Development and Optimization of ACTS Tracking Software for High Luminosity Collider Experiments

Dissertation

zur Erlangung des mathematisch-naturwissenschaftlichen Doktorgrades
„Doctor rerum naturalium“
der Georg-August-Universität Göttingen

im Promotionsprogramm Physik
der Georg-August University School of Science (GAUSS)

vorgelegt von

Joana Niermann

aus Bielefeld

Göttingen, 2024

Tag der mündlichen Prüfung: 09.08.2024

Referenz: II.Physik-UniGö-Diss-2024/05

Development and Optimization of ACTS Tracking Software for High Luminosity Collider Experiments

Abstract

With the dawn of the high luminosity era at the LHC, an unprecedented amount of data will be collected and processed at the ATLAS experiment. This will result in a drastic increase of hit combinatorics during track reconstruction with the ATLAS ITk detector, which will replace the current Inner Detector. New algorithms and methods are investigated to efficiently process the incoming data. One possibility is the deployment of hardware accelerators that provide massive parallelism, like general-purpose computing on GPUs.

In this thesis, the DETRAY library will be presented, for which a GPU-friendly tracking geometry and navigation was developed. Within the ACTS (*A Common Tracking Software*) project, which is a detector-agnostic toolkit of tracking algorithms written in modern `C++`, a dedicated R&D effort was launched to investigate the adaptation of the ACTS tracking chain to GPUs. The final GPU tracking demonstrator will provide a realistic setup of all steps of track reconstruction, from clusterisation to ambiguity resolution, and thus allow an in-depth study of both physics and compute performance of GPU-based tracking within the ATLAS experiment.

A crucial ingredient to be able to run track reconstruction is to ensure accurate and efficient modelling of the detector geometry and its material. This is done in ACTS by the *tracking geometry*, which is a purely surface based representation of the detector with a dedicated material mapping step. The current implementation has been found in previous studies to have several shortcomings concerning an adaptation to GPU computing, like its use of virtual function calls to describe different geometrical shapes for the detector surfaces, or its use of vector-of-vector data containers that rely on dynamic memory allocations.

With the DETRAY library, a tracking geometry will be made available that solves these problems by using a combination of static polymorphism and an index-based data management on global, flat data containers in memory. Using the VECMEM library for data management, the DETRAY detector can be read in from data files exported from existing tracking geometries in ACTS on the CPU and subsequently be copied to the GPU memory system to run device-side track reconstruction. The geometry description can be provided in full detail compared to ACTS, including the access to material maps.

The DETRAY tracking geometry and the track parameter navigation have been validated in a constant magnetic field against a numeric approach using the Newton-Raphson algorithm, enhanced with bisection steps, on several detector geometries, among them the current ITk tracking geometry.

This work has been sponsored by the Wolfgang Gentner Programme of the German Federal Ministry of Education and Research (grant no. 13E18CHA)

Contents

CHAPTER 1

Introduction

1.1. The Standard Model of Particle Physics

The Standard Model (SM) contains the theoretical foundation of the field of particle physics and describes the dynamics and interactions of the known subatomic particles, which can be produced and studied at laboratories like, for example, CERN. It is formulated in the language of Quantum Mechanics and Quantum Field Theory, which have been applied to great success to describe the behaviour of elementary particles [1–4].

Even though the Standard Model has been highly successful in describing the physics behind observations made at particle physics experiments, its predictions continue to be constantly checked against ever more precise measurements. One such quantity that can be measured is, for example, the rate $\mathrm{d}N/\mathrm{d}t$ at which a certain process appears in an experiment. The probability of the process occurring in a particle interaction can be calculated in the Standard Model using the *cross-section* σ. Together with knowledge about the *Luminosity* $\mathcal{L}$ of the accelerator, a machine dependent quantity that models the number of interactions in a collison event, the cross section and the measured rate can be related:

$$\frac{\mathrm{d}N}{\mathrm{d}t} = \sigma\mathcal{L}\,. \tag{1.1}$$

Small deviations in measurable parameters such as the cross section can give a hint that unknown effects or even previously undiscovered particles may be present that were not taken into account in the theoretical calculations. Measurements of this kind are consequently among the searches for new physics.

The most well known force described in the Standard Model is the electromagnetic force [5–10], however, the weak and the strong forces play pivotal roles as well. The strong force [11–16] is, for instance, responsible for keeping the atomic nucleus together against the driving electromagnetic force of the positively charged protons. It is also a major contributor to the processes that occur in proton-proton collisions at the LHC. The weak force, on the other hand, does not result in bound states, like the atom or its nucleus, but facilitates many subatomic interactions. An example are radioactive decays like the β-decay, where a neutron n decays into a proton p and an electron e with its associated anti-neutrino $\bar{\nu_e}$ [17–19]:

$$n \rightarrow p + e + \bar{\nu_e}\,. \tag{1.2}$$

The particles in the Standard Model are characterized by a set of *Quantum Numbers*, which quantify their properties, such as their spin or charge. The charges determine, whether a particle participates in one of the three fundamental forces described by the Standard Model. For example, the electromagnetic force acts upon particles that carry an electric charge ($\pm e$), while particles that have a colour (*red*, *green*, *blue*) will interact via the strong force.

In the Standard Model, the fundamental forces are mediated between particles through the exchange of another special kind of particle, the gauge bosons. As the name "boson" implies, these carry an integer spin quantum number and arise as quantizations of the underlying Quantum Fields that describe the respective force. The strength with which the gauge bosons couple to the charge carried by particles is parametrised in the *coupling constants*, which are not predicted by the Standard Model, but are rather tuning parameters that need to be determined experimentally. The momentum scale at which the interaction happens, as well as the mass of the gauge bosons will additionally influence the coupling strength [20].

The electromagnetic force is mediated by the well known photon γ, which is the quantum of the electromagnetic field and thus corresponds to the phenomenon of light. Gauge bosons can also carry charge and couple to themselves, such as the eight massless *gluons* g [21–23] of the strong force, which come with both a colour and an anti-colour. Similarly, the gauge bosons of the weak force, the $W^{\pm}$ and Z^0 bosons [24–27], also participate in weak interactions themselves, such as in the W-boson pair production [28–30].

One of the great successes of the Standard Model, was the unification of the electromagnetic and the weak interaction in the *electroweak unification* by Glashow, Weinberg and Salam [31–33]. Here, electromagnetic and weak processes can be described by a single theory with a unified charge.

Another important step in the development of the Standard Model was the formulation of *Electroweak Symmetry Breaking* [34–39] and the subsequent discovery of the Higgs boson [40, 41]. The spontaneous symmetry breaking in the electroweak force is a way to describe the appearance of mass terms in the corresponding Lagrangian density and thus provides a way to explain the fact that the gauge bosons of the weak force, $W^{\pm}$ and Z^0, have mass. It can also be used to motivate the masses of fermions via Yukawa couplings [11, 32], which arise between the fermion and the Higgs field.

Apart from the gauge bosons and the Higgs particle, all other fundamental particles in the Standard Model are *fermions*, which means, they carry non-integer spin of 1/2. Charged Leptons l, for example, are elementary particles that come in three generations and in association with a neutrino ν_l [19, 43]. The lepton generations are identical in quantum numbers, but differ most notably in particle mass and *flavour*. The charged leptons in the Standard Model are the electron e, the muon μ and the tau-lepton τ [44–49], with the corresponding neutrinos ν_e, ν_μ and ν_τ [43, 50]. The neutrinos carry neither electric charge nor colour and thus solely interact via the weak force. The weak gauge bosons couple to the left-handed isospin doublets and, in case of the Z bosons, also right-handed singlets:

$$\begin{pmatrix} \nu_e \\ e \end{pmatrix}_L, \quad \begin{pmatrix} \nu_\mu \\ \mu \end{pmatrix}_L, \quad \begin{pmatrix} \nu_\tau \\ \tau \end{pmatrix}_L. \tag{1.3}$$

Similar to the leptons, *quarks* [51–55] also come in three generations and are divided into up-type quarks with an electric charge of $+2/3$ and down-type quarks with a charge of $-1/3$, depending on the third component of their weak isospin vector [56, 57]:

$$\begin{pmatrix} u \\ d' \end{pmatrix}_L, \quad \begin{pmatrix} c \\ s' \end{pmatrix}_L, \quad \begin{pmatrix} t \\ b' \end{pmatrix}_L. \tag{1.4}$$

The prime on the down-type quarks denotes the weak eigenstates, which differ from the mass eigenstates that propagate through time. The relation between weak and mass eigenstates is determined by the 3×3 unitary *Cabibbo-Kobayashi-Maskawa* (CKM) matrix [58, 59]. From the phase difference in the weak and mass eigenstates follow flavour oscillations and a mechanism for CP-violation. CP-violation describes the effect where a process appears different under simultaneous transformation of parity (P) and charge (C, meaning the transformation between a particle and its anti-particle) [60, 61]. A similar mechanism exists for the neutrinos, described by the *Pontecorvo-Maki-Nakagawa-Sakata* (PMNS) matrix [62, 63], which results in neutrino oscillations [64, 65].

Quarks carry colour and are almost uniquely confined into composite particles, called Hadrons, which are colour-neutral. This can be achieved by grouping three quarks of different color in a *Baryon* (qqq) or a quark and an anti-quark in a *Meson* ($q\bar{q}$) [15, 56]. Mesons that have neutral electric charge are a superposition of quark-anti-quark pairs of different flavour. The only quark that is not confined to hadrons, is the top quark t [66, 67], which decays on a shorter time scale than the hadronization process can occur.

A number of questions are left open by the Standard Model. For one, the Standard Model only describes three of the four fundamental forces in nature. It does not describe either gravity or dark matter and is therefore an incomplete theory. But there are other issues as well, like the amount of CP-violation that is predicted, which cannot explain the overwhelming dominance of matter over anti-matter that is observed in the universe [68]. Furthermore, neutrinos are assumed to be massless in the Standard Model, while the measurements of neutrino oscillations show that neutrinos should have a small mass. Although a hierarchy of the masses of the different neutrinos can be inferred, the specific ordering and scale of the neutrino masses is unknown.

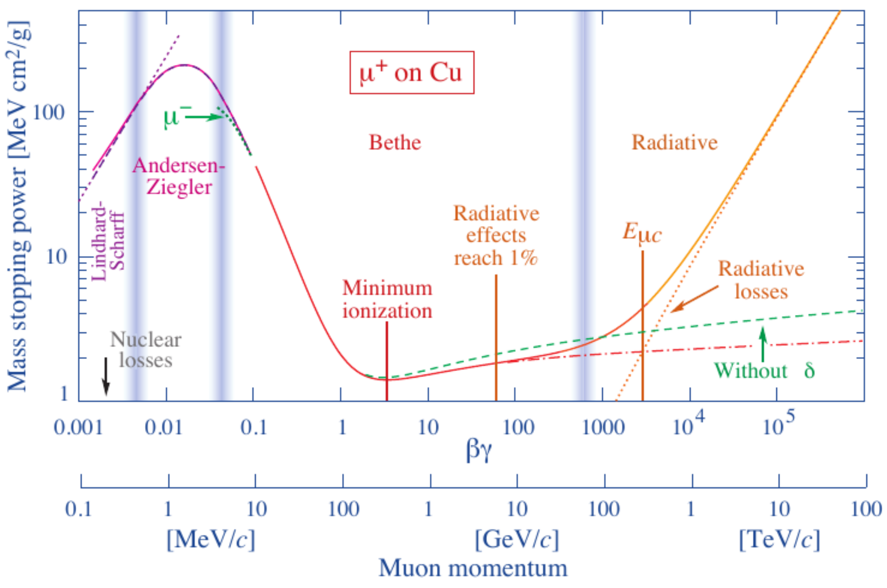


Figure 1.1.: Different regions of mass stopping power against particle velocity [42].

1.2. Interactions of Charged Particles with Matter

The detection of high energy particles is based on their interaction with the matter of particle detectors. The signals that are deposited by charged particles can be collected, converted to digital signals and processed for following reconstruction steps. In the context of particle track reconstruction, however, interactions with matter also pose a complication and introduce noise into the process. Most notably, energy loss by ionisation or Bremsstrahlung and multiple scattering will influence the particle trajectory through the detector and have to be taken into account accurately.

1.2.1. Energy Loss through Ionisation

When a particle traverses the detector material, it will engage in scattering interactions with the electrons of the atoms the material is made of. This will result in an energy loss of the particle and an excitation and ionisation of the material. Occasionally, an electron will receive enough energy to engage in ionisation itself. The mean mass stopping power of a material, that is the mean energy loss per distance x in units of $\mathrm{g}^{-1}\,\mathrm{cm}$ through the material $\langle -\mathrm{d}E/\mathrm{d}x \rangle$, can be calculated according to the empirical formula of Bethe [69–71]:

$$\left\langle -\frac{\mathrm{d}E}{\mathrm{d}x} \right\rangle = 4\pi N_A r_e^2 m_e c^2 \, \frac{Z}{A} \, \frac{z^2}{\beta^2} \left[\frac{1}{2} \ln \left(\frac{2 m_e c^2 \beta^2 \gamma^2 W_{max}}{I^2} \right) - \beta^2 - \frac{\delta\left(\beta\gamma\right)}{2} \right] . \tag{1.5}$$

The mean mass stopping power depends on the particle charge number z and its velocity $\beta = v/c$ in units of the speed of light c, the atomic number Z and the mass number A

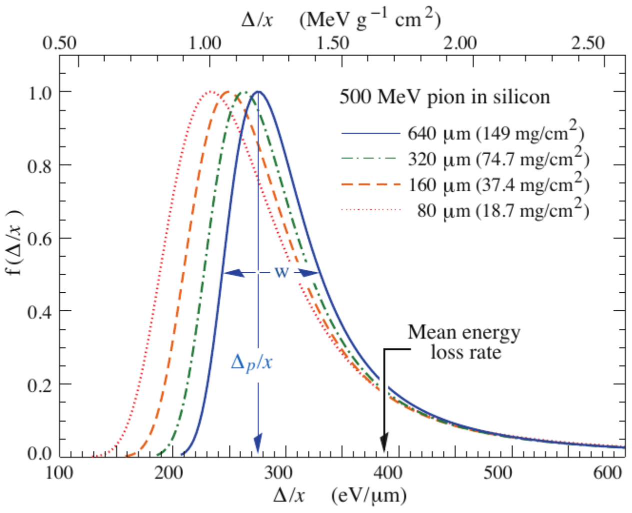


Figure 1.2.: Straggling function for 500 MeV pions, normalised to Δ_p/x [42]

of the material, as well as the Avogadro constant N_A and the classical electron radius and mass, r_e and m_e. Furthermore, the mean ionisation potential of the electrons in the atoms of the material [72, 73], as well as the maximum energy transfer W_{max} that can occur to one of those electrons have to be determined for the material. The term $\delta(\beta\gamma)$ parametrises the polarisation response of the material.

The Bethe formula is accurate for heavy particles in the range of $0.1 \lesssim \beta\gamma \lesssim 1000$, where $\gamma = \sqrt{1-\beta^2}$ follows from special relativity. For lower particle velocities, additional corrections have to be applied [74–76] and for higher velocities, radiative effects have to be taken into account [42]. A particle with a velocity $\beta\gamma \approx 3$, which lies in or close to the minimum of the Bethe function, is called a *minimum ionising particle* (MIP). A plot of the mean mass stopping power over the different regions of $\beta\gamma$ can be seen in Figure 1.1. For electrons and positions, Eq. 1.5 needs to be modified respectively, due to the similarity to the electrons of the atoms of the material [69, 73, 77].

However, the mean energy deposition for a given path length as described by Eq. 1.6 is an estimator over many particle crossings and differs from the most probable energy deposition of a single particle due to rare scattering events with large energy transfers. The distribution of the energy depositions Δ in material of a given thickness can be calculated according to the Landau-Vavilov distribution [78, 79], which is asymmetric and features a tail towards larger energy depositions, as shown in Figure 1.2. The most probable value of the energy deposition for a single particle, Δ_p, can be calculated according to Ref. [80]:

$$\Delta_p = \xi \left[\ln\left(\frac{2mc^2\beta^2\gamma^2}{I}\right) + \ln\left(\frac{\xi}{I}\right) + j - \beta^2 - \delta\left(\beta\gamma\right) \right] , \tag{1.6}$$

with $\xi = (2\pi N_A r_e^2 m_e c^2)(Z/A)z^2(x/\beta^2)$ and $j = 0.2000$ [80]. The full width at half maximum of the distribution is given by $w = 4\xi$ for absorbers of medium thickness.

