

A solid default configuration to use, simulating a test beam of 120 GeV pions, is the following:

```
[DepositionGeant4]
physics_list = QGSP_BERT
particle_type = "pi+"
particle_energy = 120GeV
particle_position = 0 0 -1mm
particle_direction = 0 0 1
number_of_particles = 1
```

7.3. DetectorHistogrammer

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

Description

This module should only give 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 produce the necessary information.

Within the module, a clustering 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"
```

7.4. ElectricFieldReader

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

Description

Adds an electric field to the detector from the standard supported sources. By default every detector has no electric field in the sensitive device.

The reader does work with two models of electric field to read:

- For *constant* electric fields it add a constant electric field in the z-direction towards the pixel implants.
- 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).
- For electric fields in the *INIT* format it parses a file the INIT format used in the PixelAV software. 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.

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**.
- `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 (slows down simulation). Disabled by default.

- `output_plots_steps` : Number of bins in both the 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 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 of 50 volt bias to all the detectors, apart from the detector with name 'dut' where a specific INIT field is added, is given below

```
[ElectricFieldReader]
model = "linear"
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"
```

7.5. GenericPropagation

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

Description

Simulates generic propagation of electrons (ignoring the corresponding holes) through the sensitive devices of every detector. Splits up the set of deposited charges in multiple smaller sets of charges (containing multiple charges) that are propagated together. The propagation process is fully independent, the individual sets of propagated charges do not influence

each other. The maximum size of the set of propagated charges and 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 electron mobility parameterization by C. Jacobini et al. in A review of some charge transport properties of silicon. 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.

An fourth-order Runge-Kutta-Fehlberg method with fifth-order error estimation is used to integrate the electric field. After every Runge-Kutta step a random walk is simulated by applying Gaussian diffusion calculated from the electron mobility, the temperature and the time step. The propagation stops when the set of charges reaches the border of the sensor.

The propagation module also produces a variety of output plots for debugging and publication purposes. The plots include a 3D line plot of the path of all separate propagated charges from their deposits, 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.

Parameters

- `temperature` : Temperature in the sensitive device, used to estimate the diffusion constant and therefore the strength of the diffusion.
- `charge_per_step` : Maximum number of charges to propagate together. Divides the total deposited charge at a specific point in sets of this number of charges and a set with the remaining amount of charges. A value of 10 charges per step is used 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 error from the fifth-order error method. Defaults to 0.1nm.
- `timestep_start` : Timestep to initialize the Runge-Kutta integration with. Better 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 very huge slow down of the simulation, it is not recommended to use this with a run of more than a single event. Disabled by default.
- `output_plots_step` : Timestep to use between two points that are 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_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.
- `output_animations_time_scaling` : Scaling for the animation to use to convert the actual simulation time to the time step in the animation. Defaults to 1.0e9, meaning that every nanosecond is equal to an animation step of a single second.
- `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.

Usage

A example of generic propagation for all Timepix sensors at room temperature using packets of 25 charges is the following:

```
[GenericPropagation]
type = "timepix"
temperature = 293K
charge_per_step = 25
```

7.6. GeometryBuilderGeant4

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

Description

Constructs the Geant4 geometry from the internal geometry. First constructs the world geometry from the internal world size, with a certain margin, using a particular world material. Then continues to create all the detectors using their internal detector models.

All the available detector models are fully supported. This builder can create extra support layers of the following materials (note that these should be specified in lowercase):

- silicon
- plexiglass
- kapton
- copper
- epoxy
- carbonfiber
- g10
- solder

Parameters

- `world_material` : Material of the world, should either be **air** or **vacuum**. Default to **air** if not specified.
- `world_margin_percentage` : Percentage of the world size to add extra 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 having any minimum margin.

- `GDML_output_file` : Optional file to write the geometry to in GDML format. Can only be used if this Geant4 version has GDML support (will throw an error otherwise). Otherwise also likely produces an error due to incomplete GDML implementation in Geant4.

Usage

To create a Geant4 geometry using vacuum as world material and with always exactly one meter added to the minimum world size on every side, the following configuration can be used.

