`Preliminary draft 16:06 29 October 2019`

29 October 2019

`acts-developers@cern.ch`

Concepts, design and Implementation of the A Common Tracking Software (Acts) project

Acts Developers
CERN, CH-1211 Geneva, Switzerland

Keywords: `Bending Magnet, path length, pole face angle, beam trajectory, survey, GEODE`

Summary

Contents

1 Introduction

The reconstruction of charged particle trajectories in the tracking detectors of the LHC experiments is the most CPU intensive task of the entire event reconstruction. While it is less pronounced at low particle multiplicities, the combinatorial character of track finding becomes more and more computationally expensive the more charged particles are present within the detector volume. At the time of writing, the LHC operates at a rate of 25 ns bunch crossing rate and yields on average more than 40 instantaneous proton-proton collisions (also called pile-up μ) per bunch crossing. The prospect of exceeding the LHC design specifications of $< \mu >$ 23 before Run-2, had caused the experiments to lead quite extensive software optimization campaigns [?] during the first long shutdown period LS-1 of the LHC. The LS-1 work included software optimization, simplification of the event data and algorithmic adjustments. The general data flow and strategy of the track reconstruction as described in [?], however, remained unchanged. While during Run-3 of the LHC the event complexity will stay approximately the same or at least only moderately increases, the upcoming upgrade of the LHC to the high luminosity machine HL-LHC will dramatically change this picture. The event pile-up will increase to $< \mu >$ 200, which will push the event reconstruction by a factor of 5 at least out of the available computing budget. Looking forward into the future, certain scenarios of the currently studied LHC successor option FCC-hh may lead to the pile-up regime of 1000 instantaneous collisions per bunch crossing [?]. One way to improve the event reconstruction throughput is to more optimally exploit the available hardware. In current projections of future computing hardware, this usually implies a shift to more concurrent design patterns. While the experiments frameworks are currently updated to support multithreaded reconstruction data flow (see, e.g. [?], [?] and [?]), also the algorithmic tools and concepts will need substantial updates, may it just be to make the software components intrinsically thread-safe.

2 The ATLAS track reconstruction software

The ATLAS track reconstruction software [?] was designed and largely written following the publication of the ATLAS Reconstruction Task Force report (RTF Report) [?] before the start up of the ATLAS experiment in the years 2003 to 2008. It was partly based on two prior reconstruction programs `bib:xKalman` and `bib:iPatRec` and served very successful data taking and Monte Carlo (MC) simulation campaigns to date. It is characterised by a vanishing low failure rate (typical MC or data taking campaigns process up to 10^9 events) and a very high reconstruction efficiency. During the first long shut down (LS-1) a dedicated clean-out and optimisation campaign was performed which led to a five-fold speed-up of the ATLAS Inner Detector track reconstruction for expected Run-2 conditions without any loss of reconstruction quality. This evolution comprised the move of all algorithmic code from the `CLHEP` [?] linear algebra component to the `Eigen` [?] library, and a first attempt to minimize heap memory allocations.

2.1 Review

Based on the component model of the underlying `Gaudi` [?] framework, the ATLAS track reconstruction software implemented a plug-and-play like

3 The `Acts` project

3.1 General design guidelines

3.1.1 Public interfaces

3.1.2 Configuration

A light weight convention has been chosen for the configuration of `Acts` components. Configurable components are requested to provide a nested configuration `struct` which is instantiated and used for the constructing the object. This struct can then be interfaced with the the experiment's configuration system, e.g. the Gaudi `declareProperty()` macro with its associated `python` binding.

```
#include <string>
/// @class A component, a real work horse
class WorkHorse {
    public:
        /// Nested configuration struct
        struct Config {
            std::string name = "Beauty";
        };

    /// Constructor
    /// @param cfg The coniguration object
    WorkHorse(const Config& cfg) = default;

    private:
        ///!< Instance of the configuration
        Config m_cfg;
};
```

The creation of two modules would read as:

```
    WorkHorse::Config myCasperCfg{"Casper"};
    WorkHorse myCasper(myCasperCfg);
```