1.2.2. Particle Showers

While ionisation is the main driver of energy loss for heavy particles, the formation of cascades has to be taken into account [42, 81–83] for electrons and photons, where the emission of Bremsstrahlung and pair production leads to the creation of several generations of secondary particles. High momentum electrons lose energy in matter predominantly by the emission of Bremsstrahlung until they fall below a critical energy threshold and ionisation becomes the main mode of energy loss. The radiation length X_0 of a material can be defined as the distance over which an electron loses $1/e$ of its energy due to Bremsstrahlung. The photons generated in this process can subsequently engage in production of e^+e^--pairs, which will themselves continue to emit Bremsstrahlung.

A similar concept can be formulated for hadrons and hadronic interactions in matter, where the mean interaction length λ is defined as characteristic scale for the energy loss. In addition to Coulomb interactions, hadrons can also engage in inelastic scattering and energy loss via the strong force with the nucleons (p and n) that constitute the atomic nucleus. This leads to the generation of secondary particles, which to a significant degree are neutral pions π^0. The dominant decay mode of the neutral pion is to two photons via the process $\pi^0 \to \gamma\gamma$ [42], which afterwards can give rise to an electromagnetic shower component.

1.2.3. Multiple Scattering

A significant source of noise during track reconstruction stems from random direction changes that happen as a result of elastic Coulomb scattering events between the incident particle and the atoms of the detector material. The theory that models multiple scattering was formulated by G. Molière and later revisited by H. A. Bethe [84, 85]. For practical use, several approximations to the scattering angle distribution of Molière exist, which reach varying degrees of accuracy [86–88].

These approximations allow to calculate the expected scattering angle θ^{rms} projected to a plane after a particle with momentum p has traversed a material of thickness x and radiation length X_0. Since the repeated scattering through small angles is a stochastic process, the expected scattering angle is modelled using a Gaussian distribution according to the law of large numbers. For a particle with charge z [42], the central 98 % of the distribution can be described by a formula given by Lynch and Dahl in Ref. [88], which is an improvement of the Highland formula [87]:

$$\theta^{rms} = 13.6\,\mathrm{MeV}\, z\, \frac{\sqrt{x/X_0}}{\beta c p} \left[1 + 0.088 \cdot \log_{10}\left(\frac{x\, z^2}{X_0\, \beta^2}\right)\right] . \tag{1.7}$$

When encountering material mixtures, it is advised to average the radiation length of the respective material components rather than adding the expected scattering angles in quadrature [42, 88]. This can be achieved by summing the radiation lengths according to the fraction of the material component w_i:

$$1/X_0 = \sum_i w_i/X_i\, . \tag{1.8}$$

CHAPTER 2

The ATLAS Experiment

The ATLAS Experiment [89] at the *Large Hadron Collider* (LHC) is a general purpose particle physics experiment situated at the CERN laboratory in Geneva, Switzerland, together with three other main experiments, ALICE [90], CMS [91] and LHCb [92], which is shown in Figure 2.1. In 2012, ATLAS together with the CMS experiment announced the discovery of the Higgs boson [40, 41]. The physics goals at the ATLAS experiment range from precision measurements, for example, on the properties of the Higgs boson, to searches for physics beyond the Standard Model, like the nature of dark matter [93].

2.1. The Large Hadron Collider

The *Large Hadron Collider* (LHC) [95] is the successor of the *Large Electron-Positron Collider* (LEP) [96], which operated from 1989 to 2000 and collided electrons and positrons with a center-of-mass energy of up to $\sqrt{s} = 209\,\text{GeV}$. The LHC was subsequently installed in the same tunnel with a circumference of 26.7 km and at about 100 m mean depth. In order to minimise synchrotron radiation, the LHC has been designed as a proton-proton collider, with a targeted center-of-mass energy of $\sqrt{s} = 14\,\text{TeV}$. The two proton beams need their own magnetic fields and for most of the distance separate beam pipes to keep them circulating in opposite directions. With this setup, it is also possible to accelerate and collide heavy ions at the LHC.

The beams consist of separate bunches of protons, which contain about 10^{11} particles each. At the interaction points, where the detectors of the different experiments are located, the beams are brought to collision at a frequency of 40 MHz. The instantaneous luminosity of the LHC can be formulated for Gaussian beam profiles as [95]:

$$\mathcal{L} = \frac{N_b^2 n_b f_{rev} \gamma}{4\pi \epsilon_n \beta^*} F\,, \tag{2.1}$$

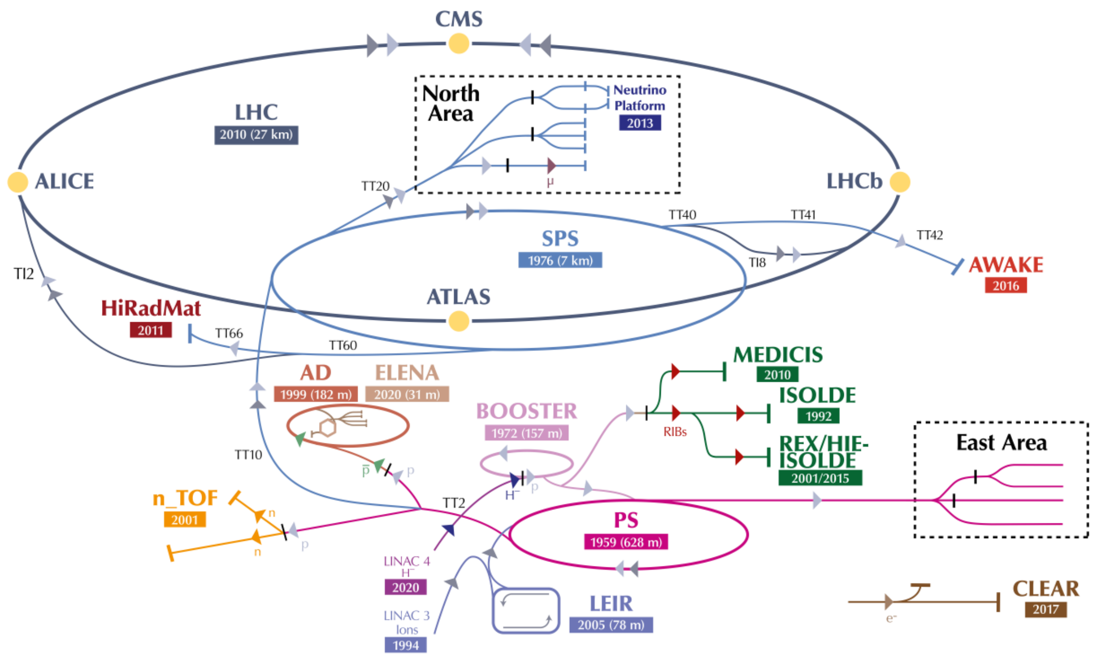


Figure 2.1.: Schematic view of the accelerator complex at CERN, including the preceding acceleration steps and the experiments. Adapted from Ref. [94].

with the revolution frequency f_{rev} of the bunches, the number of protons per bunch N_b, the number of bunches n_b and a geometric parameter F that models the beam size and the effect of the crossing angle of the bunches. The peak design luminosity was set to be $\mathcal{L} = 10^{34}\,\text{cm}^{-2}s^{-1}$. As was discussed earlier, the instantaneous luminosity of an accelerator is an important quantity for the analysis of particle interactions by the experiments, for example in Eq. 1.1. The integrated luminosity

$$L = \int \mathcal{L}\,\mathrm{d}t \tag{2.2}$$

is a measure of the total number of collected events and hence drives the statistical significance of particle physics analyses that are conducted on the data sample. Between 2015 and 2018, in Run 2 of the LHC, the ATLAS Experiment collected $140.1 \pm 1.2\,\text{fb}^{-1}$ at a center-of-mass energy of $\sqrt{s} = 13\,\text{TeV}$ in pp collisions [97].

2.2. Overview of the ATLAS Detector

The ATLAS detector [89, 99] is the largest detector at the LHC. It is made up of a number of subsystems that are arranged in layers around the Interaction Point (IP). An overview of the subsystems of the ATLAS detector is shown in Figure 2.2.

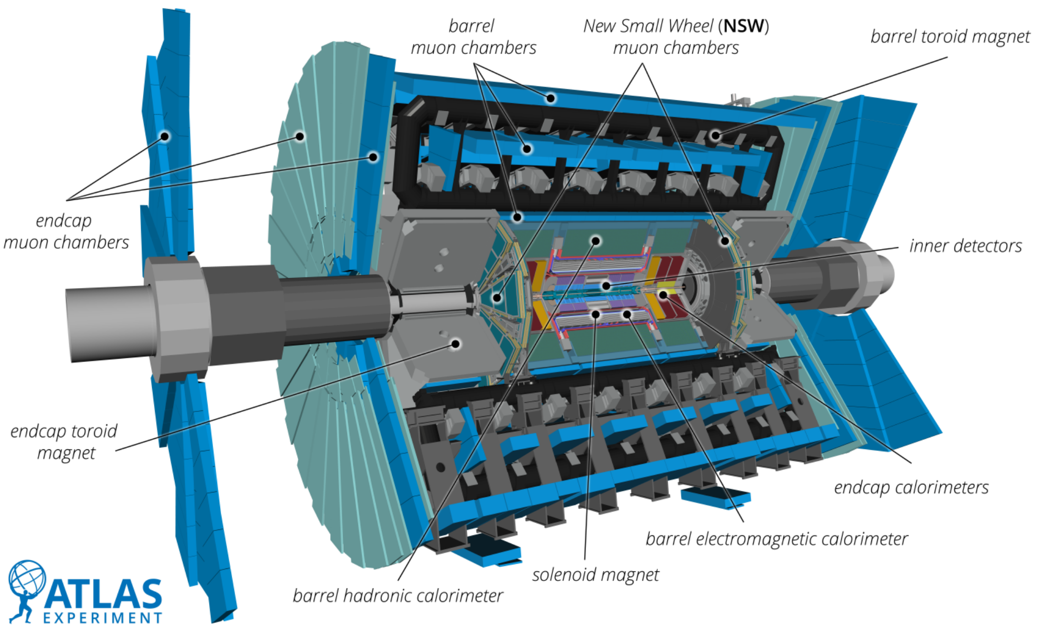


Figure 2.2.: Overview of the ATLAS detector setup and its subsystems [98].

The IP constitutes the origin of the coordinate system of the ATLAS detector. The z-axis of this right-handed coordinate system points in the direction of the beam, with the x-axis pointing towards the centre of the ring and the y-axis pointing upwards. The quantities used to describe the physics in the experiment will be expressed in this coordinate system. The transverse momentum, for example, is the projection of the particle momentum to the transverse plane according to:

$$p_T = \sqrt{p_x^2 + p_y^2}\,. \tag{2.3}$$

Another useful quantity that is often used to describe the polar angle θ of particle tracks, is the *pseudorapidity*:

$$\eta = -\ln\left(\tan\left(\frac{\theta}{2}\right)\right)\,. \tag{2.4}$$

In order to measure charged particle momentum from the curvature of the trajectory, which is described in more detail in Section 3, the ATLAS detector contains two magnet systems [100–102]. They provide a solenoid field of 2 T in the inner detector layers and a field generated by a characteristic toroidal magnet in the muon system.

Pixel and Strip Trackers Closest to the beampipe, in which the proton-proton collisions take place, lies the Inner Detector (ID) [103–105]. The ID contains a pixel silicon tracker, a strip system (SCT) and the Transition Radiation Tracker (TRT), as shown in Figure

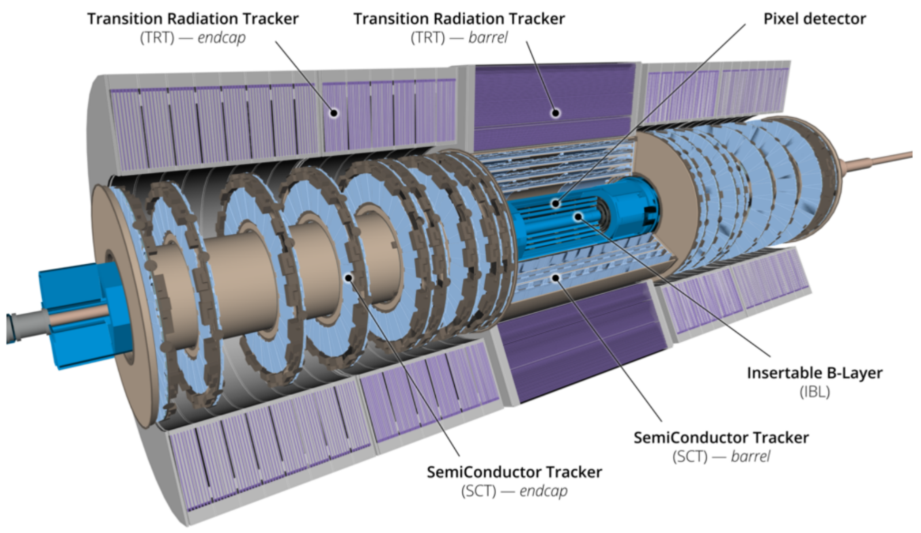


Figure 2.3.: ATLAS Inner Detector, including the pixel detector, SCT and TRT [99].

2.3. The pixel and strip systems are silicon-based semiconductor detectors [106, 107], which collect the electron-hole pairs generated by traversing charged particles through ionisation in the bulk material. The charge carriers thus produced drift in the field of the space-charge region which forms at the contact of two differently doped semiconductor materials, widened by a reverse-bias voltage. Doping describes the process of adding a different kind of atom into the lattice structure of a semiconductor, adding mobile charge carriers which are either electrons (n-doping) or holes (p-doping). At the boundary of two such materials, the charge carriers cancel each other out by diffusion, leaving a depleted region with an electric field that opposes the diffusion of further charge carriers. The signal created by a traversing particle is collected at the closest readout electrodes and subsequently encoded to either digital (yes/no) or analogue cell information.

In its final configuration, the pixel system of the ID consists of three cylindrical *barrel* layers around the IP, plus the subsequently added *Insertable B-Layer* (IBL), as well as three disc-shaped *endcap* layers that are arranged along the z-axis on both sides. The modules are placed with a small tilt angle and an overlap to ensure maximal spatial coverage without gaps. In a pixel system, the area of the sensor is segmented in both local directions on the sensor plane, which allows a precise measurement in local x and y directions. The pixel size, or pitch, ranges from $50\,\mu\text{m} \times 400\,\mu\text{m}$ for the external pixel layers of the ID [108] to $50\,\mu\text{m} \times 250\,\mu\text{m}$ for the IBL. The pixel pitch is related to the spatial resolution of the detector, which in a magnetic field lies around $10\,\mu\text{m}$ in the small pitch direction and $115\,\mu\text{m}$ in the large pitch direction [109] for the external layers and at around $10.0\,\mu\text{m}$ and $66.5\,\mu\text{m}$ for the IBL [110].

The strip system, or Semiconductor Tracker (SCT), consists of four barrel layers and nine endcap layers, surrounding the pixel detector. In a strip detector, only one local direction comes with a fine-grained segmentation. Due to this, one local direction of the measurement cannot be constrained. In order to obtain the precise position of a traversing particle in a strip detector, two sensors are mounted back to back, with a small rotation angle of 40 mrad against each other, known as the stereo angle. By observing which strips are activated on either sensor, the hit position can be determined. The strips in the barrel section have a pitch of 80 μm in the precision direction, which yields a spatial resolution of 17 μm along the z-axis and 580 μm along the $r\phi$-axis when correlating the two module sides [111]. In the endcap section, the pitch varies overall between 57 μm and 94 μm, with the strips aligned in wedge shape on the trapezoidal sensors [112].

The pixel and strip systems of the ID cover a region of $|\eta| \leq 2.5$ at a radius of about 33 mm up to 514 mm for the layers in the barrel section. Due to the high granularity, the occupancy of the pixel layers, meaning the percentage of active channels per readout, is much lower than that of the SCT. For this reason, the seeding step of the track reconstruction chain starts in the innermost ID layers, then continuing to add new measurements from the inside out. In 2008, the relative momentum resolution during cosmic ray track reconstruction in the ID was measured to be $\sigma(p)/p = (0.0483 \pm 0.0016)\%\, p_T$ [113].

Transition Radiation Tracker The last subsystem of the Inner Detector is the *Transition Radiation Tracker* (TRT) [89, 103, 114], which applies a different detection technology from the other two sub-detectors of the ID. It contains straw tubes, which are made up of a sense wire that lies in a tube filled with a gas mixture and are kept at a higher electric potential compared to the walls of the tube. The gas mixture gets ionised when a charged particle passes through it and the electrons produced in this process are gathered and amplified at the sense wire, triggering a measurable signal. At each end of the straw, the signal is sampled in short time intervals of a few nano-seconds, yielding information on the drift time in the tube. This corresponds to a *drift circle* around the wire, which measures the distance of the point of closest approach of the particle to the sense wire.