```
[GeometryBuilderGeant4]
world_material = "vacuum"
world_margin_percentage = 0
world_minimum_margin = 1m 1m 1m
```

7.7. GeometryBuilderTGeo

Maintainer: Neal Gauvin (neal.gauvin@unige.ch)
Status: OUTDATED (not supported)

Description

Constructs an TGeo representation of the internal geometry. Creates all detector devices and also add optional appliances and an optional test structure. Code is based on Geant4 geometry construction in original AllPix. Only supports hybrid pixel detectors.

Parameters

- `world_material` : Material used to use to represent the world. There are two possible options, either **Vacuum** or **Air**.
- `world_size` : Size of the world (centered at the origin). Defaults to a box of 1x1x1 cubic meter.
- `build_appliances` : Determines if appliances are enabled.
- `appliances_type` : Type of the appliances to be constructed (see source code for options). Only used if *build_appliances* is enabled.
- `build_test_structures` : Determines if the test structure has to be build.
- `output_file` : Optional ROOT file to write the constructed geometry into
- `GDML_output_file` : Optional file to write geometry to in the GDML format. Can only be used if the used ROOT version has GDML support (will throw an error otherwise).

Usage

An example to construct a simple TGeo geometry without appliances and no test structure and a world of 5x5x5 cubic meters.

```
[GeometryBuilderTGeo]
world_material = "Air"
world_size = 5m 5m 5m
build_appliances = 0
build_test_structures = 0
```

7.8. LCIOWriter

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

Description

Writes pixel hit data to LCIO file, compatible to EUTelescope analysis framework.

Parameters

- `file_name`: LCIO file to write. Extension .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]
```

7.9. RCEWriter

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

Description

Reads in the Pixel hit messages and saves track data in the RCE format, appropriate for the Proteus telescope reconstruction software. An event tree and a sensor tree and their branches are initialized when the module first runs. The event tree is initialized with the appropriate branches, while a sensor tree is created for each detector and the brances initialized from a strcut. Initially, the program loops over all the pixel hit messages, and then over all the hits in 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 configuration:

```
[RCEWriter]
```

7.10. ROOTObjectReader

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

Description

Converts all the object data stored in the ROOT data file produced by ROOTObjectWriter back in to messages (see the description of ROOTObjectWriter for more information about the format). Reads all the trees defined in the data file that contain Allpix objects. Creates a message from the objects in the tree for every event (as long as the file contains the same number of events as used in the simulation).

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

Usage

This module should be at the beginning of the main configuration. An example to read the objects from the file *data.root* is:

```
[ROOTObjectReader]
file_name = "data.root"
```

7.11. ROOTObjectWriter

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

Description

Reads in all the messages dispatched by the framework that contain Allpix objects (which should normally be all messages). Every of those messages contain a vector of objects, which are 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 having the class name of this object as name. Then for every combination of detector and message name a new branch is created in 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 that event did 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 about those separate messages is lost. It is also currently not possible to limit the data that is written to the file. If only a subset of the objects is needed than the rest of the data should be discarded afterwards.

Parameters

- `file_name` : Name of the data file (without the .root suffix) to create, relative to the output directory of the framework.

Usage

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

```
[ROOTObjectWriter]
```

7.12. 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. 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 the depth direction (normal to the pixel grid) from the implant side for a propagated charge to be taken into account.

Usage

For typical simulation purposes a *max_depth_distance* around 10um should be sufficient, leading to the following configuration:

```
[SimpleTransfer]
max_depth_distance = 10um
```

7.13. VisualizationGeant4

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

Description

Constructs a visualization 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 Geant4 version supports it.

The module allows for changing a variety of parameters to control the output visualization both for the different detector components and the display of the particle beam.

Parameters

- `mode` : Determines the mode of visualization. Options are **gui** which starts a Qt visualization window that contains the driver (as long as the chosen driver supports that), **terminal** starting both the visualization viewer together with 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 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. Moreover 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 the 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 that). Defaults to true, thus accumulating the events and only displaying the final result.
- `accumulate_time_step` : Time step to sleep every event to allow time to display it if the events are not accumulated. Only used if *accumulate* has been set to disabled. Default value is 100ms.
- `simple_view` : Determines if the visualization should be simplified, not displaying the pixel grid and other parts that are replicated multiple times. Default value is true. This parameter should normally not be changed as it will cause a very huge slowdown of the visualization for a sensor with a typical amount of pixels.
- `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 include **wireframe** and **surface**. By default the elements are displayed as solid surface.
- `transparency` : Default transparency percentage of all the detector elements, only used if the *view_style* is set to display solids. The default value is 0.4, giving a moderate amount of transparency.
- `display_trajectories` : Determines if the trajectories of the particles should be displayed. Defaults to enabled.

