Documentation

Paul Gessinger
CERN

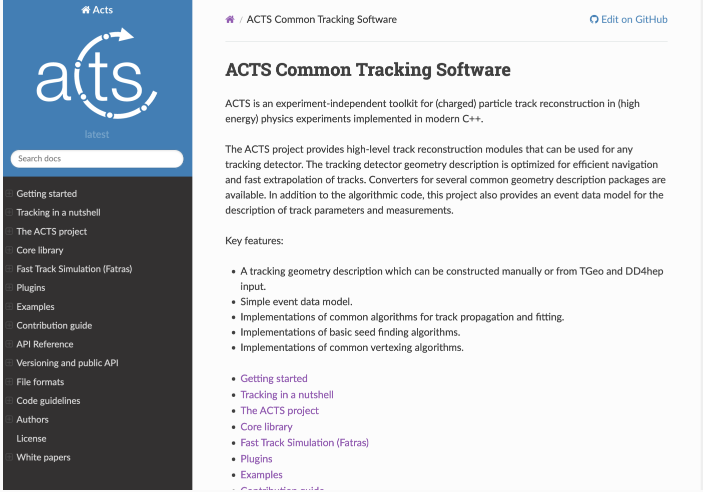
2023-11-08 - ACTS Workshop Orsay 2023

Outline

- Introduction to our documentation
- Tutorial: building the documentation
- Authoring for the documentation

Introduction

- Documentation in ACTS is built with Sphinx
- C++ level code-documentation is parsed using Doxygen
- Breathe bridges the two: C++ constructs can be pulled into Sphinx or cross-referenced
- Deployed to Read the Docs
  - Gives us easy multi-versioning
  - PR hook allows previewing documentation for changes



ACTS Common Tracking Software

ACTS is an experiment-independent toolkit for (charged) particle track reconstruction in (high energy) physics experiments implemented in modern C++.

The ACTS project provides high-level track reconstruction modules that can be used for any tracking detector. The tracking detector geometry description is optimized for efficient navigation and fast extrapolation of tracks. Converters for several common geometry description packages are available. In addition to the algorithmic code, this project also provides an event data model for the description of track parameters and measurements.

Key features:

- A tracking geometry description which can be constructed manually or from TGeo and DD4hep input.
- Simple event data model.
- Implementations of common algorithms for track propagation and fitting.
- Implementations of basic seed finding algorithms.
- Implementations of common vertexing algorithms.

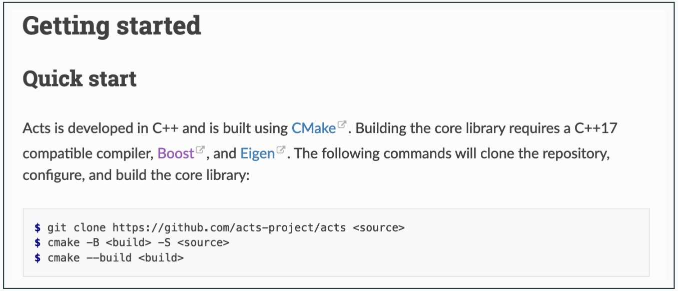
- Getting started
- Tracking in a nutshell
- The ACTS project
- Core library
- Fast Track Simulation (Fatras)
- Plugins
- Examples

Goal

- Both: high-level conceptual documentation and technical detailed documentation

Peek into the docs

- Getting started building the ACTS
- High-level tracking introduction
- Core library documentation
- Documentation of the Examples framework
- & How-to guides for Examples workflows
- Contribution guide

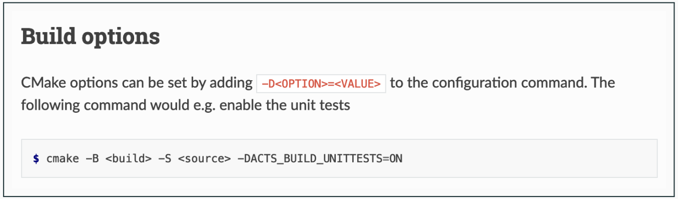


Getting started

Quick start

Acts is developed in C++ and is built using CMake. Building the core library requires a C++17 compatible compiler, Boost, and Eigen. The following commands will clone the repository, configure, and build the core library:

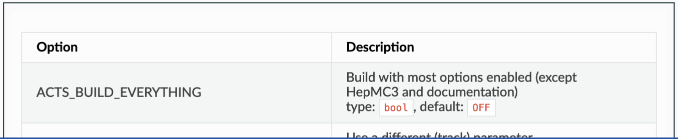
```
$ git clone https://github.com/acts-project/acts <source>
$ cmake -B <build> -S <source>
$ cmake --build <build>
```



Build options

CMake options can be set by adding `-D<OPTION>=<VALUE>` to the configuration command. The following command would e.g. enable the unit tests

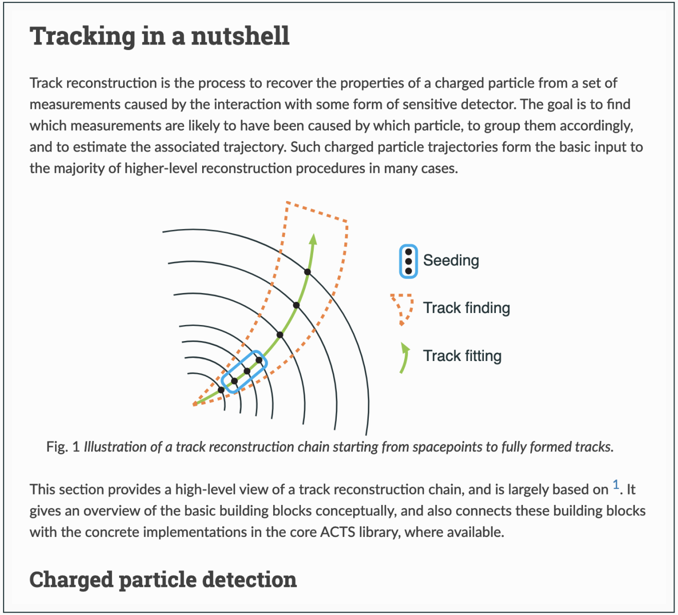
```
$ cmake -B <build> -S <source> -DACTS_BUILD_UNITTESTS=ON
```



Option	Description
ACTS_BUILD_EVERYTHING	Build with most options enabled (except HepMC3 and documentation) type: `bool`, default: `OFF`
	Use a different (track) parameter

Peek into the docs

- Getting started building the ACTS
- High-level tracking introduction
- Core library documentation
- Documentation of the Examples framework
- & How-to guides for Examples workflows
- Contribution guide



Tracking in a nutshell

Track reconstruction is the process to recover the properties of a charged particle from a set of measurements caused by the interaction with some form of sensitive detector. The goal is to find which measurements are likely to have been caused by which particle, to group them accordingly, and to estimate the associated trajectory. Such charged particle trajectories form the basic input to the majority of higher-level reconstruction procedures in many cases.

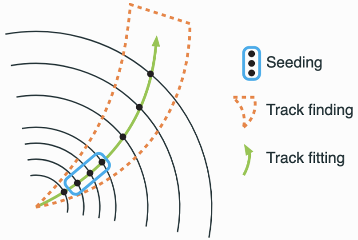


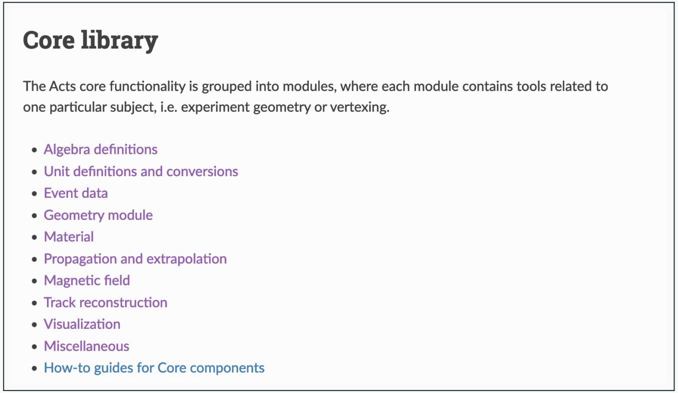
Fig. 1 *Illustration of a track reconstruction chain starting from spacepoints to fully formed tracks.*

This section provides a high-level view of a track reconstruction chain, and is largely based on [1]. It gives an overview of the basic building blocks conceptually, and also connects these building blocks with the concrete implementations in the core ACTS library, where available.