The TRT contains in total hundreds-of-thousands of straw tubes of 4 mm diameter, with a design intrinsic resolution of 130 μm. It has a radial extent of 560 mm to 1080 mm and comprises a pseudorapidity region of $|\eta| \leq 2.0$. The barrel section contains around 50 000 straws parallel to the beamline, while the endcaps are made up of around 120 000 radially aligned straws each. Depending on the detector region, the TRT contributes more than 30 measurements to a particle trajectory.

By measuring the *transition radiation* [115–117] of passing charged particles, the TRT can also perform particle identification. Transition radiation is emitted when a charged particle traverses the boundary between two media. Its intensity depends on $\gamma = 1/\sqrt{1-\beta^2}$, which is sensitive to the particle mass in combination with the momentum. The transition radiation in the TRT is triggered by polymer fibres in between the straw tubes in the barrel section and polymer foils in the endcaps. It is measured when the transition radiation photons are absorbed by the gas mixture in the straws, which can be distinguished from the ionisation signal created by the traversing particles by applying a higher signal threshold.

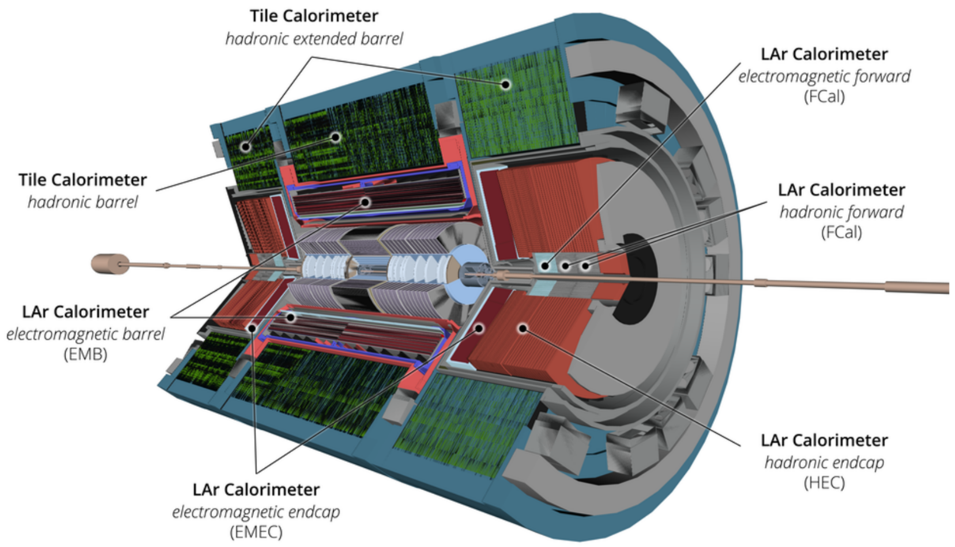


Figure 2.4.: The ATLAS electromagnetic and hadronic calorimeter systems [99].

Calorimeters The next detector subsystem is made up of two different calorimeter technologies, which measure the total energy deposition by predominantly electromagnetic (Liquid Argon Calorimeter LAr) or hadronic showers (Tile Calorimeter [89, 118, 119]). The goal of the calorimeter systems is to contain a particle and stop it in the detector material, thus measuring the complete energy it carried. To facilitate this, the calorimeters in the ATLAS detector feature dense absorber materials to induce particle showers, as described in Section 1.2. In between the absorber material, an active material is placed to quantise the amount of shower particles. An overview of the ATLAS calorimeter setup in shown in Figure 2.4.

Including both the barrel and endcap sections of the LAr electromagnetic calorimeter, it covers an η-range of $|\eta| < 3.2$. The active material is liquid argon, in which the particle showers are sampled by the ionisation of charged particles. The readout electrodes and the lead absorbers are placed in an "accordion-shape". It provides a design energy resolution of $\sigma_E/E = 10\,\%/\sqrt{E} \oplus 0.7\,\%$ for energies measured in GeV.

A section of the LAr endcap system in the range of $1.5 < |\eta| < 3.2$, is equipped with copper absorber plates and used as hadronic endcap calorimeter (HEC). This is extended to $|\eta| < 4.9$ by the liquid-argon forward calorimeter (FCal), which uses rod-shaped electrodes surrounded by a tube of liquid-argon in a metal matrix of either copper for electromagnetic measurements or tungsten for hadronic measurements. The hadronic tile calorimeter, which surrounds the LAr calorimeter, relies on steel absorbers alternating with plastic scintillator tiles in which scintillation light of shower particles is measured. The tile calorimeter covers a range of $|\eta| < 1.7$, with an energy resolution goal of $\sigma_E/E = 50\,\%/\sqrt{E} \oplus 3\,\%$ in the barrel and endcap sections. In the forward calorimeter, the design energy resolution is $\sigma_E/E = 100\,\%/\sqrt{E} \oplus 10\,\%$.

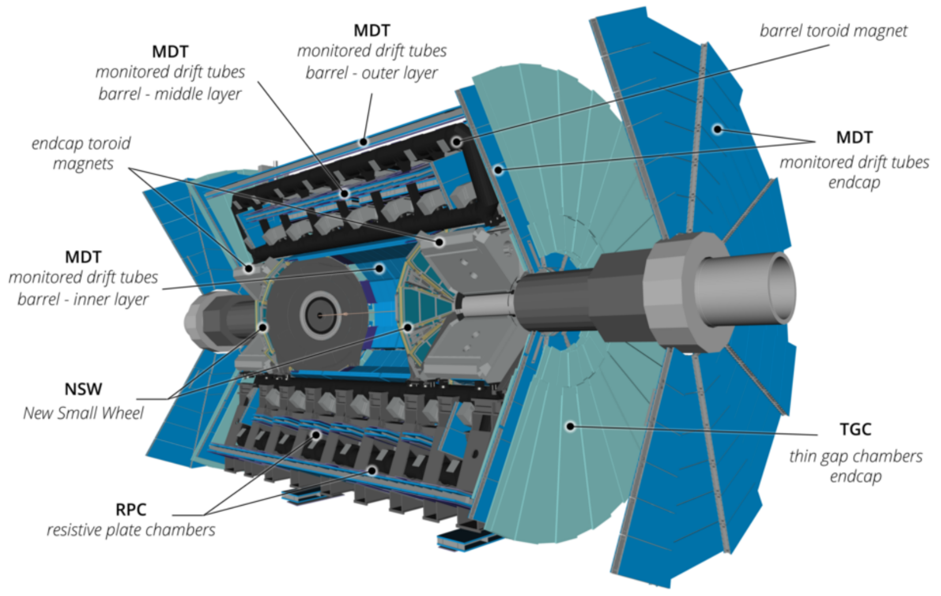


Figure 2.5.: Overview of the ATLAS Muon System, including the New Small Wheel [99].

Muon System Last but not least among the sub-detectors in ATLAS is the muon system [89, 120]. Since muons are minimum ionising particles and do not deposit much energy when traversing matter, they can pass through the inner detector layers and calorimeters and will be detected in surrounding detector layers. The ATLAS muon system consists of different chamber technologies, which are arranged into *stations* and mounted around the calorimeter system in three layers. It covers a total η-range of $|\eta| < 2.7$ with a momentum resolution goal of $\sigma_{p_T}/p_T = 10\,\%$ for muons with a transverse momentum of $p_T = 1\,\text{TeV}$. Where the different chamber technologies are deployed, is shown in Figure 2.5.

Precision momentum measurements are achieved by drift tubes, the *Monitored Drift Tubes* or MDTs, and, originally, Cathode Strip Chambers (CSC) in the region of $|\eta| > 2$. The CSCs are multiwire proportional chambers, which read out a charge signal that is induced in the segmented cathode chamber wall by an avalanche of electrons collected at multiple anode wires when a muon passes the chamber and ionises the contained gas mixture. A fast muon response is provided by the *Resistive Plate Chambers* (RPC)s and *Thin Gap Chambers* (TGCs). The fast signals from these detectors are used in the ATLAS trigger system. The RPCs are gaseous detectors, in which a drift field is produced between two parallel Bakelite plates and the ionisation signal in the contained gas mixture is read out by metal strips on either side of the chamber. In the endcap regions, the TGCs are used, which are also multiwire proportional chambers, but with a smaller distance between the wires and the cathode wall, boosting the time resolution.

For ATLAS Run 3, the *New Small Wheel* [121] has been installed in the innermost endcap region of the muon system, between $\eta = 1.3$ and $\eta = 2.7$, replacing the CSCs

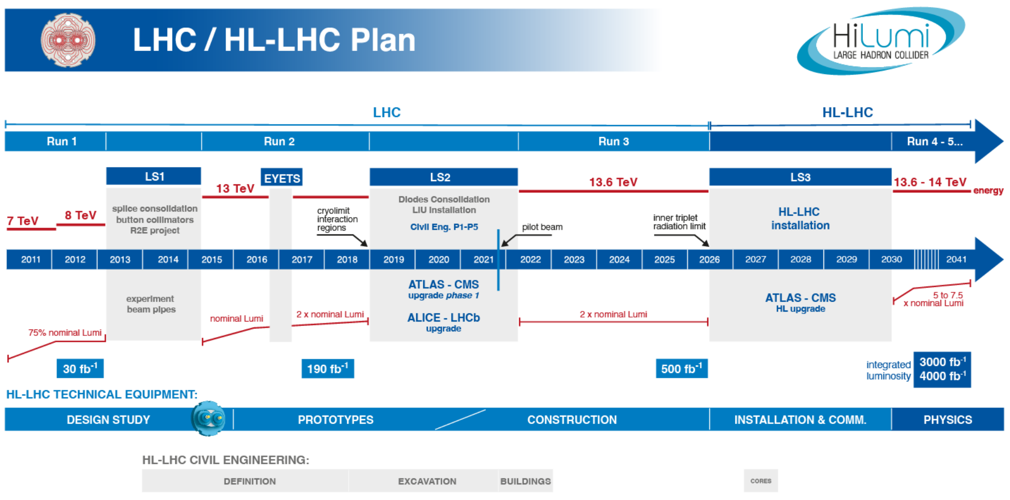


Figure 2.6.: Timeline of the LHC operation [125].

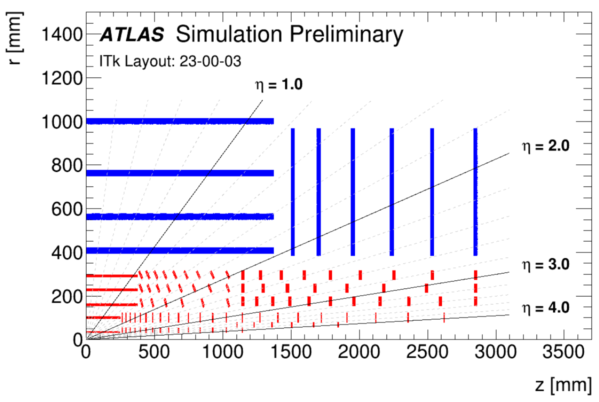
and some of the MDTs of the old Small Wheel. The replacement was done in order to maintain the tracking efficiency and resolution and to reduce the fake trigger rate in the muon system endcaps, which are both expected to degrade with the higher luminosity in Run 3 compared to Run 1 and 2.

Trigger and Computing System In order to reduce the readout rate of the full detector data towards the backend storage, ATLAS applies a two-layer trigger system [99, 122, 123]. The level-1 trigger is implemented in hardware and reduces the rates from the 40 Hz collision rate down to 100 kHz, using some limited information from the calorimeter and the muon systems. If the level-1 trigger decision is reached, the event will undergo a fast reconstruction in the software-based *High Level Trigger* (HLT), which reduces the rate further down to about 3 kHz.

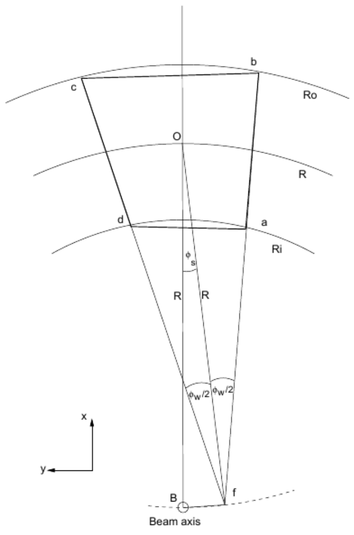
Once the decision to keep an event is made, the readout data is transferred to the *Tier-0* site in the CERN computing centre [124]. Here, a first reconstruction and calibration is performed and the raw data is archived and distributed to the compute facilities (Tier-1, Tier-2 and Tier-3) that are hosted at ATLAS member Institutes around the world for further processing, archiving and, in the end, physics analysis.

2.3. The new ATLAS Inner Tracker (ITk)

The LHC, and with it the experiments stationed along the accelerator ring, will get an upgrade designed to bring the instantaneous luminosity to an unprecedented level, with a target of $\mathcal{L} = 5 \cdot 10^{34}\,\mathrm{cm}^{-2}s^{-1}$ [126]. This will increase the statistics of the data that is



(a) Layout of the ITk: Pixel system in red and strip layers in blue [127]



(b) ITk strip endcap stereo annulus shape [130]

Figure 2.7.: The ATLAS Inner Tracker (ITk) detector setup for the HL-LHC.

collected and thereby allow much more precise physics measurements. The timeline for this upgrade to the *High Luminosity-LHC* (HL-LHC) is shown in Figure 2.6.

With the increased luminosity comes an increased mean pile-up value, denoted $\langle\mu\rangle$. It describes the number of interactions per bunch crossing, which has been around 50 or 60 for the current Run 3 and will increase possibly up to $\langle\mu\rangle \sim 200$ for the HL-LHC. This means, that many more particle interactions will take place per collision event, increasing the probability to observe rare processes, but also the occupancy in the detector systems. This will be especially true for the inner detector layers.

In order to keep the same physics performance in the high pile-up environment of the HL-LHC [127], the ATLAS detector will be equipped with a completely new silicon pixel and strip detector, called the Inner Tracker (ITk) [128, 129]. It will replace the ID, increasing the acceptance in pseudorapidity from $|\eta| < 2.5$ to $|\eta| < 4$. The ITk will be an all-silicon detector, without the equivalent of the current TRT. It will contain a pixel and a strip system, with five and four barrel layers respectively. The pixel detector will also include a number of inclined endcap layers, which align the sensors better with regards to the particle direction and allows to use smaller sensors and hence improve the material budget of the detector. The layout of the ITk is shown in Figure 2.7a.

The shape of the endcap strip sensors is called a *stereo annulus* shape [130], since it features a section of an annulus, but with the origin of the section shifted from zero, which corresponds to the beamline when mounted in the detector. This way, the sensors can be mounted with fitting radii, while at the same time featuring a stereo angle of 20 mrad per sensor. The stereo annulus shape is shown in Figure 2.7b.

CHAPTER 3

Track Reconstruction in Silicon Detectors

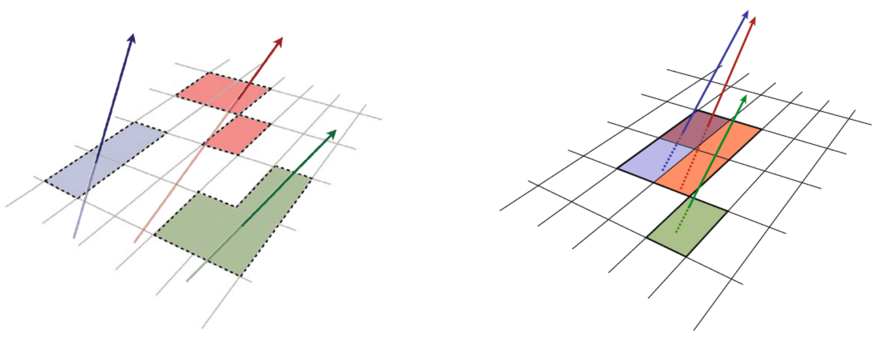
A crucial element of the reconstruction chain of any particle physics experiments is the track reconstruction step, or in the following called *tracking*. It is responsible for connecting the single measurement points that were recorded in a detector during, for example, a particle collision experiment to a coherent trajectory. A magnetic field $\mathbf{B}$ is usually used to induce bending in the tracks, which allows to measure the particle momentum. For example, in a homogeneous magnetic field, the momentum $|\mathbf{p}|$ can be inferred from the track radius R and the pitch angle λ of the track:

$$|\mathbf{p}| \cos(\lambda) = 0.3\, |\mathbf{B}| R.$$

In the tracking chain of a particle physics experiment, the full information on the particle momentum is usually obtained by a dedicated track fitting step, such as can be performed with a *Kalman Filter*. The particle momentum is a crucial piece of information in particle physics analyses, as it allows to reconstruct event kinematics and from it important information, such as the invariant masses of final and intermediate particle states. The following chapter restricts itself to the Kalman formalism for track finding and fitting in the context of ATLAS Inner Detector tracking. Other methods of track finding and fitting exist in ATLAS as well, for example in Refs. [131–133].