- `hidden_trajectories` : Determines if the trajectory paths 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 the trajectories are colored. First option is **generic**, which colors all trajectories in the same way. The next option is **charge** which bases the color on the charge. The last possible choice **particle**, colors the trajectory based on the type of the particle.
- `trajectories_color` : Color of the trajectories if *trajectories_color_mode* is set to **generic**. Default value is blue.
- `trajectories_color_positive` : Visualization color for positive 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 negative 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 if *trajectories_color_mode* is equal to **particle**. Refer to the Geant4 documentation for details about the ID's of the particles.
- `trajectories_draw_step` : Determines if the steps of the trajectories should be plotted. Defaults to enabled. Only used if the *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 the *trajectories_draw_step* is enabled.
- `trajectories_draw_step_color` : Color of the markers used to display a trajectory step. Default value is the color red. Only used if the *trajectories_draw_step* is enabled.
- `draw_hits` : Determines if the hits in the detector should be displayed. Defaults to false. Option is only useful if Geant4 hits are generated in a specific 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 with a wireframe viewing style with the same color for every particle and displaying the result after every event with 2s waiting time, is the following:

```
[VisualizationGeant4]
mode = "none"
view_style = "wireframe"
trajectories_color_mode = "generic"
accumulate = 0
accumulate_time_step = 2s
```

8. Module & Detector Development

8.1. Implementing a New Module

Before creating a module it is essential to read through 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. Thereafter the steps below should provide enough details for starting with a new module `ModuleName` (constantly replacing `ModuleName` with the real name of the new module):

1. Run the module initialization 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 toe get started together with a setup of the documentation in *README.md*.
2. Before continuing to implement the module it is recommended to check and update the introductory documention in *README.md*. Also the Doxygen documentation in *`ModuleName`.hpp* can be extended to give a basic description of the module.
3. Now the constructor, and possibly the `init`, `run` and/or `finalize` methods can be written, depending on what the new module needs.

After this, it is up to the developer to implement all the required functionality in the module. Keep considering however that at some point it may be beneficial to split up modules to support the modular design of Allpix^2. Various sources which may be primarily useful during the development of the 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 functionality other modules already provide and to look for similar modules which can help during development.
- The Doxygen (core) reference documentation included in the framework .
- The latest version of the source code of all the modules (and the core itself). Freely available to copy and modify under the MIT license at https://gitlab.cern.ch/simonspa/allpix-squared/tree/master.

Any module that may be useful for other people can be contributed back to the main repository. It is very much encouraged to send a merge-request at https://gitlab.cern.ch/simonspa/allpix-squared/merge_requests.

8.2. Adding a New Detector Model

Custom detector models can be easily added to the framework. Required information, before writing the model, is Section 5.2.1 describing the file format, Section 4.1.1 for information about the units used in Allpix^2 and the full Section 5.4 describing the geometry and detector models. In particular Section 5.4.3 explains all the parameters of the detector model. The default models shipped in *models* could serve as examples. To write your own module follow the steps below:

1. Create a new file with the internal 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.
3. Add all the required parameters and possibly other optional parameters 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 to either directly to the detector model file or the detector containing it (note that files in this path overwrite models with the same name in the default model folder).

Models can be contributed to the 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. Then the file should be added to the installation target in the *CMakeLists.txt* file in the *models* directory. Afterwards a merge-request can be created at https://gitlab.cern.ch/simonspa/allpix-squared/merge_requests.

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 given 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 detectors one should add a parameter `type` with the type of the detector model (as given in the detector configuration file by the `model` parameter).

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 specialize the location where the messages from the modules are send to and received from. By default both the input and the output of module defaults to the message without a name.

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.

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 a `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 specialized in the detector configuration file. This feature is available to, for example, use models with different sensor thicknesses. To specialize a detector model the key that should be changed in the standard detector model (like `sensor_thickness`) should be added as key to the section of the detector configuration (which is always required to already contain the position, orientation and the base model). 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.

How do I access the history of a particular object?