3.1.3 Parallelism

`Acts` strives to provide a fully multithreaded track resconstructed toolkit, hence dedicated care has to be taken in order to allow for inter and intra-event parallel execution of modules. A trivial step towards ensuring a save parallel execution is to apply `const` correctness throughout the software stack. However, in several applications of track reconstruction, intermediate caching of information is required to optimise the execution speed. An example for this is e.g. the magnetic field, where the current magnetic field cell (which is likely also the magnetic fill cell of the subsequent call) is cached. Internal caching is evidently violating `const` correctness. To allow caching during execution, any `Acts` component that requires the caching of runtime information has to provide a nested `State struct` that is instantiated before call and provided by the caller to ensure the cache object stays protected within its call context.

```
#include <string>
/// @class A component, a real work horse
class WorkHorse {
    public:
        /// Nested configuration struct
        struct Config {
            std::string name = "Beauty";
            double maxHoursOut = 1000_h;
        };

        /// Nested State object
        struct State {
            double hoursOnField = 0.;
        };

        /// Method that requires caching
        /// @param state cache recording the hours
        void workOnField(State& state) const;
};
```

Note that the method that performs the operation is marked `const`, while certainly in the operation a caching for this operation can be done. If e.g. a parallel execution is performed, the multiple `State` objects guarantee that each operation respects its on own caching constraints.

3.1.4 Contextual data

Apart from processing caches there is also contextual data that may prevent a seamless parallel execution. Detector conditions, such as calibration constants, noise, detector status and alignment and/or even the magnetic field strength may be subject to change during a reconstruction job. Although in general those changes may be quite infrequent special processing streams with stringent event selections can easily evoke a system where - when executing events in parallel - multiple detector alignment and calibration conditions need to be present and available in memory.

In `Acts` this is solved with so-called *Context* objects that have to be provided for any contextual call. Three different flavors exist:

- `Acts::CalibrationContext`: this object holds the contextual information for measurement calibrations
- `Acts::GeometryContext`: this object holds the contextual information for all geometry related parameters
- `Acts::MagneticFieldContext`: this object holds the contextual information for magnetic field related parameters

The implementation of the context objects can be chosen freely, in `Acts` they are practically only provided through the call chain such that they can be resolved at the right moment. In general, retrieval of the contextual information (e.g. changed alignment or calibration) has to happen in this design at maximum once per event (or even less)[1].

The context objects guarantee that an operation/data retrievals performed within the right context, e.g. the request for a surface transform has to be done by providing the correct alignment context

```
auto surfaceTransform = surface.transfrom(geoContext);
```

3.1.5 Plugins

`Acts` is designed with minmial dependency on other libraries, the core library itself only requires the `Eigen` math library as a minimal depenency. However, it is certainly useful to allow interfacing or using other external libraries, such as e.g. the `ROOT` [?] framework, `DD4hep` [?] or `Geant4` [?].

3.2 Unit and integration testing

4 Repository components

4.1 The Surface and Geometry component

The `Acts::Surface` class

The key geometric component of the `Acts` package is the `Acts::Surface` class. Surfaces build up the higher level geometry objects that are described by the `Acts::Layer` and `Acts::TrackingVolume` classes, but also combine event data model (and track parameterisation) with the detector geometry.

The `Acts::TrackingVolume` class

4.1.1 Geometry and Navigation

In the following, the concepts of the `Acts` geometry and navigation structure is described with a simple one layer detector[2]. Figure 4 shows a conceptional single layer detector (for simplicity only in the x-y plane) that is built from certain detector components. For the use of full simulation, this detector has to be described with a geometry modeller, in the current case this has been done with the use of the `DD4hep` [?] toolkit, which leads to illustration displayed in Fig. ??. Evidently, when describing a real detector in a 3D modeller, some simplifications have to be done as in general not all shapes are possible to be described efficiently in such a model. However, a certain level of detail is required to reflect the actual detector geometry and material distribution. Especially for the full simulation, usually carried out with the

[1]Often various database requests are needed for establishing the detector conditions, hence this operation should be done as infrequent as possible