3.1. From Detector Activations to Track Parameters

The tracking chain [133, 135, 136] starts with producing points in global 3D and local 2D coordinates on the sensors from the detector measurements, which are a pre-requisite for the following track finding and fitting steps. For the single cell activations, originating from either the pixels or strips of the silicon trackers, this is generally done with a *Connected Component Analysis* (CCA) [137]. This algorithm will iteratively identify adjacent cells if they either share a boundary or a corner and add them to *cluster* objects,



(a) Single clusters (b) Merged clusters

Figure 3.1.: Clusters and incident particles in a silicon pixel detector. Figure from [136].

which represent the charge deposition by ionisation of a particle crossing through the sensor, as shown in Figure 3.1.

The position of a cluster is an estimation of the point where the particle crossed the sensor plane and can be inferred from the mean of the positions of its cells, optionally weighted with the charge collected per cell. In dense environments, such as particle jets, multiple particles may contribute to the same cluster, as can be seen in Figure 3.1b, which can subsequently be split using a neural network approach [138, 139]. This set of neural networks is also capable of performing the initial cluster identification and position estimation steps.

Once the cluster positions are known and transformed into the global detector coordinate frame, the initial track parameters with which to start the track finding step need to be produced. This is done in the *track seeding* and *parameter estimation* steps [140], where triplets of compatible 3D space points are formed according to a number of cuts and quality criteria. After the best seeds have been filtered, a set of track parameters is derived from them, which can be used for the track state propagation.

3.2. Track State Propagation

In order to infer particle properties for physics analyses, it is critical to describe the particle's trajectory through the detector and magnetic field precisely. In the absence of material effects, the Equation of Motion with regard to the arc length $\mathrm{d}s = v\,\mathrm{d}t$ of a particle of charge q and momentum p, in a magnetic field $\mathbf{B}(\mathbf{r})$ is given by the Lorentz force, parametrised along the path length s:

$$\frac{\mathrm{d}^2\mathbf{r}}{\mathrm{d}s^2} = \frac{q}{p}\left(\frac{\mathrm{d}\mathbf{r}}{\mathrm{d}s} \times \mathbf{B}(\mathbf{r})\right). \tag{3.1}$$

The solution in the case of a homogeneous field is a helical trajectory $\mathbf{r}(s)$ with its tangent, given by $\mathbf{T}(s) = \mathbf{p}(s)/|\mathbf{p}(s)|$, and a particular start position $\mathbf{r}_0 = \mathbf{r}(s_0)$ and direction

$\mathbf{T}_0 = \mathbf{T}(s_0)$ [141, 142]:

$$\begin{aligned}\mathbf{r}(s) &= \mathbf{r}_0 + \frac{\gamma}{Q}\,(\theta - \sin\theta)\,\mathbf{H} + \frac{\sin\theta}{Q}\,\mathbf{T}_0 + \frac{\alpha}{Q}\,(1-\cos\theta)\,\mathbf{N}_0,\\ \mathbf{T}(s) &= \frac{\mathrm{d}\mathbf{r}}{\mathrm{d}s} = \gamma\,(1-\cos\theta)\,\mathbf{H} + \cos\theta\,\mathbf{T}_0 + \alpha\sin\theta\,\mathbf{N}_0,\end{aligned} \tag{3.2}$$

where $\mathbf{H} = \mathbf{B}/|\mathbf{B}|$, $\alpha = |\mathbf{H}\times\mathbf{T}|$, $\mathbf{N} = (\mathbf{H}\times\mathbf{T})/\alpha$, $\gamma = \mathbf{H}\cdot\mathbf{T}$, $Q = -q\,|\mathbf{B}|/|\mathbf{p}|$ and $\theta = Q\cdot s$. In general, however, no closed solutions exist for the inhomogeneous fields used by particle physics experiments, including ATLAS. Then, Equation 3.1 must be solved numerically, as described, for example, in Section 3.2.1.

The *track parametrisation* is key to the description of the particle trajectory in an arbitrary field. It allows to express the state of the track in a fully constrained parameter space that relates to the measurements provided by the detector setup or are required by the numerical integration. In the ATLAS *Event Data Model* (EDM) [143, 144] the track parametrisation is defined in the global Cartesian coordinate frame (also called free parametrisation) or with regard to a detector reference surface. Depending on whether it is a global or a local parametrisation, it contains either global $(x,\,y,\,z)$ or surface-local $(l_0,\,l_1)$ positions, two angles ϕ and θ at the track position in the global frame or the tangential track direction $\mathbf{T}$, where:

$$\begin{aligned}p_x &= p\,T_x = p\sin\theta\cos\phi,\\ p_y &= p\,T_y = p\sin\theta\sin\phi,\\ p_z &= p\,T_z = p\cos\theta,\end{aligned}$$

as well as the particle charge over total momentum $(\lambda = q/p)$. These can be written as vectors in a 7- or 5-dimensional space for global $\mathbf{G}$ and local $\mathbf{L}$ track parameters, respectively:

$$\begin{aligned}\mathbf{G} &= \left[x,\,y,\,z,\,T_x,\,T_y,\,T_z,\,\lambda\right]^{\mathrm{T}},\\ \mathbf{L} &= \left[l_0,\,l_1,\,\phi,\,\theta,\,\lambda\right]^{\mathrm{T}}.\end{aligned} \tag{3.3}$$

For the local track parametrisation, the reference surface has to be known in addition to fully constrain the state of a track. A track object in the ATLAS EDM consists of a collection of *track states* which capture the track parameters at a given surface, as well as potentially corresponding measurements and some algorithm specific information.

In order to pick up new measurements during track reconstruction that are consistent with the current track hypothesis, the covariance matrix for the track parametrisation at the corresponding reference surface is used. Its elements for the local track parameter vector are defined as follows:

$$\boldsymbol{\Sigma} = \begin{bmatrix}\mathrm{var}(l_0) & \mathrm{cov}(l_0,l_1) & \dots & \mathrm{cov}(l_0,\lambda)\\ & \mathrm{var}(l_1) & \ddots & \vdots\\ & & \ddots & \mathrm{cov}(\theta,\lambda)\\ & & & \mathrm{var}(\lambda)\end{bmatrix}. \tag{3.4}$$

The covariance matrix is initially estimated from the measurement error on the first surface along the track and subsequently updated in a process that is detailed in Section 3.2.2.

3.2.1. Field Integration

As a track object is assembled, the track parameters need to be transported through the magnetic field by integrating Equation 3.1. In ATLAS, this is done using an adaptive Runge-Kutta-Nyström (RKN) algorithm [145, 146], which is designed to iteratively solve systems of second order differential equations of the form:

$$\frac{\mathrm{d}^2 y(x)}{\mathrm{d}x^2} = y''(x) = f(x,\, y(x),\, y'(x)), \qquad y_0 = y(x_0), \qquad y'_0 = y'(x_0),$$

with x_0, y_0 and y'_0 as the start parameters. The RKN algorithm is a fourth order method, meaning that the approximation error is $\mathcal{O}(h^4)$, where h is the integration step size. To achieve this, four intermediate *stages* are calculated in an iteration step n as

$$\begin{aligned}
k_1 &= f(x_n,\, y_n,\, y'_n),\\
k_2 &= f(x_n + \frac{h}{2},\, y_n + \frac{h}{2}\, y'_n + \frac{h^2}{8}\, k_1,\, y'_n + \frac{h}{2}\, k_1),\\
k_3 &= f(x_n + \frac{h}{2},\, y_n + \frac{h}{2}\, y'_n + \frac{h^2}{8}\, k_1,\, y'_n + \frac{h}{2}\, k_2),\\
k_4 &= f(x_n + h,\, y_n + h\, y'_n + \frac{h^2}{2}\, k_3,\, y'_n + h\, k_3)
\end{aligned} \tag{3.5}$$

to produce the solution at step $n + 1$:

$$\begin{aligned}
y'_{n+1} &= y'_n + \frac{h}{6}\,(k_1 + 2\,k_2 + 2\,k_3 + k_4),\\
y_{n+1} &= y_n + h\, y'_n + \frac{h^2}{6}\,(k_1 + k_2 + k_3),\\
x_{n+1} &= x_n + h.
\end{aligned} \tag{3.6}$$

An adaptive Runge-Kutta algorithm is characterised by an additional prescription on how to dynamically set the step size according to the truncation error in a given step. For this, a higher order term is calculated and subtracted from the current solution. By calculating the step size for the subsequent step from the current error estimate and a user-provided error tolerance, the algorithm can reduce or increase the step size automatically to keep the integration error low, while at the same time minimising the number of steps that need to be taken. If the resulting error estimate exceeds the error tolerance, the step is rejected and redone using the updated smaller step size h. In Ref. [146], the error estimate is calculated from the fourth order term of the Taylor expansion around the half-step $x_n + 1/2\, h$ in the RKN method as:

$$\epsilon = \frac{h^2}{6}\,(k_1 - k_2 - k_3 + k_4), \tag{3.7}$$

thus reusing the RKN stages that were calculated during the integration step, which makes this error estimation very efficient to compute.

3.2.2. Covariance Transport

The track parameter covariance in Equation 3.4 can be transported along the track to linear order by a transformation with a Jacobian J [147, 148], derived from the propagation function which transforms the track parameters from their initial to their final values:

$$\mathbf{\Sigma}^f \approx \mathbf{J}\,\mathbf{\Sigma}^i\,\mathbf{J}^T. \tag{3.8}$$

Here, the Jacobian is expressed as a matrix of the individual derivatives:

$$\mathbf{J} = \begin{bmatrix} \frac{\partial l_0^f}{\partial l_0^i} & \cdots & \frac{\partial l_0^f}{\partial \lambda^i} \\ \vdots & \ddots & \vdots \\ \frac{\partial \lambda^f}{\partial l_0^i} & \cdots & \frac{\partial \lambda^f}{\partial \lambda^i} \end{bmatrix}. \tag{3.9}$$

For a helical track in a homogeneous field, the propagation function and hence the Jacobian can be calculated analytically, as shown in Refs. [142, 149, 150]. When propagating the track parameters through an inhomogeneous magnetic field, the Jacobian has to be determined using numerical methods. In ATLAS, the Jacobian is calculated in a *semi-analytical* way using the *Bugge-Myrheim-Method* for the RKN-based propagation, which is described in Refs. [147, 148]. Deriving Equation 3.6 with respect to the global track parameters yields an iterative approach to calculating the transport Jacobian

$$\mathbf{J}_{n+1}^{RKN} = \mathbf{D}_n \cdot \mathbf{J}_n^{RKN}, \tag{3.10}$$

with the transport matrix $\mathbf{D}_n$ expressed as

$$\mathbf{D}_n = \frac{\partial\left(\mathbf{F}_n, \mathbf{G}_n\right)}{\partial\left(\mathbf{u}_n, \mathbf{u}'_n\right)}, \tag{3.11}$$

where $\mathbf{F}\left(\mathbf{u}_n, \mathbf{u}'_n\right)$ and $\mathbf{G}\left(\mathbf{u}_n, \mathbf{u}'_n\right)$ are a short-hand for Equation 3.6, substituted with the equation of motion in Equation 3.1 and an additional term for the track state energy energy loss. The full derivation of the 8×8 matrix $\mathbf{D}_n$ can be found in Ref. [147].

The transport matrix is expressed in global coordinates, so that the initial and final contributions to the full Jacobian need to encompass the Jacobians for the coordinate transformations from and to the local track parameters.

3.3. Track Finding and Fitting

The Kalman Filter (KF) [151] is an iterative approach to track fitting that makes use of the current and all previous measurements along the track, as well as their uncertainties.

It revises the track states along the trajectory of a charged particle by applying updates to the track parameters according to the measurement found for that particular step. The explicit mathematical formulation in the context of track reconstruction can be found in Refs. [152–156].

The filter update starts with an initial track state and then performs a prediction of the track parameters at the next detector surface based on the current track parameters. To first order, this can be done with a linear transformation of the track state. Similar to the track parameters, the KF will make a prediction of the track parameter covariance matrix at the next surface.

The predicted track parameters and covariance matrix are then updated with the new measurement information found at the detector surface in what is called the filtering step. The update of the track parameters is done using the *Kalman Gain Matrix* on the residual between measurement and the predicted track parameters, which are projected into the measurement space. The gain matrix depends on the predicted covariance matrix and the measurement error and is also used to filter the covariance matrix prediction itself.

Since this process leaves the filtered states from the beginning of the fit decoupled from information found at later stages, a final smoothing pass needs to be added. In this step, the Kalman Filter is run backward along the track states and updates them one more time with the full information from all subsequent measurements. It can be shown, that the final track fit afterwards yields optimal track parameters with regards to the squared residuals χ^2, if the noise due to material interactions and the measurement errors can be modelled by Gaussian distributions.

In order to perform the track finding step that yields the track states and measurements for the KF to run over, the large combinatorics in the inner layers of a silicon tracker have to be taken into account. This can be done, for example, by exploiting the track state prediction step of the Kalman filter formalism and then *branching* into a new track candidate with every compatible measurement that is close to the predicted state [157–159]. This is then repeated with the set of old and new track candidates at the next step, leading to a tree-like progression of track candidates. This method is called a *Combinatorial Kalman Filter* (CKF) and can be started as soon as track parameters and their associated covariances have been estimated from a track seed.

After track finding and fitting (but sometimes before the fitter runs), an ambiguity solver is run over the track candidates, so as to filter out spurious tracks that likely do not correspond to an actual particle trajectory, called *fake tracks*. Various criteria for this exist, such as the overall number of hits, the number of hits that are shared with another track candidate, as well as the number of holes. The latter refers to instances where no compatible measurement could be found at a sensitive detector surface.

CHAPTER 4

The ACTS Project

ACTS, or *A Common Tracking Software* [160, 161], is a community software project to develop a detector independent toolkit of tracking data structures and algorithms which are written in modern `C++` and feature a threadsafe, maintainable design. Being a toolkit, the ACTS algorithms do not constitute a stand-alone tracking framework, but should instead be incorporated into the reconstruction software stack of particle physics experiments. It has been adopted, for example, by the sPHENIX [162] collaboration and is planned to become the backbone of the tracking software for the Phase-II Upgrade in ATLAS (ITk) [126].

Its main components are a tracking *Event Data Model* (EDM), which defines the data structures to describe, for example, track states or measurements, a tracking geometry description including material handling, a track state propagation that manages the track parameter and covariance transport, as well as a vertexing module, which is used to find the origin of tracks that stem from a common source. The tracking implementation in ACTS is based on the tracking chain of ATLAS [143, 144], for which the main components in the context of silicon trackers were outlined in Chapter 3. An overview of the ACTS components is shown in Figure 4.1. In this chapter the main geometry and material related concepts will be presented.

4.1. Tracking Geometry and Navigation

The tracking geometry [163] is a core ingredient for track reconstruction as it provides the link between the global track state and the detector surfaces with which the measurements are associated. In contrast to geometry descriptions like Geant4 [164], the ACTS geometry description is purely surface based and in effect more lightweight, since the fully detailed modelling of the detector, as it is used in particle simulations, is not needed for the purpose of track reconstruction. Where volumes are needed, like in the

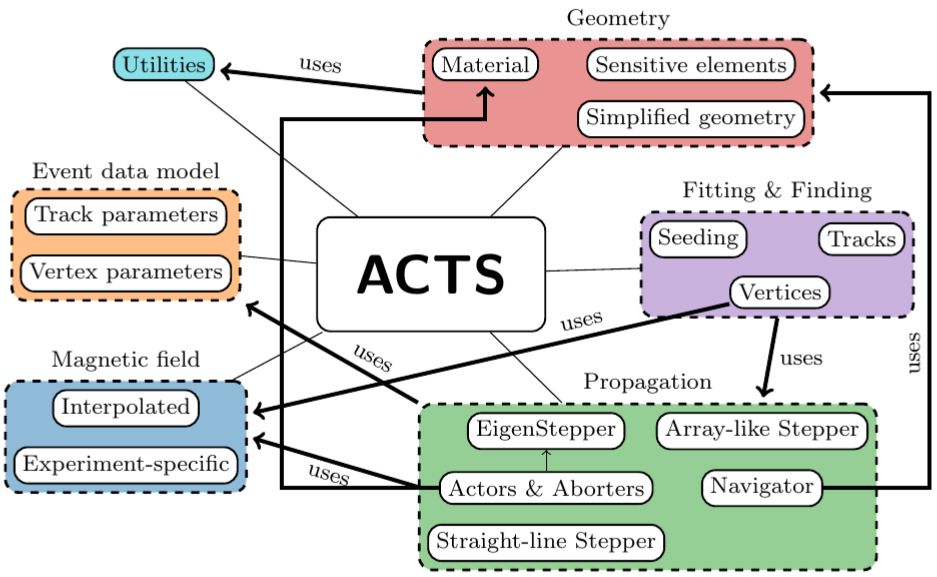


Figure 4.1.: Overview of the components of the ACTS project, highlighting its main modules. Taken from Ref. [160].

detector navigation, they are represented by their boundary surfaces, which they share in conjunction with so-called *portals* with their neighbouring volumes. Portals are those parts of a boundary surface, where a track can move from one volume to a neighbouring one. This is important for track following algorithms like the CKF.