Many objects can include an internal link to related other objects (for example `getPropagatedCharges` in the `PixelCharge` object), containing the history of the object (thus the objects that were used to construct the current object). These referenced objects are stored as special ROOT pointers inside the object, which can only be accessed if the referenced object is available in memory. In Allpix^2 this

requirement can be automatically fullfilled by also binding the history object in a module, assuming the creating module actually saved the history with the object which is not strictly required. During analysis the tree holding the referenced object should be loaded and pointing to the same event entry as the object that request the reference to load it. If the referenced object can not be loaded an exception is required to be thrown by the retrieving method.

How do I access the Monte Carlo truth of a specific PixelHit?
The Monte Carlo truth is just part of the indirect history of a PixelHit. This means that the Monte-Carlo truth can be fetched as described in the question above. However take notice that there are multiple layers between a PixelHit and its MCParticles, which are the PixelCharge, PropagatedCharges and DepositedCharges. 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` convenience method is available which searches all the layers of the history and returns an exception if any of the in between steps is not available or not loaded.

Can I import an electric field from TCAD and use that 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 given in Section 4.4.

10. Additional Tools & Resources

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 5.6.3) to support internal ROOT and Geant4 classes. This allows for directly reading configuration parameters to those types, making the code in the modules both shorter and cleaner. Besides this, some other conversions functions are provided together with other useful utilities (for example to display a ROOT vector with units attached).

10.1.2. Runge-Kutta integrator

A fast Eigen-powered [3] Runge-Kutta integrator is provided as a tool to solve differential equations. The Runge-Kutta integrator is build genericly and supports multiple methods using different tableaus. It allows to integrate every system of equations in several steps with customizable timestep. This time step can also be updated during the integration depending on the error of the Runge-Kutta method (when a tableau with errors is used).

10.2. TCAD DF-ISE mesh converter

This code takes as input the .grd and .dat files from TCAD simulations. The .grd file contains the vertex coordinates (3D or 2D) of each mesh node and the .dat file contains the module 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 coinciding necessarily 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 4 closest, no-coplanar, neighbour vertex nodes that respective tetrahedron encloses the query point. For the neighbours search, the software uses the Octree implementation from the paper "Efficient Radius Neighbor Search in Three-dimensional Point Clouds" by J. Behley et al (see below).

The output .init file (with the same .grd and .dat prefixe) can be imported into allpix (see the User's Manual for details). 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 visalisation tool.

Usage

Example .grd and .dat files can be found in the data folder with the example_data prefix. To run the program the following 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. Default values are assumed for the options not used. These are -R = "bulk" -r = 1 um -r = 0.5 um -m = 10 um -c = std::numeric_limits::min() -x,y,z = 100 (option should be set using -x, -y and -z)

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

INIT files are read by specifying a file_name containing an .INIT file. 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 shown with the -h option. A default value of 100 is used for the binning, but this can be changed. 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 ploted 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.

Octree

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

Copyright 2015 Jens Behley, University of Bonn. This project is free software made available under the MIT License. For details see the OCTREE LICENSE file.

10.3. ROOT Analysis Macros

Collection of macros to analyze the data generated by the framework. Currently contains a single macro to convert the TTree of objects to a tree containing typical standard data users are interested in. This is useful for simple comparisons with other frameworks.

Comparison tree

Read all the required tree from the given file and bind their contents to the objects defined in the framework. Then creates an output tree and bind every branch to a simple arithmetic type. Continues to loop over all the events in the tree and converting the stored data of 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 using a very simple direct center of gravity calculation using the charges without any corrections.

To construct a comparison tree using this macro follow these steps: 1. Open root with the data file attached like `root -l /path/to/data.root` 2. Load the current library of objects with `.L path/to/libAllpixObjects.so` 3. Build the macro with `.L path/to/constructComparisonTree.C++` 4. Run the macro with `auto tree = constructComparisonTree(_file0, "name_of_dut")` 5. Open a new file with `auto file = new TFile("output.root", "RECREATE")` 6. Write the tree with `tree->Write()`

10.3.1. Remake project

Simple macro to show the possiblity to recreate source files for legacy objects stored in ROOT data files from older versions of the framework. Can be used when the corresponding dynamic library for that particular version is not accessible anymore. It is not possible to add the methods of the objects and it is therefore not easily possible to reconstruct the stored history (when available).