Charged particle detection

Peek into the docs

- Getting started building the ACTS
- High-level tracking introduction
- Core library documentation
- Documentation of the Examples framework
- & How-to guides for Examples workflows
- Contribution guide



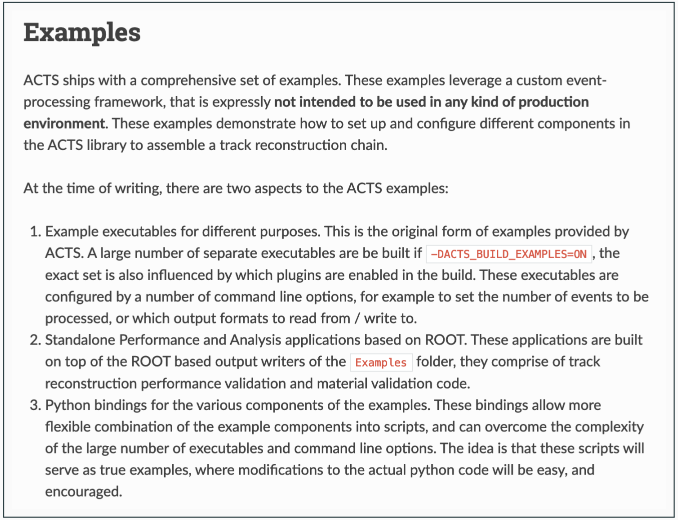
Core library

The Acts core functionality is grouped into modules, where each module contains tools related to one particular subject, i.e. experiment geometry or vertexing.

- Algebra definitions
- Unit definitions and conversions
- Event data
- Geometry module
- Material
- Propagation and extrapolation
- Magnetic field
- Track reconstruction
- Visualization
- Miscellaneous
- How-to guides for Core components

Peek into the docs

- Getting started building the ACTS
- High-level tracking introduction
- Core library documentation
- Documentation of the Examples framework
- & How-to guides for Examples workflows
- Contribution guide



Examples

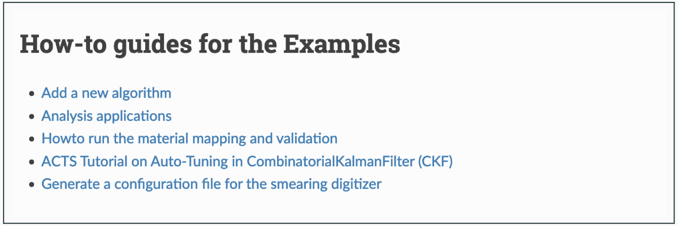
ACTS ships with a comprehensive set of examples. These examples leverage a custom event-processing framework, that is expressly **not intended to be used in any kind of production environment**. These examples demonstrate how to set up and configure different components in the ACTS library to assemble a track reconstruction chain.

At the time of writing, there are two aspects to the ACTS examples:

1. Example executables for different purposes. This is the original form of examples provided by ACTS. A large number of separate executables are be built if `-DACTS_BUILD_EXAMPLES=ON`, the exact set is also influenced by which plugins are enabled in the build. These executables are configured by a number of command line options, for example to set the number of events to be processed, or which output formats to read from / write to.
2. Standalone Performance and Analysis applications based on ROOT. These applications are built on top of the ROOT based output writers of the `Examples` folder, they comprise of track reconstruction performance validation and material validation code.
3. Python bindings for the various components of the examples. These bindings allow more flexible combination of the example components into scripts, and can overcome the complexity of the large number of executables and command line options. The idea is that these scripts will serve as true examples, where modifications to the actual python code will be easy, and encouraged.