A surface is described by its underlying geometry, for example, a plane surface or a cylinder surface, and defined by its boundaries, like a rectangle or a trapezoid. In order to model a detector as a tracking geometry, the sensors are represented by sensitive surfaces, which have a call-back mechanism to an implementation of a *detector element* that makes the full information about the sensor available if needed. The implementation of the detector element is, in general, meant to be done by the client experiment.

The sensitive surfaces are assembled in layers, of which many can exist in a detector volume. A layer object can represent, for example, one of the layers of a pixel detector, which itself may be subdivided into volumes according to the barrel, as well as positive and negative endcap sections. Recently, a new "layerless" design has been implemented in ACTS as an experimental geometry, inspired by the initial DETRAY implementation. This simplifies the navigation through the geometry, as no extra step to resolve the layers in a volume is needed anymore.

Broadly speaking, the navigation determines the distance to the closest surface in the detector, thus providing a maximal step length for the Runge-Kutta-Nyström algorithm. The latter is implemented in the *stepper* class and performs the integration of the equation of motion of a charged particle in the presence of a magnetic field, as detailed in Section 3.2. While the stepper advances the track parameters through the detector, the navigator

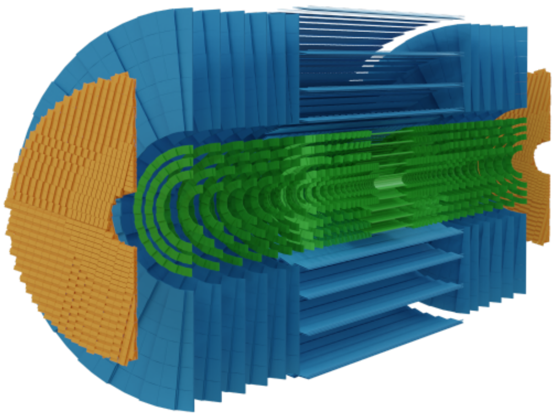


Figure 4.2.: 3D representation of the ACTS ITk tracking geometry. The pixel detector is shown in green, the strip system in blue and the *High Granularity Timing Detector* (HGTD) [165] in orange. Figure taken from Ref. [160].

determines whether or not a particular detector surface was reached by the track. The interplay of both is steered by the *propagator* class, together with a number of pluggable *actors* and *aborters* which can implement custom tasks. In case the surface that was reached is a sensitive surface, the tracking algorithms will run as actors, otherwise the layer or volume navigation is triggered.

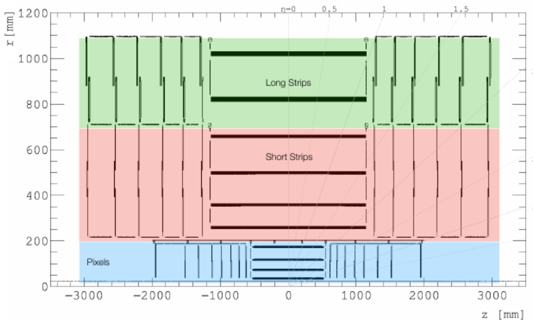
4.1.1. The Open Data Detector

Inspired by the ACTS Generic detector, which was used with an accompanying data set in a machine learning tracking contest [166], the *Open Data Detector* (ODD) was made available [167] for generic tracking research. It features a more realistic setup than the Generic detector, including passive material, such as support structures and cooling pipes. The full geometry is described using the DD4hep library [168] and supports Geant4 based simulation. The tracking geometry is constructed in ACTS with a specific plugin. It will be used in the context of this thesis as a realistic test detector geometry to validate the DETRAY propagation implementation.

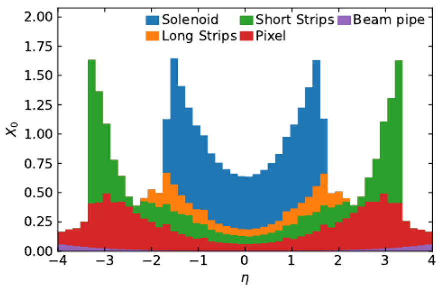
The ODD has a pixel detector with four barrel layers and a number of endcaps. The surrounding layers contain a short and a long strip system with a total of six layers. The ODD geometry and the associated material budget are shown in Figure 4.3.

4.2. Material Mapping

Similar to the tracking geometry, the material description in ACTS is inspired by the ATLAS material map description [163] and presents a simplified setup with regard to the exact placement of the material that is needed for simulation. In the material mapping procedure, the material budget is gathered from the full geometry description along a



(a) Layout of the ODD projected to rz: Pixel system in blue, short strips in red and long strips in green.



(b) Material budget in X_0 for different sub-systems of the ODD.

Figure 4.3.: Geometry layout and material of the ODD. Figures taken from Ref. [166].

material track described by a Geantino [164]. A Geantino is a test particle type that can be used in Geant4 to record geometry properties during particle transport. The material is then averaged and placed on the closest surface in the tracking geometry that was marked as a material surface. In general, these will be the boundary surfaces that define a layer. This process can be seen in Figure 4.4. Once associated to a surface, the material will be put into a grid structure to allow for a more fine-grained description. A homogeneous material description, without binning, is also possible.

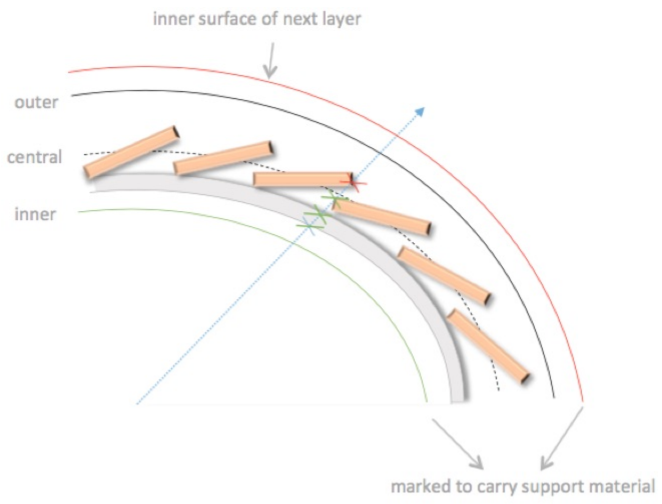


Figure 4.4.: Schematic display [169] of the material assignment during the material mapping process in ACTS. Material is accumulated along a track and mapped to the closest material surface in the tracking geometry.

CHAPTER 5

The ACTS Parallelization R&D Project

5.1. Motivation

During ATLAS offline reconstruction, one of the computationally most expensive tasks is the track reconstruction in the ID [133]. Additionally, the seeding and CKF-based track finding steps will be impacted by the combinatorics resulting from the increased number of hits in the inner detector layers at the high pile-up levels of the HL-LHC. Improvements in the computing performance of track reconstruction algorithms are therefore investigated in order to be able to run as efficiently as possible under HL-LHC conditions, while keeping the same or better physics performance. A first study has been published in Ref. [170], using a *Fast Tracking Chain*, which greatly reduces CPU resource requirements by, for example, adjusting track selection cuts during track finding and in turn skipping the costly ambiguity resolution step. This came at the expense, however, of a slight physics performance loss in efficiency and momentum resolution.

A complementary approach is the application of massive parallelism using accelerator chips, such as *General-Purpose Computing on Graphics Processing Units* (GPGPU). It is expected, that GPUs will be part of the hardware setup at many sites in the LHC computing infrastructure and potentially also used in the future in ATLAS online reconstruction [126, 171]. Therefore, being able to leverage all available hardware at High Performance Computing (HPC) sites will be advantageous.

Track finding and fitting show trivial data parallelism (called *embarrassingly parallel*) due to the independence of single events and tracks, so that an adaptation for massive parallelism might be possible. GPUs are capable of running a large number of threads scheduled in blocks on thousands of compute cores in parallel [172]. Within these blocks, the threads are arranged into groups of 32 or 64, called a warp, in which the compute instructions for all threads are generally issued simultaneously. This parallelisation approach is frequently called *SIMT* (*Single Data, Multiple Threads*) [173], specialising the

SIMD (*Single Instruction, Multiple Data*) classification in Flynn's taxonomy [174]. SIMD describes the case where a single instruction, for example, a floating point addition, is applied to multiple data of the same kind, like is the case in CPU vector instructions (*vectorization*). SIMT applies this principle, by issuing a thread with the same instruction on every piece of data. Branching into different instructions is possible in SIMT, but incurs a performance penalty (thread divergence).

GPUs are programmable through specific C/C++ language extensions like, fro example, CUDA [172, 175] or higher level abstractions such as SYCL [176]. However, existing code bases need to be adapted and restrictions that come with these programming models have to be dealt with. Furthermore, the separate memory systems of the CPU (host) and GPU (device), in general, require explicit allocations and memory copies before the computations can be run on the device.

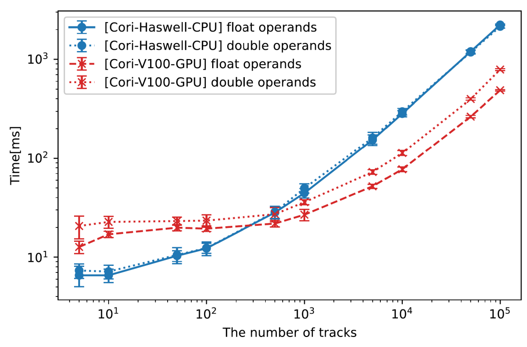
5.2. Initial GPU Studies in ACTS

Following the considerations outlined in the previous Section, a number of first studies were undertaken within the ACTS community to port specific algorithms, e.g. the Kalman fitter [177] or the track seeding [178], to CUDA. Experience that was gathered on these projects has since provided valuable insight into the expected benefits but also challenges of adapting the complete ACTS tracking chain to run on a GPU device.

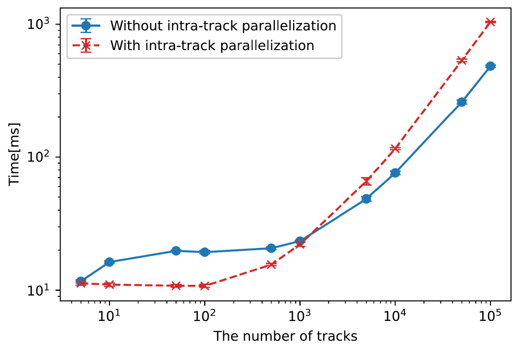
The GPU Kalman fitter implementation in [177] leverages the embarrassingly parallel nature of tracking, meaning that each track can be handled completely independently of each other until ambiguity resolution is applied. It is therefore straightforward to assign one or multiple GPU threads per track. Both modes, *inter*- and *intra*-track parallelisation, were investigated in that study, with the intra-track parallelisation targeting the matrix operations in the covariance transport and Kalman filter. At the time, using the matrix inversion of the Eigen3 [179] linear algebra library which is used in ACTS, was not possible in device code, and so a custom matrix inversion implementation in double precision was used. This resulted in a negative performance impact rendering the GPU implementation less performant than a CPU implementation using the Eigen3 matrix inversion. The performance study comparing CPU and GPU, where both use the custom inversion algorithm, however, yielded a speedup of up to 4.6 for events with more than 1000 tracks. This can be seen in Figure 5.1, which shows the algorithm latency vs. the number of tracks for both hardware backends.

Despite the achieved speedup, a number of design problems were also becoming apparent. One problem, which will be discussed in greater detail in Chapter 6, is the implementation of the ACTS tracking geometry description. Due to its use of an object-oriented design, based on polymorphism that is resolved at runtime, as well as a *vector-of-vector* memory layout, it became impossible to copy the tracking geometry data structure to device. All pointers that are internally used to either interconnect geometrical objects or resolve the polymorphism would be invalidated upon a relocation in memory.

Hence, unless the geometry instance is constructed directly on the GPU, it cannot be moved and used there. Geometry construction, on the other hand, is a complex task



(a) Fitting time vs. no. tracks for different floating point precision using the custom matrix inversion. A speedup of the GPU over the CPU can be seen for more than 1000 tracks.



(b) Fitting time vs. no. tracks on a Cori-V100 GPU for inter- and intra-track parallelisation. Intra-track parallelisation using shared memory shows a speedup for less than 1000 tracks.

Figure 5.1.: Results from the computing performance study in Ref. [177] on a Cori[1]-Haswell CPU and a Cori-V100 GPU.

that typically relies on data file input and it is therefore not desirable to perform on the GPU. Another frequent issue is the use of data structures with a *vector-of-vector* layout in memory. These arise when collections of data contain other data collections as elements. For example, a complete detector geometry will hold a collection of volumes (e.g. for sub-detectors), which will in turn each own a collection of geometrical surfaces representing the sensor modules. Per-module data structures like hits or measurements also typically come in a vector-of-vector layout. Data structures such as this do not form contiguous memory blocks and are therefore problematic to copy to device.

For the study in Ref. [177], the detector setup and input data layout was therefore strongly reduced in complexity, as shown in Figure 5.2, by making use of only one geometrical shape for the module surfaces and ensuring that tracks cannot miss surfaces. This way, each track would contain the same number of hits and C++ class polymorphism was not needed to resolve geometric properties, which also bypasses thread divergence at different surface shapes. Additionally, the magnetic field was kept constant, which reduces the amount of data that needs to be transferred to device and has the advantage that the field strength at a given position does not need to be determined by interpolation.

Another problem, that was solved in Ref. [177] by ensuring that every track had the same number of hits, is an a priori unpredictable size for the data memory allocation. This can occur, for example, when the number of outputs of an algorithm are determined at runtime by its input data and is difficult or impossible to estimate by other means. Situations like these frequently arise in HEP reconstruction algorithms that produce a collection of reconstructed data objects as outputs, such as clusters or track candidates. Addressing the memory for each of them within a contiguous block of allocated memory is complicated by the fact that the number of preceding elements cannot simply be

[1]The *Cori* architecture refers to the compute nodes of the NERSC supercomputer [180]

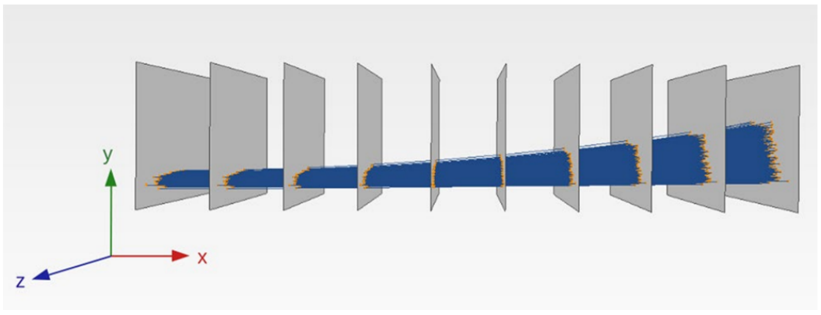


Figure 5.2.: Telescope geometry setup used for the performance study in [177] with the entire sample of 10 000 simulated single muons.

calculated from the number of inner vectors, but has to be determined by a prefix sum algorithm. In case the number of elements in each inner vector is altogether unknown before the allocation needs to be made, a precise memory allocation becomes impossible and other strategies need to be applied. Assuming the maximum possible number of elements for the size of each inner vector is wasteful and might lead to poor performance. Such an allocation can leave a large fraction of the memory unused and could, for example, degrade cache efficiency.

In the end, the large number of simplifications and also changes toward the ACTS core implementation in order to deploy just the GPU Kalman Fitter led to the decision to not include it in the main project repository. However, it also served as a valuable example for the following R&D projects.