[2]These steps can be followed in the framework repository

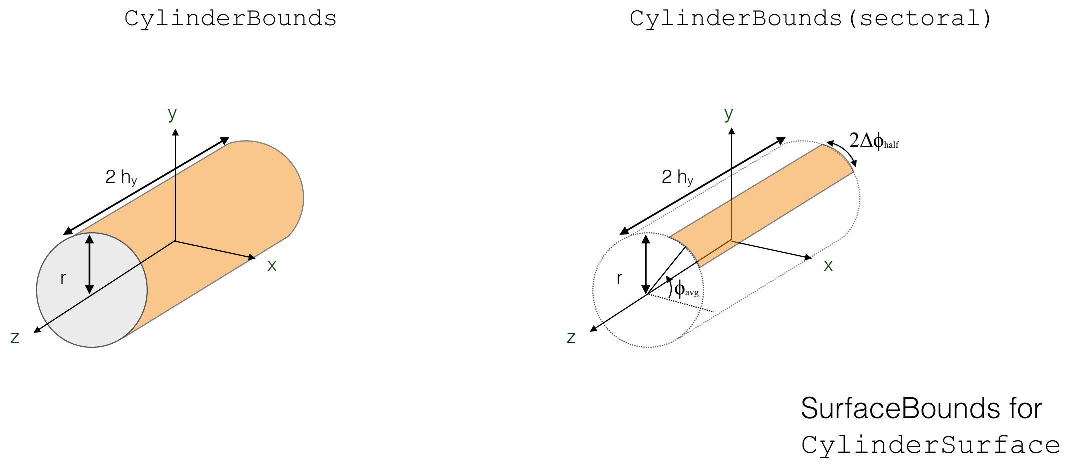

Figure 1: Examples of cylindrical surfaces in `Acts` with different cylinder boundaries.
.

`Geant4` [?] toolkit, a detailed description is needed in order to guarantee good agreement between simulated and taken data. `DD4hep` offers a convenient translation of the underlying detector geometry (which is described with the `TGeo` component of the `ROOT` library) into a `Geant4` description, which allows to run a full simulation directly on `DD4hep` input. The resulting tomography of the detector (in this context performed with non interacting virtual particles, called *Geantinos* is displayed in Fig. 6.

For track reconstruction, on the other hand, the detailed description used for full simulation is in general not necessary and would lead to unwanted CPU overhead when trying to resolve the navigation through such a complicated geometry. This is, because in track reconstruction, the interaction of the particle with the detector material is only taken into account in a stochastic manner. The position, orientation and conditions of the sensitive detector elements, however, are to be described with full detail as they are needed to determine the track measurements to the greatest detail. In `Acts` this is guaranteed by requiring that the representing sensitive sensors, tubes or other detection devices are implemented as an extension of the `Acts::DetectorElementBase` base class, which provides a surface representation the rest of the `Acts` application. If a `Acts::Layer` contains several sensitive detectors, such as, e.g. a collection of sensitive silicon wafers, they out to be grouped in a so-called `Acts::SurfaceArray` which gives the layer an internal structure, see Fig 7. As long as track reconstruction is concerned, only the sensitive detector elements and the material description of the layer structure is relevant. When sub structure of a layer is present, the layer has in general a thickness different from 0, and the layer boundaries will be guarded by so called *approach surfaces*[3]. In case of a layer without sub structure, such as e.g. a simple representation of a beam pipe as a cylinder, the layer thickness is forced to be zero and the approach surfaces are omitted, see Fig 8.

The navigation between layers is straight forward: `Acts::Layer` objects are interlinked at

[3]For many classiscal layer setups, the `Acts` toolkit creates the layer sub structure automatically by parsing the sensitive sensor dimensions and positions.