Peek into the docs

- Getting started building the ACTS
- High-level tracking introduction
- Core library documentation
- Documentation of the Examples framework
- & How-to guides for Examples workflows
- Contribution guide

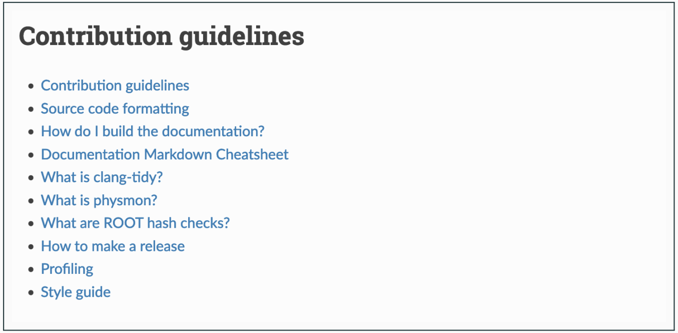


How-to guides for the Examples

- Add a new algorithm
- Analysis applications
- Howto run the material mapping and validation
- ACTS Tutorial on Auto-Tuning in CombinatorialKalmanFilter (CKF)
- Generate a configuration file for the smearing digitizer

Peek into the docs

- Getting started building the ACTS
- High-level tracking introduction
- Core library documentation
- Documentation of the Examples framework
- & How-to guides for Examples workflows
- Contribution guide

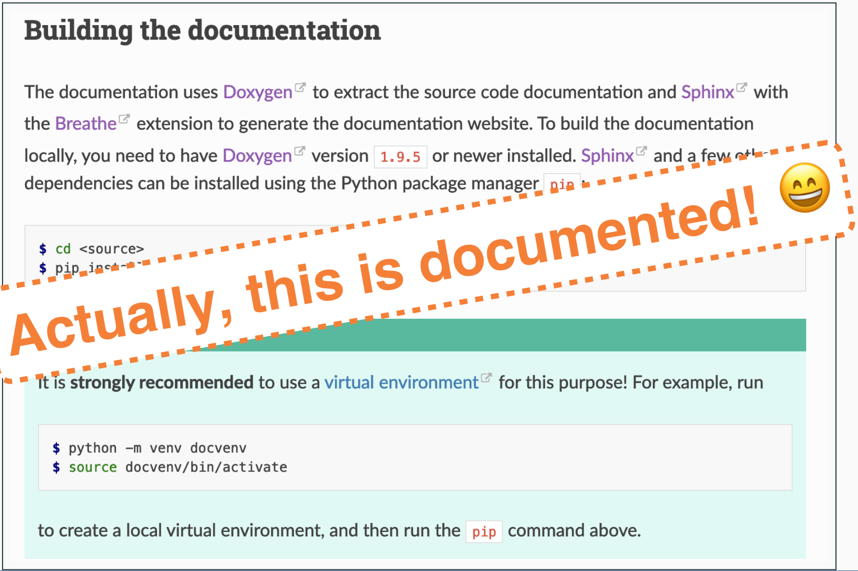


Contribution guidelines

- Contribution guidelines
- Source code formatting
- How do I build the documentation?
- Documentation Markdown Cheatsheet
- What is clang-tidy?
- What is physmon?
- What are ROOT hash checks?
- How to make a release
- Profiling
- Style guide

Building the documentation

Building the documentation



Building the documentation

The documentation uses Doxygen to extract the source code documentation and Sphinx with the Breathe extension to generate the documentation website. To build the documentation locally, you need to have Doxygen version `1.9.5` or newer installed. Sphinx and a few oth dependencies can be installed using the Python package manager `pip`:

```
$ cd <source>
$ pip inst
```

It is **strongly recommended** to use a virtual environment for this purpose! For example, run

```
$ python -m venv docvenv
$ source docvenv/bin/activate
```

to create a local virtual environment, and then run the `pip` command above.