5.3. Project Overview

In the beginning of 2020, a dedicated R&D effort was launched as part of the ACTS main project. It consists of several sub-projects that have since investigated critical aspects of a complete implementation of the ACTS tracking chain on an accelerator GPU device. The GPU R&D project was started in a stand-alone code base, minimising interference with the existing ACTS library. A guiding idea behind the design of the demonstrator chain has been to take the algorithms of the main project without making simplifying assumptions as to the complexity of the setup to which they can be deployed and find an implementation that runs as efficiently as possible on an accelerator chip. Furthermore, the complete chain will be implemented and studied, including steps that are more difficult to run efficiently on a GPU. This allows for a realistic characterisation of its performance and can identify issues ahead of deployment, such as inefficient data transfers between algorithms of the chain.

The main advantage of this approach is, that once implemented, the physics performance of the R&D chain is expected to match experiment requirements effectively by design and existing expertise in the collaboration on the peculiarities of the algorithms under study can be leveraged. A major downside might be, of course, that algorithms

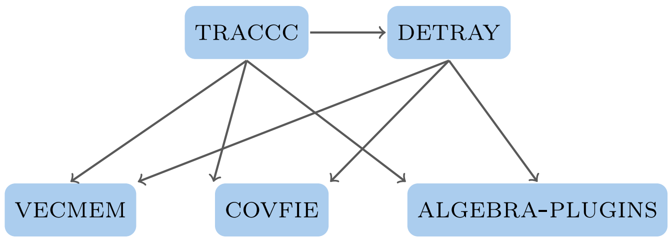


Figure 5.3.: Sub-projects of the ACTS R&D line and their interdependencies, which are indicated by arrows (*makes use of*).

initially designed and optimised for a CPU can show poor resource utilisation and overall performance when begin adapted for a different hardware platform. A modular or even heterogenous approach can prove advantageous in this regard, since it allows to either swap the implementation of a particular step in the chain against another, potentially more efficient one, or to run parts of it on the host. Especially considering a heterogenous approach, however, it remains to be seen if any of the algorithms under consideration performs badly on device to a degree, where the copying of data from device to host and vice versa would amortise a faster execution on the CPU. More likely, algorithms that do not provide a clear speedup or are even slower when run on the GPU, might still be left executing on the GPU to prevent the transfer of intermediate results back to host memory.

Moreover, a fully heterogeneous implementation comes with the risk that not just the algorithm, but the implementation itself cannot be adapted optimally to either host or device execution and will therefore potentially suffer from an additional source of performance degradation. At the same time, since tracking is embarrassingly parallel, host and device can both run reconstruction on different data at the same time and a common source code for host and device algorithms is a clear advantage in maintainability of the project. The most practical approach in this, which was chosen for large parts of the R&D project already, is likely a mixture of the two, namely to define building blocks or sub-routines of an algorithm that can be shared, which increases the code reuse and allows to adapt other parts to hardware specific behaviour at the same time.

The input data to the GPU tracking demonstrator chain will be simulated hits of charged particles in a given detector geometry and its output will be track objects that can be used in downstream reconstruction steps. The tracking chain will contain implementations for the steps that were outlined in Chapter 3, from clusterisation to ambiguity resolution, which will each share code between the CPU and GPU implementations to various degrees. Only the linear algebra and detector geometry implementations are a single source code implementation at this point. This may, however, still undergo some changes in the future, once the profiling and performance benchmarking campaign is underway.

The primary tracking demonstrator where all sub-projects come together and the chain is assembled, is the TRACCC project, as shown in Figure 5.3, which offers additional tooling for performance profiling and benchmarking. Here, the main steps in the tracking

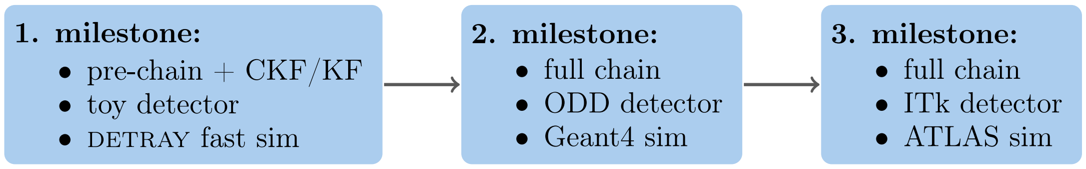


Figure 5.4.: Project milestones for the ACTS GPU R&D effort.

chain are implemented in a chain of algorithms and, finally, evaluated for their physics and computation performance.

The detector geometry, as well as the track state propagation in inhomogeneous magnetic fields, are handled by the DETRAY project. It will offer the same detail in the modelling of the geometry as ACTS and will also allow to convert existing ACTS tracking geometries directly to the DETRAY format. The COVFIE library [181, 182] can describe vector fields by type composition and is used to interpolate the magnetic field in DETRAY and TRACCC. The 4-, 6- and 8-dimensional linear algebra implementations needed mainly in the track finding and fitting steps are kept in the ALGEBRA-PLUGINS repository [183, 184], while VECMEM [185, 186] abstracts the memory handling between host and device.

Milestones for the completion of a first stand-alone demonstrator are in general oriented at the complexity of the detector geometry as well as the simulated data the chain can be successfully deployed on. More information can be seen in Figure 5.4. Its success in this case is measured by the correctness of the reconstruction chain, meaning its physics performance. Criteria under study here are the efficiency in tracking and seeding, while computing performance is only being monitored. A dedicated computing performance profiling and optimisation study is largely to be conducted after the completion of the final milestone and its outcome will be a major contribution to the overall project statement.

A first milestone, that has been reached by the TRACCC project in mid 2023, was the validation of the DETRAY *toy detector* on data generated in a fast simulation suite that was developed in the R&D project in the absence of a full Geant4-based simulation that could operate on a DETRAY detector description. The toy detector is based on the pixel section of the ACTS generic detector [187], and has been used mainly for algorithm development, as well as unit- and integration testing. Since the first steps in the tracking chain have initially been developed on a different simulated data set, this "pre-chain", meaning clusterisation to spacepoint formation in this context, is not yet connected to the track finding and fitting steps. The latter comprise the CKF and KF implementation, which are run instead on the DETRAY fast simulation in the toy detector. Regarding the split chain as well as the simplistic detector setup and data set, this first milestone serves mainly as a feature demonstrator.

The next step in complexity is the validation of the ACTS Open Data Detector (ODD) [166], which comes with a full geometry description in DD4Hep [168] and can therefore be used to produce a Geant4-simulated data set. The Geant4 simulation includes detailed material interactions, secondary particle production and propagation in an inhomogeneous magnetic field.

The ODD does, however, not have a native geometry description within the DETRAY project and has to be converted from the ACTS tracking geometry description to DETRAY. A conversion pipeline via `json` [188] files has been put in place to bridge the gap between the two projects until a full integration is completed. This milestone consequently showcases the full tracking chain run on a detector tracking geometry that was converted from ACTS and uses a realistic detector simulation as input and was reached workshop in early 2024 [189].

The final milestone will be the porting of the chain from the ODD to the ITk detector and ATLAS simulated data, the way they are used in the ATLAS offline reconstruction chain with ACTS. This milestone will hence comprise the final, fully realistic GPU demonstrator, which can subsequently be used for an in-depth performance study.

A first deployment of the GPU R&D project is envisioned within the ATLAS Athena framework, either directly or through ACTS, the specifics of which are still under development at the time of writing. Furthermore, TRACCC is considered for the EF-tracking GPU demonstrator and might consequently be adopted to run in the High Level Trigger in Run 4 [190, 191], if the demonstrator is successful.

5.3.1. Memory Management

A core functionality of any library that offloads to hardware accelerators is the memory management. In the ACTS GPU R&D project, this is done by the VECMEM library. The basic design principle behind VECMEM is based on the `C++17` polymorphic memory resource, which can be used in the `C++` Standard Library to steer the memory allocations of data containers.

Data containers, such as `std::vector` are an essential tool in in `C++` development, as they provide key data structures to the programmer and are hence heavily used throughout high energy physics reconstruction software. At the same time, a large portion of the `C++` Standard Library functionality in general cannot be used in device code, since it is, for example, lacking the required function decorators or will be attempting to allocate and own memory on the device. In VECMEM, memory resources were developed that facilitate the allocation of memory on hardware accelerators, which are at the same time composable, meaning they can be combined in order to model complex allocation schemes. Containers and views onto the allocated memory can give STL-container semantics in host and device code and help to abstract algorithmic code on different hardware.

A VECMEM view can be generated from a host container or allocated memory buffer and given to a device kernel as lightweight function argument to pass on the information about where to find the memory. Around this view, a device-side container can be built which exhibits the same interface as the host side container, thus facilitating the development of heterogeneous software.

CHAPTER 6

The Detray Project

A key ingredient to particle reconstruction algorithms is an accurate model of the detector geometry. It is needed to determine the position and local coordinate systems of the sensitive detector elements and passive detector material, in order to be able to pick up the correct measurements of particle crossings during the process of track finding and fitting. In addition, the error estimation in this part of the track reconstruction chain relies on the proper transport of the covariance matrix along the track.

For reconstruction, the geometry description is not required to the same detail as it is needed for detector simulation, where the position and type of material has to be known precisely to accurately simulate the physical interactions of a traversing particle. Therefore, both in ATLAS and ACTS, the tracking chain uses a dedicated geometry and material description, which yields a greatly simplified and hence more computationally efficient model when compared to software packages like Geant4 [164], TGeo [192] or DD4hep [168]. However, as discussed in Chapter 5, the ACTS tracking geometry description cannot be used in device code since it strongly relies on concepts like `C++` runtime polymorphism and a *vector-of-vector* memory layout. Additionally, not all of the `C++` Standard Library functionality that is used in ACTS can be adopted into the kernel code that is run on the GPU. It was therefore decided to reimplement the geometry description and the corresponding track state propagation, instead of using the ACTS core implementation for the GPU demonstrator.

The reimplementation has been done in the DETRAY project [193, 194], for which an initial version was available by early 2021 [195], but was at that stage still restricted to running on CPU. Furthermore, it did not yet provide a track state and covariance transport, an inhomogeneous B-field integration or a material description, which were added outside the scope of this thesis. However, the project already provided an implementation that was using tuple containers instead of runtime polymorphism and a geometry without a distinct *layer* class, as opposed to ACTS. This reduced the complexity of the

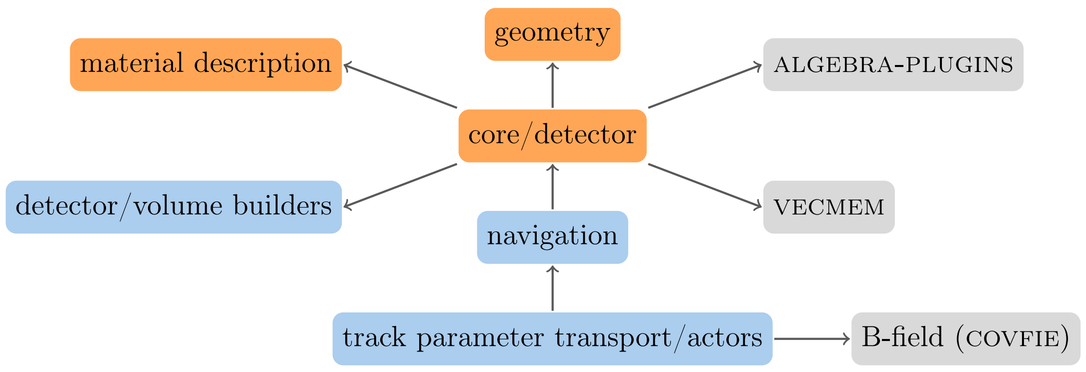


Figure 6.1.: Software architecture of the DETRAY core library. The orange nodes are components that are constructed on the host and then moved to the device as part of the detector. The grey nodes are provided by external libraries.

geometry description and especially the navigation through it greatly. The adaptation of this initial geometry description and navigation in DETRAY to a heterogeneous, GPU-ready implementation will be described in this chapter. DETRAY is developed by many authors and only those aspects to which major contributions were made as part of this thesis will be presented in the following.

The DETRAY project has been structured in layers, with the geometry description being the most basic. It is used by both the detector class and navigation layer to model the detector geometry and any other instances in which geometric properties are needed, such as grid acceleration data structures. The detector class in DETRAY is responsible for the memory management, which relies on the VECMEM library as an abstraction layer towards the concrete hardware backend. It is part of the core implementation, which also defines the container types and data access. A dedicated builder layer handles the construction of a detector object, separating this functionality from the detector class.

The navigation sits on top of the geometry and detector layers and provides the interface of the geometry and material access to the propagation. The propagation layer is responsible for the transport of track parameters and their covariances, offering customisation points through so-called `actors`, similar to ACTS. Among the actors is, for example, the material interaction implementation, which relies on the material description as part of the detector. The magnetic field implementation is provided to DETRAY by the COVFIE project, which allows to model vector fields for both host and device.

Additional functionality can be added via plugins, of which there are currently the linear algebra implementations provided by the ALGEBRA-PLUGINS project and the `SVG` based geometry visualisation tool [196], using the ACTSVG library [197]. A dedicated IO library has been added to interface to the core builders, so that a detector can be read and written from and to data files, as described in Section 6.4. The DETRAY unit- and integration tests, for both host and device code, as well as any validation and benchmark code, is available in the optional *tests* library and checked as part of the github-based Continuous Integration [198]. The core project structure is shown in Figure 6.1.

6.1. Geometry Description

6.1.1. Implementing a GPU-friendly Detector Geometry

In order to maintain compatibility, the DETRAY geometry description follows the same conceptual design as the ACTS geometry description, which is explained in Chapter 4. This is key, since the main method to load a tracking geometry into DETRAY will be the conversion via the ACTS IO chain, as discussed in Section 6.4.

Since the geometry will therefore be constructed in host code and then moved to the device at the beginning of a reconstruction job, a number of restrictions have to be observed for the code design, as already motivated in Chapter 5. Following that discussion, one of the design principles in the DETRAY geometry description has been to avoid any dynamic polymorphism in the geometry description. This means, that abstract base classes or base classes containing any virtual methods cannot be used to define interfaces, for instance, for the properties of the geometric shapes of detector surfaces. For one, this makes the abstraction of the geometry description more difficult, inviting direct usage of specific classes that are part of the geometry implementation. Such a design can lead to an increased risk of coupling between internal components in DETRAY, but also to client code such as TRACCC. Increased coupling between separate units is commonly considered detrimental to code maintainability and therefore likely to increase the time spent by developers on code modifications and extensions.

Another complication that results from avoiding dynamic polymorphism has to do with the lack of dynamic heterogeneous containers in `C++`, where instead instances of different types are held in a single homogeneous container by pointers to a common base class. Without the definition of a common base class which could dispatch the individual functionality of an instance of a specific type, a distinct container per type is needed. For example, a detector volume that contains surfaces of different shapes now has to define a container for each shape type, the number of which is determined by the requirements of the respective detector geometry. In order to not permanently restrict the number of shapes that can be used to describe a detector by hard-coding the required containers or function overloads, DETRAY makes use of *tuple containers* and *variadic templates* in the `C++17` standard [199] instead. Tuples are containers that can, hold elements of different types, often by exploiting recursive inheritance. However, these types need to be fully defined at compile-time and can also only be accessed using information that is available at compile-time, thus posing strong restrictions on further code design.

For example, while an element in a `std::vector` can be accessed by an index that can be calculated by an algorithm according to runtime conditions, fetching an element from a `std::tuple` can only be done by an index that has to be fixed before the algorithm is actually run. Instead, all different options of fetching an element from the tuple are compiled and an index that depends on runtime conditions can then be compared against every option. Consequently, all code paths that depend on the different shapes of detector surfaces have to be fully defined and available at compile-time, as well. The advantage of using tuples here is, that the generation of the code paths for the different options can be done by the compiler using generic programming, instead of writing them directly into the code. Due to the resulting extensive application of class and function templates, DETRAY is currently a header-only library. The number of code paths that will be generated is mainly influenced by the type of the detector that is being compiled.

In order to be able to relocate the detector in memory easily, its underlying data structures cannot be interconnected using pointers, as is often done in tree-like data structures that model the hierarchical organisation of a geometry. This is particularly useful for placing multiple instances of the same object without duplicating their data. In DETRAY, the data are instead linked by indices that encode offsets into the underlying data containers, thus providing data structures that are invariant when moved in memory, while still allowing to avoid data state duplication. DETRAY has defined a dedicated index type (`detray::dtyped_index`) for its geometric objects that contains both a type id, in order to choose the correct tuple-based code path, as well as an offset into the collection of objects of the same type to fetch a concrete instance.

Another design principle for DETRAY was to pool the memory allocations needed for the geometry data globally in an effort to reduce the complexity of the host-device memory management. This resulted in the number of containers that make up the data state of a detector, and hence need to be moved from host to device, being in the order of tens instead of hundreds. Moreover, this quantity scales with the number of different types needed for the detector, instead of with the number of volumes it contains.