To recreate the project source files 1. Open root with the data file attached like `root -l /path/to/data.root` 2. Build the macro with `.L path/to/remakeProject.C++` 3. Recreate the source files using `remakeProject(_file0, "output_dir")`

11. Acknowledgments

- **Mathieu Benoit**, **John Idarraga**, **Samir Arfaoui** and all other contributors to the first version of AllPix, for their earlier work that inspired Allpix^2.
- **Neal Gauvin** for interesting discussion, his experiments with TGeo and his help implementing a visualization module.
- **Paul Schütze** for contributing his earlier work on simulating charge propagation and providing help on simulations with electric fields.
- **Marko Petric** for his help setting up several software tools like continous integration and automatic static-code analysis.
- **Salman Maqbool** for comments on the documentation and his help with porting the detector models from the original AllPix.
- **Moritz Kiehn** for his contributions to the code and helpful discussions on different matters concerning the simulation process.
- **Mateus Vicente** for his help in implementing a tool to interpolate electric fields from a TCAD mesh format to the grid used in Allpix^2.

We would also like to thank all others not listed here, that have contributed to the source code, provided input or suggested improvements.

A. Output of Example Simulation

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

```
(S) Welcome to Allpix^2 v0.3alpha5+14^g44961d2
(S) Initialized PRNG with system entropy seed 2756271465933902033
(S) Loaded 7 modules
(S) Initializing 14 module instantiations
(I) [I:DepositionGeant4] Using G4 physics list "QGSP_BERT"
(I) [I:DepositionGeant4] Not depositing charges in telescope1 because there is
   no listener for its output
(I) [I:ElectricFieldReader:telescope1] Setting linear electric field from 50V
   bias voltage and 50V depletion voltage
(I) [I:ElectricFieldReader:dut] Setting linear electric field from 50V bias
   voltage and 50V depletion voltage
(I) [I:ElectricFieldReader:telescope2] Setting linear electric field from 50V
   bias voltage and 50V depletion voltage
(S) Initialized 14 module instantiations
(S) Running event 1 of 5
(W) [R:DepositionGeant4] Dispatched message Message<allpix::MCParticle> from
   DepositionGeant4 has no receivers!
(I) [R:DepositionGeant4] Deposited 182856 charges in sensor of detector dut
(W) [R:DepositionGeant4] Dispatched message Message<allpix::MCParticle> from
   DepositionGeant4 has no receivers!
(W) [R:DepositionGeant4] Dispatched message Message<allpix::MCParticle> from
   DepositionGeant4 has no receivers!
(I) [R:DepositionGeant4] Deposited 191740 charges in sensor of detector
   telescope2
(W) [R:DepositionGeant4] Dispatched message Message<allpix::MCParticle> from
   DepositionGeant4 has no receivers!
(I) [R:GenericPropagation:dut] Propagated 91428 charges in 1829 steps in average
    time of 5.04796ns
(I) [R:GenericPropagation:telescope2] Propagated 95870 charges in 1918 steps in
   average time of 5.04183ns
(I) [R:SimpleTransfer:dut] Transferred 91428 charges to 4 pixels
(I) [R:SimpleTransfer:telescope2] Transferred 95870 charges to 4 pixels
(I) [R:DefaultDigitizer:dut] Digitized 4 pixel hits
(I) [R:DefaultDigitizer:telescope2] Digitized 4 pixel hits
(W) [R:DefaultDigitizer:telescope2] Dispatched message Message<allpix::PixelHit>
    from DefaultDigitizer:telescope2 has no receivers!
(S) Running event 2 of 5
(W) [R:DepositionGeant4] Dispatched message Message<allpix::MCParticle> from
   DepositionGeant4 has no receivers!
(I) [R:DepositionGeant4] Deposited 56190 charges in sensor of detector dut
(W) [R:DepositionGeant4] Dispatched message Message<allpix::MCParticle> from
   DepositionGeant4 has no receivers!
```

```
(W) [R:DepositionGeant4] Dispatched message Message<allpix::MCParticle> from
   DepositionGeant4 has no receivers!
(I) [R:DepositionGeant4] Deposited 328756 charges in sensor of detector
   telescope2
(W) [R:DepositionGeant4] Dispatched message Message<allpix::MCParticle> from
   DepositionGeant4 has no receivers!
(I) [R:GenericPropagation:dut] Propagated 28095 charges in 562 steps in average
   time of 5.04909ns
(I) [R:GenericPropagation:telescope2] Propagated 164378 charges in 3293 steps in
    average time of 4.37604ns
(I) [R:SimpleTransfer:dut] Transferred 28095 charges to 4 pixels
(I) [R:SimpleTransfer:telescope2] Transferred 164378 charges to 12 pixels
(I) [R:DefaultDigitizer:dut] Digitized 3 pixel hits
(I) [R:DefaultDigitizer:telescope2] Digitized 11 pixel hits
(W) [R:DefaultDigitizer:telescope2] Dispatched message Message<allpix::PixelHit>
    from DefaultDigitizer:telescope2 has no receivers!
(S) Running event 3 of 5
(W) [R:DepositionGeant4] Dispatched message Message<allpix::MCParticle> from