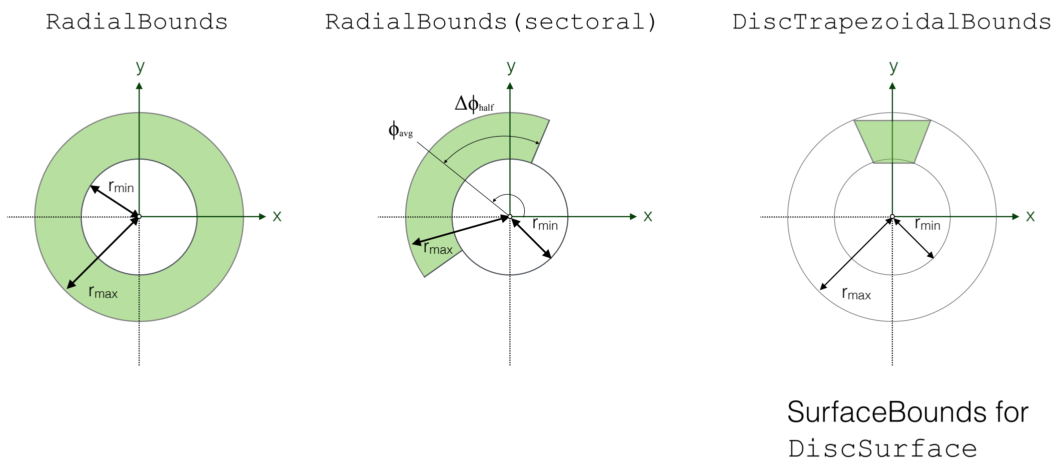

Figure 2: Examples of disk-like surfaces in `Acts` with different disk boundaries.
.

construction time and hence approximate layers are registered to the current `Acts::Layer` object. When navigation towards a layer, the layer provides in case of sub structure the approach surface candidate, or - in absence of sub structure - the `Acts::Surface` representation, respectively. From the intersection with the approach surface, the binned `Acts::SurfaceArray` object provides compatible candidate sub structure surfaces (sensitive or non sensitive) to the navigation module. If layers are in different volumes, the *glueing* mechanism of the geometry building guarantees a common boundary surface between the adjunct volumes, displayed in Figure 9. Finally, Fig. 10 shows the resulting navigation and propagation steps of the `Acts::Propagator` with its internal navigation through this detector setup.

4.2 Event Data Model

The track reconstruction event data model (EDM) has to describe track parameterisation, a measurement expression, track objects and vertices.

4.2.1 Track parameterisation

The set of track parameters describes the parametrisation of a trajectory with respect to a surface. Hence, different flavors of track parameters exist depending on the different surface types. `Acts` provides a default track parameterisation, which is a combination of local and global parameters, the first two parameters l_0 and l_1 (the ones defined by the surface reference frame) are expressed locally, while the remaining parameters ϕ, θ, q/p and t (time) are in global parameter space.

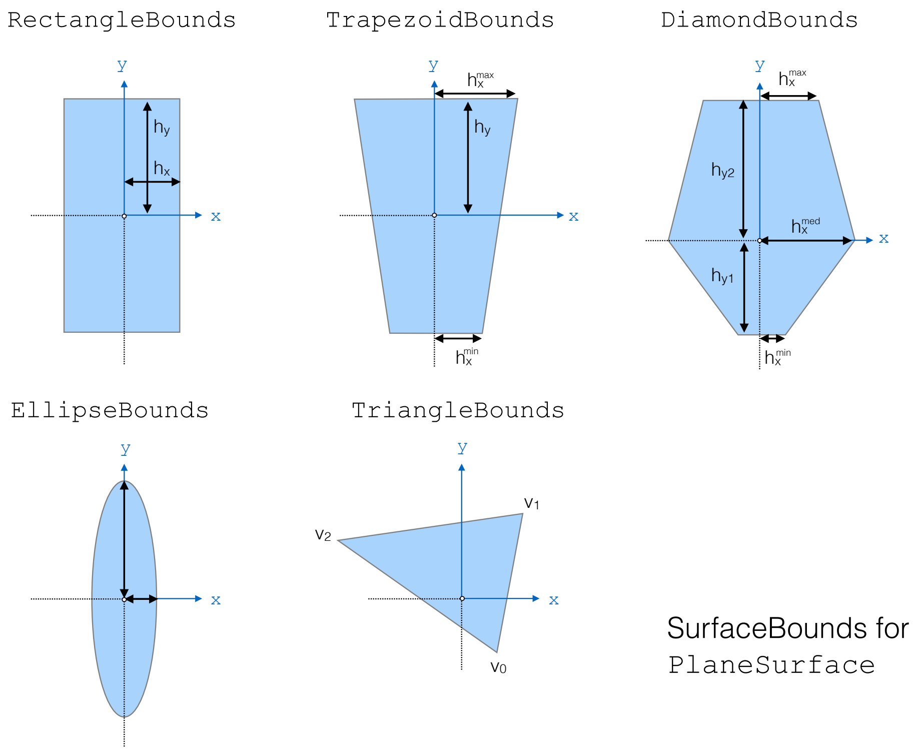

Figure 3: Examples of planar surfaces in `Acts` with different planar boundaries.

.