Last but not least, the complexity of the code base needs to be considered, since the geometry description and propagation implementation are among the most complicated tasks in tracking and the DETRAY core implementation already spans in the order of 30 000 lines of code. Code duplication between the host and device implementation should therefore be minimised to ease maintainability. Sacrificing ultimate performance, the DETRAY core library therefore provides a single implementation that can be used in host as well as device code and no specialised implementations exist for the different compute backends.

Apart from the fact that backend specific functionality cannot be used, this brings also some further restrictions on the `C++` code that can be applied. For example, not only dynamic polymorphism, but also function pointers and dynamic memory allocations are forbidden in SYCL kernels [200] and are consequently not used anywhere in DETRAY. DETRAY classes can therefore be used in SYCL code with the corresponding VECMEM memory handling without further adaptations. Furthermore, the utilities and algorithms from the `C++` Standard Library are not specified as device functions and can thus only be used in kernel code under certain circumstances. If a function is declared as `constexpr`, meaning completely defined at compile-time, an experimental feature in CUDA allows to still call it from kernel code. Other functionality has to be re-implemented in DETRAY or taken from device-compatible libraries, like *Thrust* [201].

An important aspect of the design of DETRAY that follows from avoiding dynamic memory allocations is, that any algorithms in DETRAY will need all of the memory they can make use of allocated before the algorithm can be run, even if it targets host execution. This also includes any memory for an a priori unknown number of intermediate results.

6.1.2. Data Containers and Memory Management

DETRAY relies on the VECMEM project, described in Chapter 5, as an abstraction to the memory allocations for its basic data containers. The initial adoption of VECMEM in DETRAY featured the ability to equip any class with either a host- or a device-specific implementation of a given container, as well as an additional, custom VECMEM *view* implementation from which a device-side instance of the class could be produced. This way, it became possible to use the DETRAY types in both host and device code and transfer their data states to the GPU. In the following, the generalisation of this implementation will be discussed, which both reduced the amount of extra code that was needed to define a VECMEM view for a class, and allowed to deploy generic containers for the detector that can handle the memory copies for their elements when moved to device.

The most widely used type of container in DETRAY is the `C++` *vector*, which dynamically allocates memory to hold a number of elements of the same type. As discussed before, VECMEM offers a host and a device implementation of a vector container, which in turn can be given to a DETRAY class as a template parameter argument to be able to use it either in host or device code. The interfaces for accessing and iterating over the elements in the vector is the same between both versions, so that no further adaptations to the DETRAY implementation are necessary. Apart from the `vecmem::device_vector`, another key container type is the tuple, as discussed above. The tuple implementation for DETRAY was initially taken from the Thrust library, but later replaced by a custom implementation [202], since the `thrust::tuple` type cannot hold more than ten elements. The `detray::tuple` works in both host and device code, since it does not rely on memory allocations directly.

Every DETRAY class that needs to manage memory has to be given a VECMEM memory resource, which it passes on to its underlying VECMEM containers. The memory resource determines the memory allocation scheme, for example, to switch between host or device memory allocation. From any such DETRAY object, both a non-owning view, as well as a memory-owning buffer type can be obtained by calling `detray::get_data()` and `detray::get_buffer()`, respectively. The view is small in memory and can be passed directly as an argument to a device kernel, communicating where the memory was allocated. If the object was allocated in *unified memory* [172], which is a type of memory that is effectively accessible from both host and device, calling `detray::get_data()` and passing the view to the kernel is enough to access the data on device. For an explicit copy to the device, the buffer representation has to be used instead.

The creation of a VECMEM data buffer from a DETRAY object additionally requires a VECMEM *copy*-object, as well as another memory resource that steers the allocation of the buffer. Most commonly, the buffer will be allocated in host-inaccessible device memory and the data from the DETRAY instance will be copied over using the copy object. An additional flag can be passed to make the copy operation either *synchronous*, meaning host-side execution is halted until the copy operation is finished, or *asynchronous*. From the buffer object, a view can then be obtained as well, which passes the information about the copied data to the GPU kernel.

The following example, adapted from the DETRAY tutorials, shows how to make the data state of an entire detector object available to the device, once using unified memory and once with an explicit copy:

```cpp
// Detray include(s)
#include "detray/test/common/build_toy_detector.hpp"

// Vecmem include(s)
#include <vecmem/memory/cuda/device_memory_resource.hpp>
#include <vecmem/memory/cuda/managed_memory_resource.hpp>
#include <vecmem/memory/host_memory_resource.hpp>
#include <vecmem/utils/cuda/copy.hpp>

int main() {
    // memory resource(s)
    vecmem::host_memory_resource host_mr;          //< host memory
    vecmem::cuda::managed_memory_resource mng_mr; //< unified memory
    vecmem::cuda::device_memory_resource dev_mr;  //< device memory

    // CUDA unified memory

    // Create toy detector with vecmem CUDA managed memory resouce
    auto [det_mng, names_mng] =
        detray::build_toy_detector<algebra_t>(mng_mr);

    // Get the detector's data view directly, as this memory is
    // already accessible to device
    auto det_data = detray::get_data(det_mng);

    // Pass the view and call the 'print' example kernel
    detray::tutorial::print(det_data);

    // Explicit copy to CUDA device memory

    // Create toy detector in host memory
    auto [det_host, names_host] =
        detray::build_toy_detector<algebra_t>(host_mr);

    // Helper object for performing memory copies to CUDA devices
    vecmem::cuda::copy cuda_cpy;

    // Copy the data to device-side buffer (synchronous copy)
    auto det_buff =
        detray::get_buffer(det_host, dev_mr, cuda_cpy, detray::copy::sync);

    // Pass the buffer's view to the 'print'-kernel
    detray::tutorial::print(detray::get_data(det_buff));
```

The device kernel can then create the device-side object from the data view that was passed as one of its arguments. Using the example of the `detray::tutorial::print` kernel again:

```
/// Kernel that prints some information about a given toy detector
__global__ void print_kernel(
    detector<toy_metadata, host_container_types>::view_type det_data) {

    // Setup of the device-side detector
    detector<toy_metadata, device_container_types> det(det_data);

    // Use single thread for this example
    if (threadIdx.x + blockIdx.x * blockDim.x > 0) {
        return;
    }

    // Print some statistics
    printf("Number of volumes: %d\n", det.volumes().size());

    [...]
}

void print(
    detector<toy_metadata, host_container_types>::view_type det_data) {

    // Run the tutorial kernel
    print_kernel<<<1, 1>>>(det_data);

    // CUDA error check
    DETRAY_CUDA_ERROR_CHECK(cudaGetLastError());
    DETRAY_CUDA_ERROR_CHECK(cudaDeviceSynchronize());
}
```

The local type definition `view_type` is key to this setup, which is being used in the example above to define the type of the data view argument to the kernel. It contains the entire information on the memory held by all containers in the class, and is available from the type itself to allow a certain degree of introspection. In the cases where DETRAY classes contain other classes that define a custom view type or contain multiple different vectors, these *sub-views* can be bundled into a tuple and later be unpacked and passed to the constructors of the members once the device type gets constructed.

To this end, the DETRAY *multi-view* type (`dmulti_view`) has been added. It allows a centralised and structured handling of hierarchical views, wrapping and incorporating the VECMEM view types. A similar implementation has been added for the buffers, including the stand-alone `detray::get_buffer()` function, which performs the allocation and copy to a different memory location. For this, it makes use of the view type of a *viewable* DETRAY class, which can be determined by dedicated type traits.

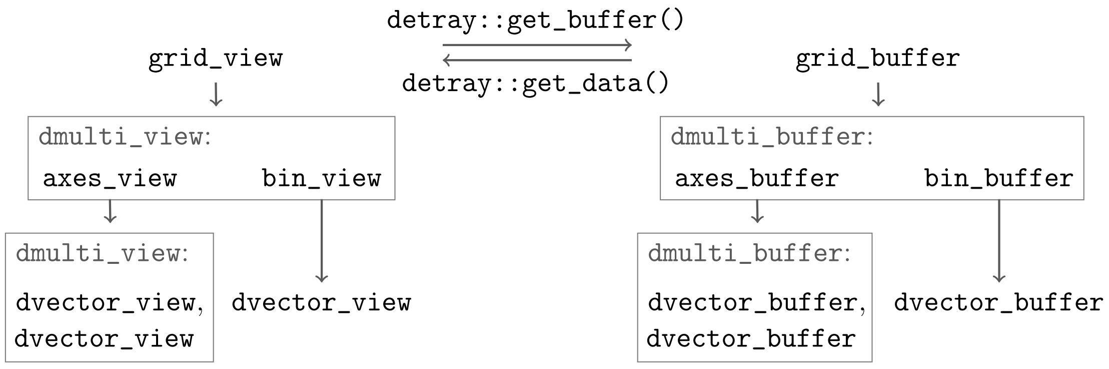


Figure 6.2.: Example of a hierarchical structure of the sub-views and -buffers using the `dmulti_view` and `dmulti_buffer` to be able to define a single view type per class.

The view of a class allows to access the sub-views down to the simple VECMEM vector views (wrapped by `dvector_view` to tag them as views within DETRAY), with which a VECMEM vector buffer can be generated and filled. This is done recursively, so that the resulting `dmulti_buffer` automatically exhibits the same hierarchical structure as the initial class view. A similar recursion is used to retrieve the view type of the buffer, ensuring that it also follows the same structure as the original view type of the class. This makes it possible to give the buffer view to the same constructor in device code as the original view type of the class, but this time constructing the device-side instance from the data held by the buffer. This structure is depicted for the DETRAY `grid` class as an example in Figure 6.2.

Based on this view and buffer handling, two generic containers, called *data stores*, have been implemented, which can infer the view and buffer types of their elements and generate their own view and buffer types as the corresponding aggregation of those. Upon construction in device code, they will hand down the correct sub-views to every respective element they contain. The only condition is, that each element must define a single view and buffer type, which are accessible as local type definitions called `view_type` and `buffer_type`, implying that an element that is itself a collection of viewable DETRAY objects needs to provide an abstraction of the views of its own elements. This way, VECMEM-viewable DETRAY objects can be copied to the device automatically by copying the container that holds them, which is central to the maintainability of the detector class.

In addition to the view and buffer handling, the data stores also provide contextual calls to retrieve their elements. The data context will in the future be used to access data in the geometry according to conditions data, such as calibrations or detector alignment. The current CPU version wraps an index that can be used as an offset to a memory region where data has been loaded according to different *Intervals of Validity* (IoV), which basically encode the duration for which the particular piece of conditions data needs to be taken into account during reconstruction. In the device code, the sub-view corresponding to the updated data collection is simply switched in the detector view instance before passing it to a given kernel [203].

The most important of the two data stores is the `multi_store`, since it can hold collections of different data types, wrapping a `tuple_container` internally which allows for the type id based polymorphism discussed in the previous Section. The `tuple_container` as a separate container [204] presents an adapter around an underlying tuple type (now `detray::tuple`). It provides the local view type definition, as well as a method that allowed to centralise the *unwrapping* of the tuple according to the type id. Given a functor, which is a class that can be called like a function, that contains some specific functionality and a `dtyped_index`, it calls that functor on a particular tuple element.

This has subsequently transitioned into the `visit` method that can be accessed through the `multi_store` class interface and allows to call a functor on, for example, an acceleration structure in a detector volume or the mask of a particular detector surface, see below. Furthermore, the `multi_store` provides functionality to connect the type id with the contained element types, and in case the elements are containers themselves, their value types. This is particularly useful in conjunction with the DETRAY detector, for example, to statically match a particular mask or material type to its type id:

```
/// Match the types to their position in the multi store
enum class type_ids {
    e_double = 0u,
    e_int = 1u,
    e_float = 2u,
    e_size = 3u, //< number of registered types
};

/// Multi-store that keeps three different vector-based collections
using store_t =
    detray::multi_store<type_ids, vecmem::vector, float, int, double>;

/// Get the value type of the first data collection, which is 'float'
using value_type =
    typename store_t::template get_type<type_ids::e_float>;

/// Get the type id of the collection that contains integers
constexpr type_id id{typename store_t::template get_id<int>};
```

The second data store, the so-called `single_store`, is in essence a special case of the `multi_store` in so far as it only holds a single element or data collection instead of multiple. However, it presents a simpler implementation than a `multi_store` with only one element type.

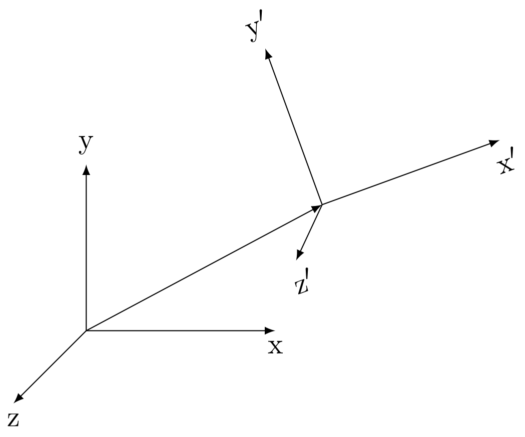


Figure 6.3.: A geometric object is positioned by the rotation and translation of its local axes.

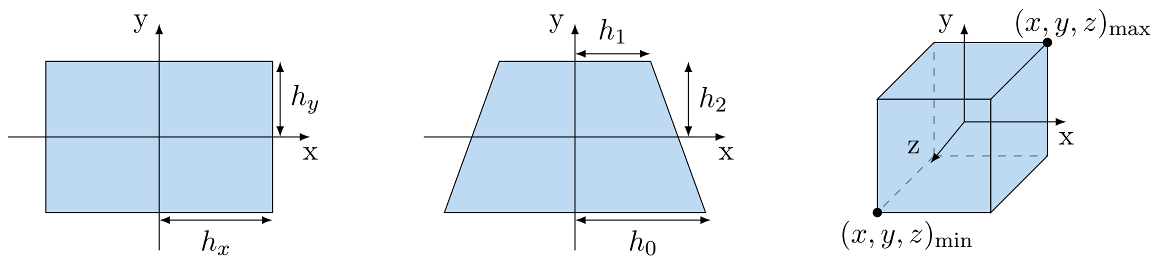
6.1.3. Geometric Object and Shape Implementation

A geometric object in DETRAY, such as a surface or any other class with geometric properties, generally consists of a prescription of how to position it in the global coordinate frame of the detector, as well as a geometric *shape*. The global frame of the detector is a three-dimensional Cartesian coordinate system, in a silicon tracker geometry usually with the z-axis aligned along the beamline. The positioning of a geometric object is performed by a rotation around the origin, followed by a translation to its final position, as shown in Figure 6.3. This is described in homogeneous coordinates, meaning a single four-dimensional transformation matrix is used, where the first three columns are the directions of the local x, y and z-axes and the last column contains the translation **t**:

$$\mathbf{T}_{3D} = \begin{bmatrix} x_0 & y_0 & z_0 & t_0 \\ x_1 & y_1 & z_1 & t_1 \\ x_2 & y_2 & z_2 & t_2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6.1}$$

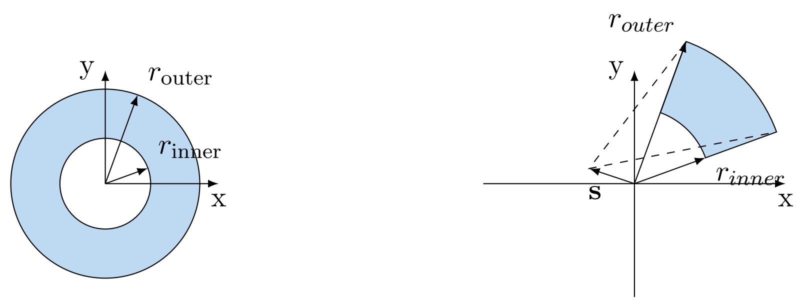
A position vector consequently consists of three coordinates and a *one* in the last entry, so that it can pick up the translation during the transformation, while a direction vector will contain a *zero* in the last element and will be invariant under translations. This means, that a simple four-dimensional matrix multiplication can be used to perform the transformation to and from the local Cartesian frame, which can be advantageous for vectorization on the host. The projection from Cartesian coordinates into the local coordinate system, for instance into polar or cylindrical coordinates, is determined by the shape of the object.