   DepositionGeant4 has no receivers!
(I) [R:DepositionGeant4] Deposited 53386 charges in sensor of detector dut
(W) [R:DepositionGeant4] Dispatched message Message<allpix::MCParticle> from
   DepositionGeant4 has no receivers!
(W) [R:DepositionGeant4] Dispatched message Message<allpix::MCParticle> from
   DepositionGeant4 has no receivers!
(I) [R:DepositionGeant4] Deposited 42496 charges in sensor of detector
   telescope2
(W) [R:DepositionGeant4] Dispatched message Message<allpix::MCParticle> from
   DepositionGeant4 has no receivers!
(I) [R:GenericPropagation:dut] Propagated 26693 charges in 534 steps in average
   time of 5.04742ns
(I) [R:GenericPropagation:telescope2] Propagated 21248 charges in 425 steps in
   average time of 5.03504ns
(I) [R:SimpleTransfer:dut] Transferred 26693 charges to 4 pixels
(I) [R:SimpleTransfer:telescope2] Transferred 21248 charges to 4 pixels
(I) [R:DefaultDigitizer:dut] Digitized 3 pixel hits
(I) [R:DefaultDigitizer:telescope2] Digitized 4 pixel hits
(W) [R:DefaultDigitizer:telescope2] Dispatched message Message<allpix::PixelHit>
    from DefaultDigitizer:telescope2 has no receivers!
(S) Running event 4 of 5
(W) [R:DepositionGeant4] Dispatched message Message<allpix::MCParticle> from
   DepositionGeant4 has no receivers!
(I) [R:DepositionGeant4] Deposited 43632 charges in sensor of detector dut
(W) [R:DepositionGeant4] Dispatched message Message<allpix::MCParticle> from
   DepositionGeant4 has no receivers!
```

```
(W) [R:DepositionGeant4] Dispatched message Message<allpix::MCParticle> from
    DepositionGeant4 has no receivers!
(I) [R:DepositionGeant4] Deposited 48516 charges in sensor of detector
    telescope2
(W) [R:DepositionGeant4] Dispatched message Message<allpix::MCParticle> from
    DepositionGeant4 has no receivers!
(I) [R:GenericPropagation:dut] Propagated 21816 charges in 437 steps in average
    time of 5.03719ns
(I) [R:GenericPropagation:telescope2] Propagated 24258 charges in 486 steps in
    average time of 5.06338ns
(I) [R:SimpleTransfer:dut] Transferred 21816 charges to 4 pixels
(I) [R:SimpleTransfer:telescope2] Transferred 24258 charges to 4 pixels
(I) [R:DefaultDigitizer:dut] Digitized 2 pixel hits
(I) [R:DefaultDigitizer:telescope2] Digitized 4 pixel hits
(W) [R:DefaultDigitizer:telescope2] Dispatched message Message<allpix::PixelHit>
     from DefaultDigitizer:telescope2 has no receivers!
(S) Running event 5 of 5
(W) [R:DepositionGeant4] Dispatched message Message<allpix::MCParticle> from
    DepositionGeant4 has no receivers!
(I) [R:DepositionGeant4] Deposited 40924 charges in sensor of detector dut
(W) [R:DepositionGeant4] Dispatched message Message<allpix::MCParticle> from
    DepositionGeant4 has no receivers!
(W) [R:DepositionGeant4] Dispatched message Message<allpix::MCParticle> from
    DepositionGeant4 has no receivers!
(I) [R:DepositionGeant4] Deposited 63184 charges in sensor of detector
    telescope2
(W) [R:DepositionGeant4] Dispatched message Message<allpix::MCParticle> from
    DepositionGeant4 has no receivers!
(I) [R:GenericPropagation:dut] Propagated 20462 charges in 410 steps in average
    time of 5.06975ns
(I) [R:GenericPropagation:telescope2] Propagated 31592 charges in 632 steps in
    average time of 5.03731ns
(I) [R:SimpleTransfer:dut] Transferred 20462 charges to 4 pixels
(I) [R:SimpleTransfer:telescope2] Transferred 31592 charges to 4 pixels
(I) [R:DefaultDigitizer:dut] Digitized 3 pixel hits
(I) [R:DefaultDigitizer:telescope2] Digitized 4 pixel hits
(W) [R:DefaultDigitizer:telescope2] Dispatched message Message<allpix::PixelHit>
     from DefaultDigitizer:telescope2 has no receivers!
(S) Finished run of 5 events
(I) [F:DepositionGeant4] Deposited total of 2103360 charges in 4 sensor(s) (
    average of 105168 per sensor for every event)
(I) [F:GenericPropagation:dut] Propagated total of 188494 charges in 3772 steps
    in average time of 5.04917ns
(I) [F:GenericPropagation:telescope2] Propagated total of 337346 charges in 6754
     steps in average time of 4.71792ns
```

```
(I) [F:SimpleTransfer:telescope1] Transferred total of 0 charges to 0 different
    pixels
(I) [F:SimpleTransfer:dut] Transferred total of 188494 charges to 4 different
    pixels
(I) [F:SimpleTransfer:telescope2] Transferred total of 337346 charges to 12
    different pixels
(I) [F:DefaultDigitizer:telescope1] Digitized 0 pixel hits in total
(I) [F:DefaultDigitizer:dut] Digitized 15 pixel hits in total
(I) [F:DefaultDigitizer:telescope2] Digitized 27 pixel hits in total
(I) [F:DetectorHistogrammer:dut] Plotted 15 hits in total, mean position is
    (125.667,125.6)
(S) Finalization completed
(S) Executed 14 instantiations in 2 seconds, spending 41% of time in slowest
    instantiation DepositionGeant4
(S) Average processing time is 401 ms/event, event generation at 2 Hz
```