Building the documentation

Prerequisites

- Doxygen $\geq$ `1.9.5`
- Sphinx and dependencies using `pip`:

```
$ pip install -r docs/requirements.txt
```

- 🚨 **Always use a virtual environment!** 🚨

1. Configure the project

```
$ cmake -B <build> -S <source> -DACTS_BUILD_DOCS=on
```

2. Run the `docs` command

```
$ cmake --build <build> --target docs
```

- Tip: to view the documentation, you can start a local web-server in python using

```
$ python -m http.server -d <source>/docs/_build/html
```

Authoring documentation

- Sphinx natively uses `reStructuredText` for authoring
  - Very powerful but cumbersome
- Our documentation supports Markdown for authoring using `MyST`
- **Recommendation is to use Markdown whenever possible**
- Documentation is a hierarchy using nested `toctree`s
  - New pages need to be linked in `toctree` to show up
  - `core/core` is an example of a nested `toctree`

Example: `docs/index.rst`

```
.. toctree::
:maxdepth: 1

getting_started
tracking
acts_project
core/core
Fast Track Simulation (Fatras) <fatras/fatras>
plugins/plugins
examples/examples

Contribution guide <contribution/contribution>

api/api

versioning
formats/formats
codeguide
authors
license
```

rST syntax

Authoring (`MyST` specific syntax)

```
(segmentation)=
:::{figure} /figures/tracking/segmentation.svg
:align: center
:width: 400px
Illustration of a one-dimensional (a) and
a two-dimensional segmentation (b) of a
silicon sensor.
:::
...
Silicon sensors are usually segmented in
one dimension (*strips*) or in two dimensions
(*pixels*) (see {numref}`segmentation`).
```

- `:::` starts a *directive*, `figure` is the directive name
  - `:width:` are parameters to the directive
  - Test is given a *content* to the directive
- `(segmentation)=` adds an explicit anchor

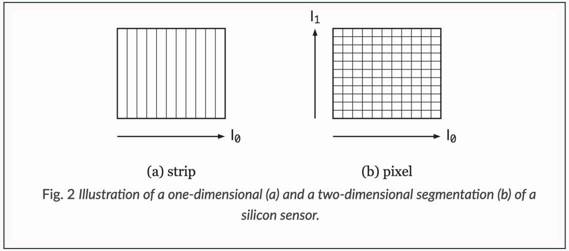
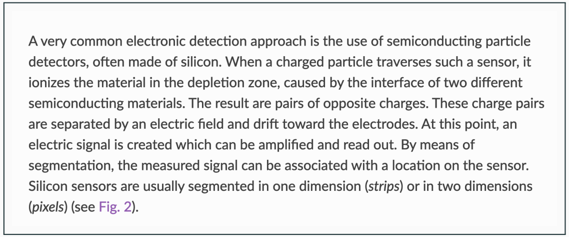


Fig. 2 *Illustration of a one-dimensional (a) and a two-dimensional segmentation (b) of a silicon sensor.*



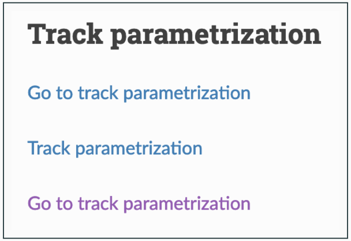
A very common electronic detection approach is the use of semiconducting particle detectors, often made of silicon. When a charged particle traverses such a sensor, it ionizes the material in the depletion zone, caused by the interface of two different semiconducting materials. The result are pairs of opposite charges. These charge pairs are separated by an electric field and drift toward the electrodes. At this point, an electric signal is created which can be amplified and read out. By means of segmentation, the measured signal can be associated with a location on the sensor. Silicon sensors are usually segmented in one dimension (*strips*) or in two dimensions (*pixels*) (see Fig. 2).

- `{numref}`segmentation`` is a *role*
  - This adds a numbered cross-reference!