4.2.2 Measurement description

Following the `Acts` philosophy only minimal requirements are put onto the a correct and complete measurement description. In order to minimise unnecessary operations measurements are best expressed in the same coordinate system as the track parameterisation. By doing so, the so-called *measurement mapping function* that relate a track parameterisation to the actual measurement are reduced to simple projection matrix operations.

5 Performance examples

6 Conclusion

References

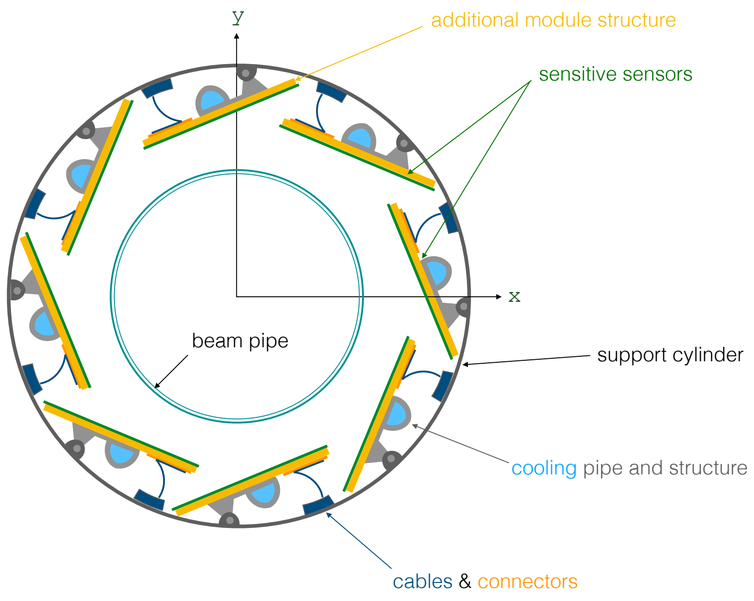

Figure 4: A drawing of a conceptional single layer detector with a beam pipe and a certain sub structure.

.

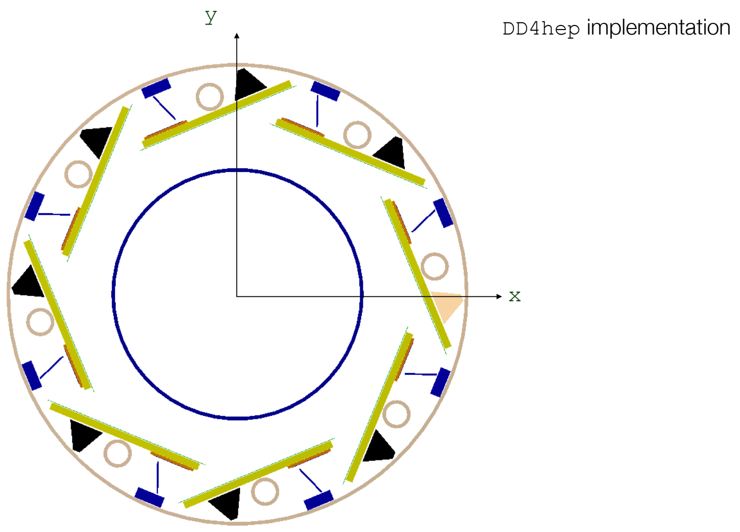

Figure 5: `DD4hep` representation of the conceptual single layer detector. A certain level of simplification has to be applied to allow a description in a 3D geometry modelling toolkit.

.