References

[1] S. Agostinelli et al. “Geant4 - a simulation toolkit”. In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 506.3 (2003), pp. 250–303. ISSN: 0168-9002. DOI: https://doi.org/10.1016/S0168-9002(03)01368-8.

[2] *ROOT - An Object Oriented Data Analysis Framework*. Vol. 389. Sept. 1996, pp. 81–86.

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

[4] Mathieu Benoit et al. *The AllPix Simulation Framework*. Mar. 21, 2017. URL: https://twiki.cern.ch/twiki/bin/view/Main/AllPix.

[5] Mathieu Benoit, John Idarraga, and Samir Arfaoui. *AllPix. Generic simulation for pixel detectors*. URL: https://github.com/ALLPix/allpix.

[6] Daniel Hynds, Simon Spannagel, and Koen Wolters. *The Allpix^2 Project Issue Tracker*. July 27, 2017. URL: https://gitlab.cern.ch/simonspa/allpix-squared/issues.

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

[8] 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/.

[9] Daniel Hynds, Simon Spannagel, and Koen Wolters. *The Allpix^2 Project Repository*. Aug. 2, 2017. URL: https://gitlab.cern.ch/simonspa/allpix-squared/.

[10] S. Aplin et al. “LCIO: A persistency framework and event data model for HEP”. In: *Nuclear Science Symposium and Medical Imaging Conference (NSS/MIC), IEEE*. Anaheim, CA, Oct. 2012, pp. 2075–2079. DOI: 10.1109/NSSMIC.2012.6551478.

[11] X. Llopart et al. “Timepix, a 65k programmable pixel readout chip for arrival time, energy and/or photon counting measurements”. In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 581.1 (2007). VCI 2007, pp. 485–494. ISSN: 0168-9002. DOI: http://dx.doi.org/10.1016/j.nima.2007.08.079.

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

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

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

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

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

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

[18] Michael Kerrisk. *Linux Programmer's Manual. ld.so, ld-linux.so - dynamic linker/loader*. URL: http://man7.org/linux/man-pages/man8/ld.so.8.html.

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

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

[21] 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.