Authoring: anchors

- Anchors also work on headlines

```
(track-par-section)=
## Track parametrization
...
[Go to track parametrization](#track-par-section) % <- explicit anchor, explicit title
[](#track-par-section) % <- explicit anchor, automatic title
[Go to track parametrization](#track-parametrization) % <- implicit anchor
```



Track parametrization

Go to track parametrization

Track parametrization

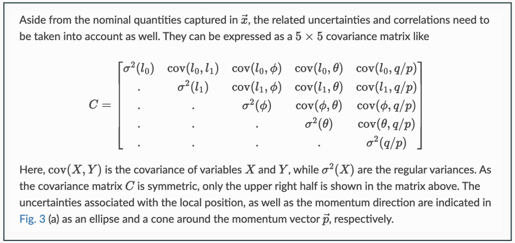
Go to track parametrization

- **Recommendation: use explicit anchors whenever possible**
- Can use explicit title or automatic title

Authoring equations

```
\begin{equation*}
  C =
  \begin{bmatrix}
  \sigma^2(l_0)& \text{cov}(l_0,l_1)
    & \text{cov}(l_0, \phi)
    & \text{cov}(l_0, \theta) & \text{cov}(l_0, q/p) \\
  . & \sigma^2(l_1) & \text{cov}(l_1, \phi)
    & \text{cov}(l_1, \theta) & \text{cov}(l_1, q/p) \\
  . & . &  \sigma^2(\phi) & \text{cov}(\phi,\theta)
    & \text{cov}(\phi, q/p) \\
  . & . & . & \sigma^2(\theta) & \text{cov}(\theta, q/p) \\
  . & . & . & . & \sigma^2(q/p)
  \end{bmatrix}
\end{equation*}
```

✅ This works in Markdown!



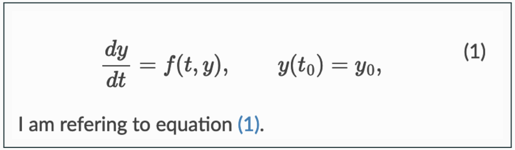
Aside from the nominal quantities captured in $\vec{x}$, the related uncertainties and correlations need to be taken into account as well. They can be expressed as a 5×5 covariance matrix like

$$C = \begin{bmatrix} \sigma^2(l_0) & \text{cov}(l_0,l_1) & \text{cov}(l_0,\phi) & \text{cov}(l_0,\theta) & \text{cov}(l_0,q/p) \\ . & \sigma^2(l_1) & \text{cov}(l_1,\phi) & \text{cov}(l_1,\theta) & \text{cov}(l_1,q/p) \\ . & . & \sigma^2(\phi) & \text{cov}(\phi,\theta) & \text{cov}(\phi,q/p) \\ . & . & . & \sigma^2(\theta) & \text{cov}(\theta,q/p) \\ . & . & . & . & \sigma^2(q/p) \end{bmatrix}$$

Here, $\text{cov}(X,Y)$ is the covariance of variables X and Y, while $\sigma^2(X)$ are the regular variances. As the covariance matrix C is symmetric, only the upper right half is shown in the matrix above. The uncertainties associated with the local position, as well as the momentum direction are indicated in Fig. 3 (a) as an ellipse and a cone around the momentum vector $\vec{p}$, respectively.

- Simple equations largely work, sophisticated / specialized packages not supported

```
$$
\frac{dy}{dt} = f(t,y), \qquad y(t_0) = y_0,
$$ (diffeq)
I am refering to equation [](#diffeq).
```

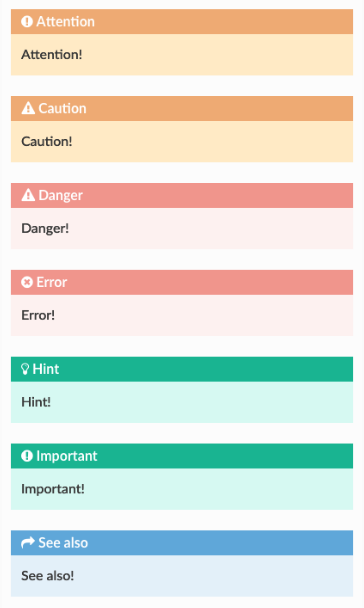


$$\frac{dy}{dt} = f(t,y), \qquad y(t_0) = y_0, \tag{1}$$

I am refering to equation (1).