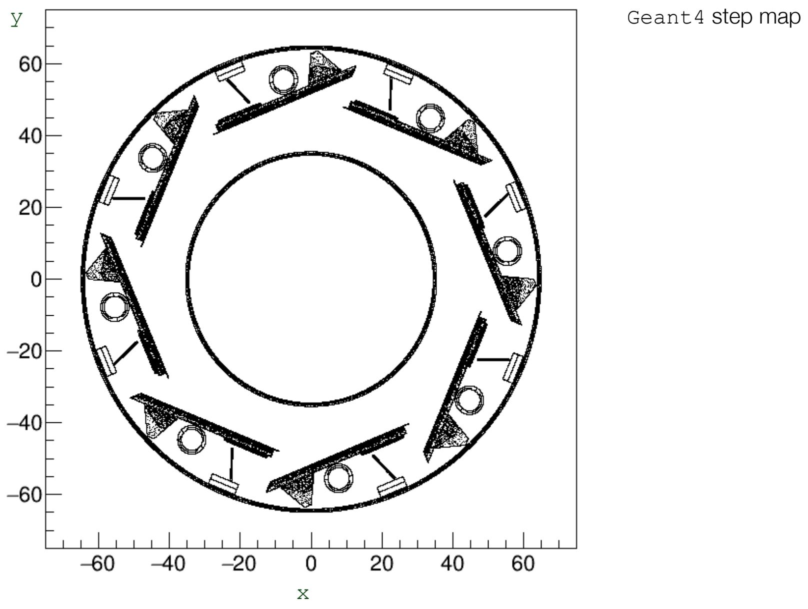

Figure 6: **Geant4** tomography of the conceptual single layer detector. The single structure given by the **DD4hep** description can be clearly seen in this step recording.

.

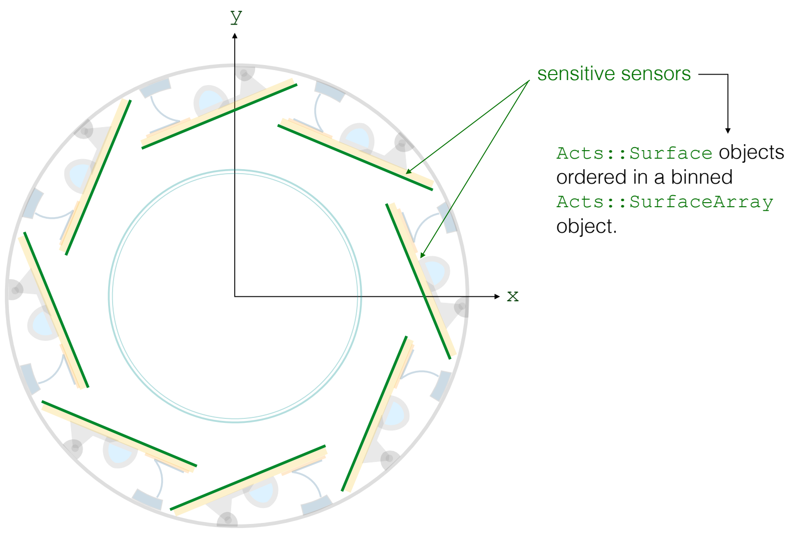
y
x
sensitive sensors
Acts::Surface objects ordered in a binned Acts::SurfaceArray object.

Figure 7: Sensitive detector elements representing the sensors in the conceptual single layer detector.

.

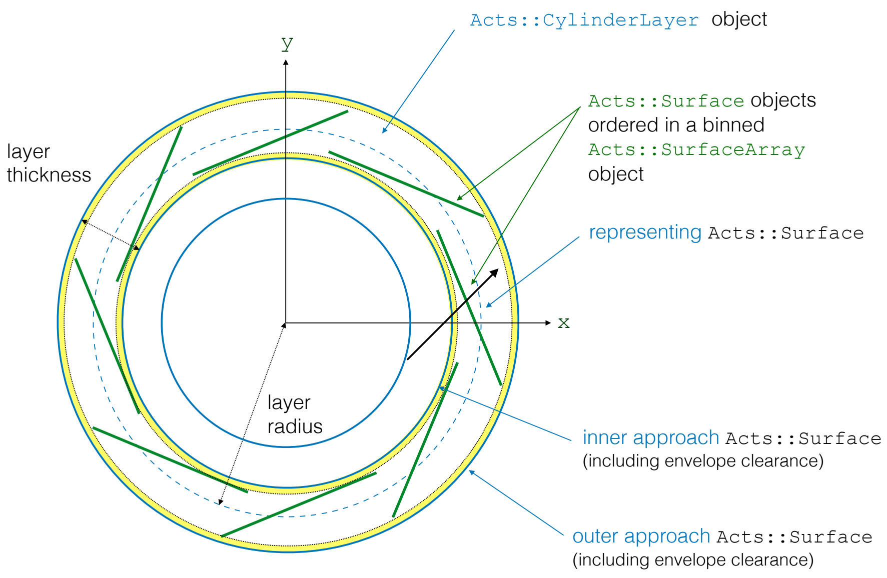
Acts::CylinderLayer object
y
Acts::Surface objects
ordered in a binned
Acts::SurfaceArray
object
layer
thickness
representing Acts::Surface
x
layer
radius
inner approach Acts::Surface
(including envelope clearance)
outer approach Acts::Surface
(including envelope clearance)

Figure 8: Sensitive detector elements representing the sensors in the conceptual single layer detector, leading to a layer with sub structure and non trivial thickness and approach surfaces. The beam pipe representation is without sub structure and hence neither a layer thickness nor approach surfaces are present.

.

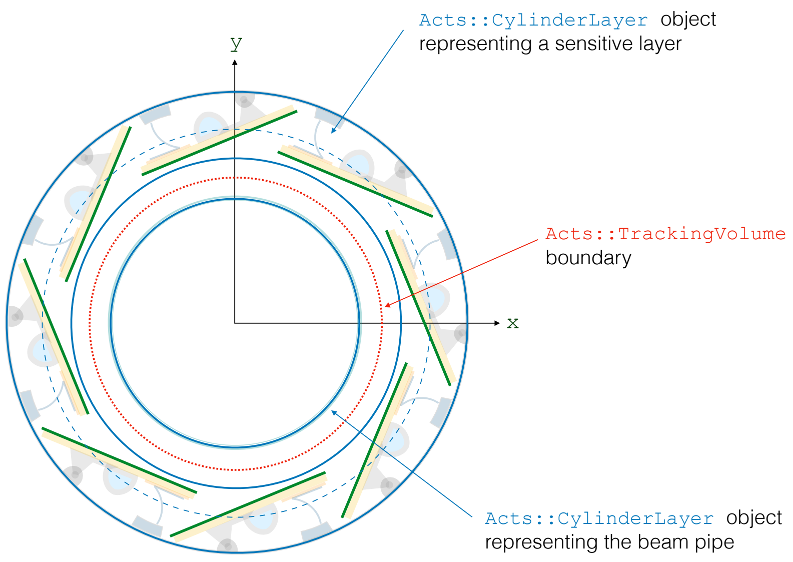
Acts::CylinderLayer object
representing a sensitive layer
y
Acts::TrackingVolume
boundary
x
Acts::CylinderLayer object
representing the beam pipe

Figure 9: Relevant `Acts` surface descriptions for the navigation through the conceptual single layer detector. The boundary surface between the beam pipe volume and the detection layer volume is automatically created during the geometry construction and volume glueing process.

.

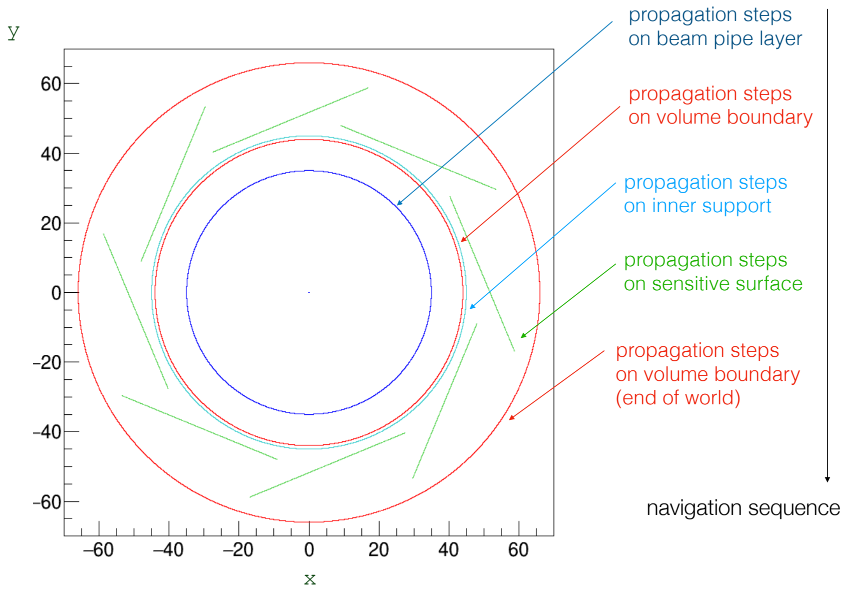

Figure 10: Resulting propagation steps on sensitive, passive, boundary and approach surfaces in the conceptual single layer detector. The navigation sequence from inside out can be followed.

.