Admonitions

```
:::{attention}
Attention!
:::

:::{caution}
Caution!
:::

:::{danger}
Danger!
:::

:::{error}
Error!
:::

:::{hint}
Hint!
:::

:::{important}
Important!
:::
```



Attention

Attention!

Caution

Caution!

Danger

Danger!

Error

Error!

Hint

Hint!

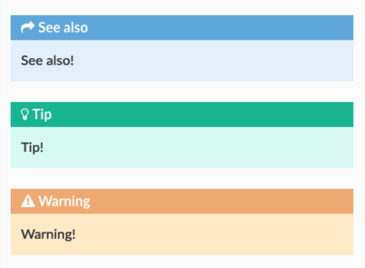
Important

Important!

See also

See also!

```
:::{seealso}
See also!
:::

:::{tip}
Tip!
:::

:::{warning}
Warning!
:::
```



See also

See also!

Tip

Tip!

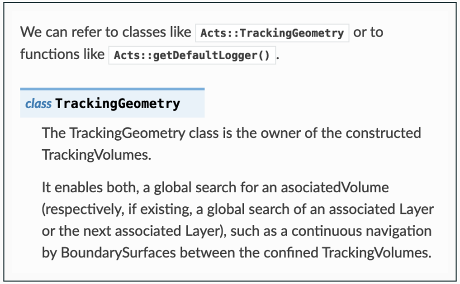
Warning

Warning!

Authoring: code documentation

```
We can refer to classes like {class}`Acts::TrackingGeometry`
or to functions like {func}`Acts::getDefaultLogger`.

:::{doxygenclass} Acts::TrackingGeometry
:::
:::{doxygenfunction} Acts::getDefaultLogger
:::
```

- Roles `{class}`xxx`` and `{func}`yyy`` can be used to link to the auto-generated documentation
  - Need to be inserted *somewhere*. Classes, structs, typedefs and enums should be auto-generated
  - Otherwise you need to put `{doxygenXXX}` directivs somewhere in the documentation
- Consider adding the *main* part of the doc in the comment, and pull it into Sphinx
  - Makes documentation available in code, easier to keep in sync!

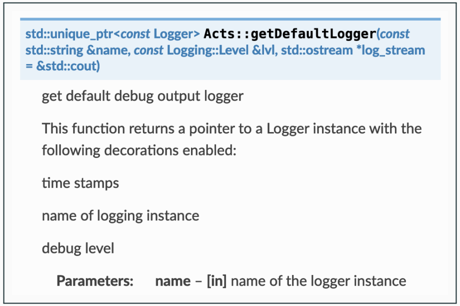


We can refer to classes like `Acts::TrackingGeometry` or to functions like `Acts::getDefaultLogger()`.

class **TrackingGeometry**

The TrackingGeometry class is the owner of the constructed TrackingVolumes.

It enables both, a global search for an asociatedVolume (respectively, if existing, a global search of an associated Layer or the next associated Layer), such as a continuous navigation by BoundarySurfaces between the confined TrackingVolumes.



std::unique_ptr<*const* Logger> **Acts::getDefaultLogger**(*const* std::string &name, *const* Logging::Level &lvl, std::ostream *log_stream = &std::cout)

get default debug output logger

This function returns a pointer to a Logger instance with the following decorations enabled:

time stamps

name of logging instance

debug level

Parameters: **name** – **[in]** name of the logger instance

Conclusion

- Have comprehensive set of tools for documentation:
  1. Doxygen comments for inline documentation of code
  2. Sphinx-generated documentation for higher-level descriptions
  - Mechanism to pull 1. into 2.

What do we want in the documentation?

- Low-level description of interfaces
- Reasons behind decisions in classes, methods and functions
- Meaning of configuration members
- Design and implementation decisions of algorithms
- Derivations / motivation of equations
- In-depth algorithm descriptions
  - Either directly in the documentation, or as a white paper (see Felix' talk next)
- How-to guides on how to use core components in experiments
- How-to guides on running the Examples