Instead of abstracting the functionality of a geometric shape behind an interface, DETRAY requires a template parameter on any class that has geometric properties, which is substituted by a concrete shape type during compilation. The shape types are defined by a number of small, stateless `structs` that contain local definitions of geometric properties, such as the local coordinate system type, as well as shape-specific methods, such as how to obtain the shape's vertex positions or how to check whether a point lies



(a) `rectangle2D` (b) `trapezoid2D` (c) `cuboid3D`

Figure 6.4.: Shapes based on two- and three-dimensional Cartesian frames. The local coordinates are (x, y) and (x, y, z), respectively.



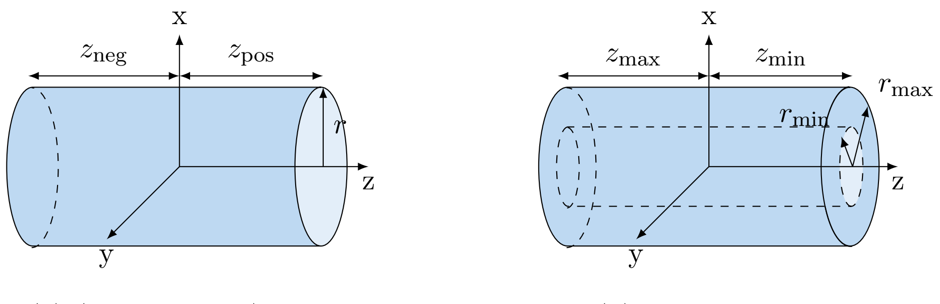
(a) `ring2D` (b) (stereo) `annulus2D`

Figure 6.5.: Shapes based on the polar coordinate frame, the local coordinates are (r, φ), accordingly. The focal point of the annulus shape can be shifted from the origin by $\mathbf{s}$, and with it the boundaries in φ, to model the ITk stereo annulus shape in Figure 2.7.

inside or outside of an object of that shape. In this manner, classes in DETRAY can be equipped with a local geometry in a straight-forward, yet flexible way. Classes that require a shape are first and foremost the boundaries of the surfaces that make up a detector (called *masks*), but shapes are also used for other geometric objects, for example, to define the surface grid types for the acceleration structures in the navigation or for bounding volumes.

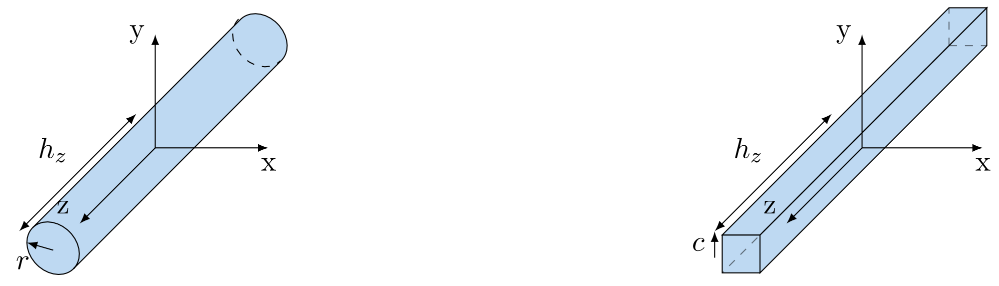
A position that has been transformed to the local coordinate frame can be checked to lie inside or outside of the shape by comparing it to a number of boundary values that are unique to every shape. These boundary values are provided by the concrete geometric object, usually as part of a mask. The local coordinate system implementations are separate from the shapes, so that they can be reused between the different shapes. The most important shapes, which are based on their ACTS counterparts, together with their local coordinate definitions, are displayed in Figures 6.4–6.7.

Apart from this, a shape allows to calculate its measure, which, depending on its



(a) (`concentric_`)`cylinder2D` (b) `cylinder3D`

Figure 6.6.: Shapes based on cylindrical coordinate frames. The two-dimensional cylinder surfaces (a) use local coordinates $(r \cdot \varphi, z)$ and (φ, z) for the concentric cylinder, which only allows translations along the z axis. The three-dimensional cylinder (b), which is used for grids, uses cylinder coordinates (r, φ, z).



(a) `line_circular` (b) `line_square`

Figure 6.7.: Shapes based on the line coordinate frame [205], based on the ACTS line surface implementation. The local coordinates are the radius of closest approach, signed by whether the position is on the positive or negative side of the z-axis, and the z-coordinate $(\pm d_r, z)$.

dimension, is either the area or volume defined by the points that lie inside the shape. For every shape, the centroid and minimal local bounds can be calculated, as well. The minimal local bounds are the two defining points of an axis aligned bounding box in the local Cartesian frame of the geometric object, before projecting to the local coordinates. Additionally, the vertices of most shapes can be returned. However, this is only available on the host so far, since this method returns a dynamically allocated vector of points.

A detector geometry is defined using two-dimensional shapes exclusively, since for the purpose of navigation these can be composed to higher-dimensional objects, like detector volumes, by simply aggregating the boundary surfaces. Furthermore, the bound track parameter and covariance transport handling, introduced in Chapter 3, is based on two-dimensional local positions. Three-dimensional shapes are currently used for the definition of some grid types, which need a way to project global positions onto their axes to perform a bin lookup, and bounding volumes, which should be equipped with specialised intersection methods, instead of checking all boundary surfaces separately.

As already mentioned, the class that straps a shape and a set of concrete boundary values together with some additional linking information between geometric objects used for geometry navigation, is the `detray::mask`. It thus represents the data state of the surface boundaries that is held by the detector as part of a `multi_store` instance.

To obtain mask types of different shapes with a given linear algebra implementation one can use, for example:

```
using algebra_t = detray::array<float>;

/// Rectangular boundary values
using rectangle = detray::mask<detray::rectangle2D, algebra_t>;
/// Concentric cylinder boundary values for portals
using cylinder_portal =
    detray::mask<detray::concentric_cylinder2D, algebra_t>;
/// Boundary values for disc shaped portals
using disc_portal = detray::mask<detray::ring2D, algebra_t>;
/// Boundary definition for a straw tube
using straw_tube = detray::mask<detray::line_circular, algebra_t>;
```

6.1.4. The Central Detector Class

The DETRAY detector class (`detray::detector`) is the owner of all geometry and material related data. It is responsible for handling host-device memory transfers via the view and buffer types described before. To this end, it uses both the `single_store` to hold all transform matrices of the various geometric objects (called *transform store*), as well as three distinct types of `multi_store`s, which contain the surface masks, material and acceleration structures (called *mask store*, *material store* and *accelerator store*, respectively). A number of plain VECMEM vector containers hold additional data that describe the linking between the aforementioned detector components via type ids and indices.

The different mask, material and acceleration structure types that a particular detector knows and holds collections for in its data stores, as well as the type ids that are assigned to them, are detailed in a separate *metadata* type. A metadata type can be used to customise the detector type for a given experiment. A number of metadata implementations already exist to model different detector setups, ranging from the DETRAY toy detector to the ATLAS ITk.

Within such a custom metadata implementation, the types and ordering of the data collections in the `multi_store`s can be steered, so as to only generate those code paths in the later navigation and propagation implementation that are really required by the detector setup or in order to add new, custom types for a specific experiment. The ITk, for example, should not be compiled against track parameter transport to the sense wires of a wire chamber detector. Conversely, a telescope detector setup will not need the unique stereo annulus shape that is used in the ITk strip endcaps. This reduction to

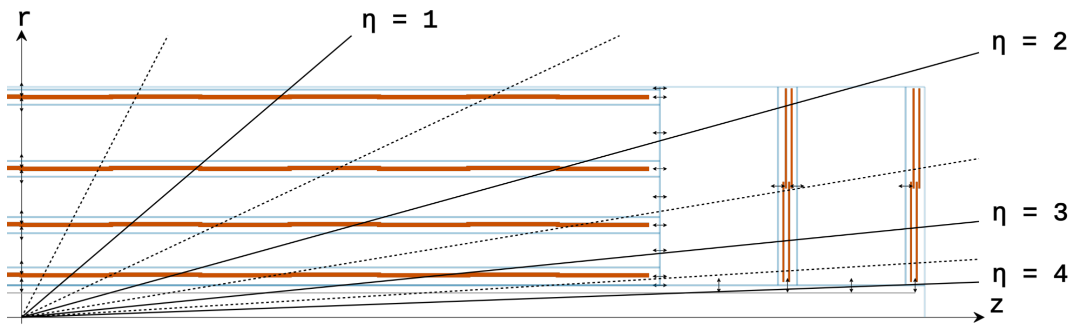


Figure 6.8.: The surfaces of the DETRAY toy detector with four barrel layers and two endcaps, shown in rz-view for positive z. Volume boundary surfaces (portals) are shown in blue, while the sensitive surfaces are red and passive material surfaces are grey. The linking between neighbouring volumes is indicated by arrows on the portals.

the required components can also reduce compile time and boost maintainability for an experiment, because it decreases the number of dependencies per detector.

Alternatively, DETRAY provides the `default_metadata`, which contains all mask shapes, types of material description and acceleration structures that are available in the library. Since the type id enumerations are detailed as part of the metadata, the data of any detector can be filled into the default metadata, just in a different position in the data stores and by leaving data store collections that are not needed to be empty. This is handled automatically during detector construction. The default metadata is, for example, used extensively in TRACCC so that a single algorithm can be called for all possible detector setups.

The construction of a detector instance is handled separately from the `detector` class itself, using a chain of dedicated builders and factories that can be composed according to the specific requirements of the detector under construction. After construction, a detector instance cannot be modified anymore and only gives read-access to the data it contains, so as to not break any of the intricate linking between its components. This is especially important in the light of sorting requirements or possible data deduplication, where single data instances may be shared between many components of the detector.

The composition of the DETRAY detector closely follows the design of the ACTS tracking geometry, from which it will be obtained via a dedicated conversion step. A detector is conceptually divided into a number of fully connected volumes, which represent different navigation domains when transporting a track through the detector geometry. The tracking volumes are defined by their boundary surfaces, which are also called portals since they allow to exit a volume and provide the linking information to enter the next volume. Within the volumes an arbitrary number of sensitive detector elements and passive material surfaces can be contained. Figure 6.8 shows the DETRAY toy detector as an example.

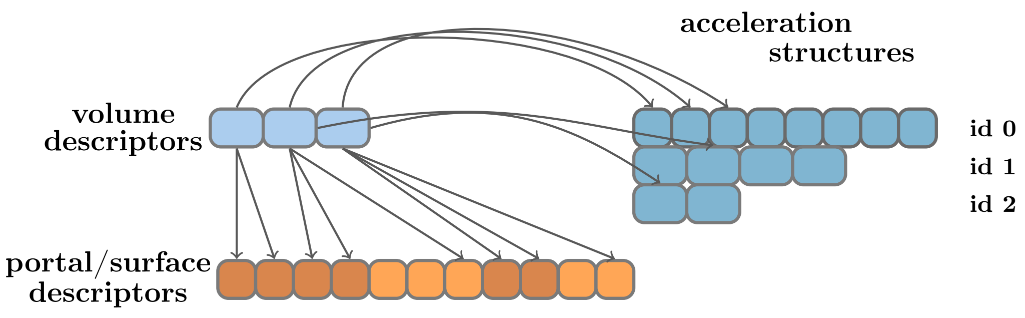


Figure 6.9.: The DETRAY volume descriptor links the data of a tracking volume: Portals (dark orange), passive and sensitive surfaces (light orange) and acceleration structures. The acceleration structure container to the right is a `multi_store` type container with different acceleration structures in every entry, addressed by different type ids. Volume material is not included in this graphic.

However, in order to avoid a vector-of-vector data layout, the various surfaces cannot be held by the volumes directly. Instead, the detector holds a vector container of *volume descriptors*, which exclusively contain indexing information on the volume constituents. They provide the volume's id, signalling whether it is for instance a cylindrical or cuboidal volume, and the volume's index, which is the position of the volume descriptor in the detector volume container and can be used to retrieve a particular volume from the detector. Apart from that, the volume descriptor holds the links for the volume data. These comprise an index to retrieve the volume's transform from the detector transform store, a typed index for possible volume material, as well as the accelerator link, which is used to link the volume to the acceleration structures it contains. Section 6.2 contains an explanation of how the accelerator link is structured.

Finally, the volume's surfaces are linked to the volume as index ranges in a detector container that holds all *surface descriptors* for the entire geometry globally. Within this *surface lookup* container, the surfaces of a given volume need to be arranged in a contiguous range, so that they can be referenced by the volume descriptor with start and end indices. Within that range, the surface descriptors additionally need to be ordered into sub-ranges, which allows, for example, to address the volumes portals separately from the contained sensitive detector surfaces. This is depicted in Figure 6.9. A volume descriptor also allows to address the entire range of its surfaces by finding the minimum and maximum index of all of its surface subranges, as well as transforming the index of a particular surface descriptor from its global index in the detector surface container to a *volume-local* one. A volume-local index counts the index relative to only the surfaces that are contained in the particular volume, starting from zero.

The way the data contained in the detector stores is linked to form a conceptual detector tracking surface, be it a portal, sensitive module or passive material surface, is similar to what is done for the volume descriptors. The aforementioned surface descriptors also

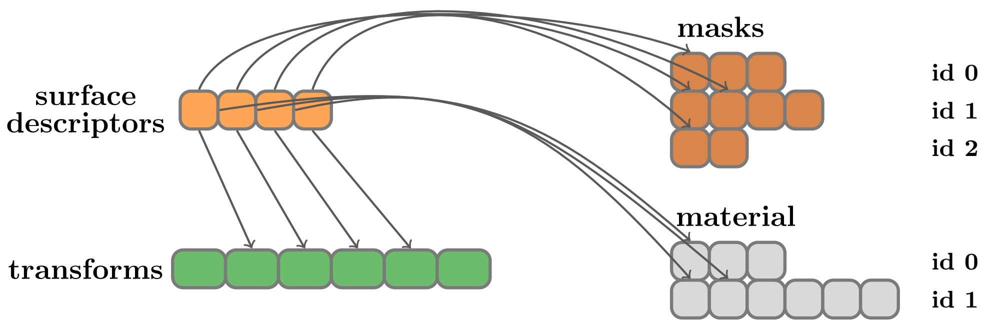


Figure 6.10.: The DETRAY surface descriptor links the data of a surface instance: Transforms, masks and material.

hold a surface id and a global index, but this time the id encodes whether it is a *passive*, *sensitive* or *portal* surface. The index is again used to fetch a given surface descriptor from the detector surface lookup container, which is why it has to relate to the global surface count instead of the volume-local one.

Other links in the surface descriptor connect it to its transform, masks and material, as shown in Figure 6.10. The mask and material links are *typed indices*, meaning they contain both the information on the mask shape and material type to compare against the different collections in the mask and material stores, as well as the index or index ranges into the specific collection, to be able to fetch the concrete mask or material instances that belongs to the surface. Together with the index of the volume to which the surface belongs, as well as some optional user defined information, the surface id and surface index form the global identifier of an individual surface. This global identifier is bitwise encoded on a single 64 bit integer that can be used as a hash value for the particular surface, the same way as it is done in the ACTS `GeometryIdentifier`. However, since the information encoded in the two identifiers is not the same between DETRAY and ACTS, for example due to the global surface index in DETRAY, they cannot be used interchangeably. Considering also the different number of portal surfaces in a DETRAY detector compared to an ACTS detector, as well as different sorting in the global detector containers, inferring the DETRAY identifier from the ACTS one and vice versa is in general infeasible.

Surfaces in DETRAY are therefore associated to their *source* with another 64 bit integer value that can be filled with the ACTS identifier of the corresponding surface in the ACTS detector, or any other source identifier. By virtue of the surface lookup container in the DETRAY detector, any DETRAY surface descriptor can be fetched using its global index, surface identifier or source identifier. The latter results in a search through the container to find the matching surface descriptor, while the former two access methods can be handled in constant time. The source link is not part of the surface descriptor itself, however, since it is not needed internally in DETRAY. Instead it is simply saved together with the surface descriptor as part of an entry in the surface lookup container.

Since the surface identifier hash is encoded in only 64 bits and the rest of the descriptor consists only of a few additional indices, it is in total only 128 bits large. This makes the surface descriptor sufficiently small to be handed through the rest of the code directly in many places where otherwise only the identifier would be used. Doing this potentially saves an additional indirection into the detector surface lookup container in order to fetch the data linking information that is only part of the surface descriptor, but not of the surface identifier itself. The direct storage of a surface descriptor is done, for example, in the surface grids which are used to find detector surfaces close to a track position.

Tracking Volumes and Surfaces

Since the concrete linking between the detector components is complex and requires a deeper understanding of the DETRAY detector as a data structure, direct usage of the volume and surface descriptors in downstream code is discouraged. This could also lead to coupling of the internal detector implementation to client code, such as TRACCC, which would make changes to the DETRAY code more difficult to deploy.

In order to provide a more intuitive, object-oriented interface to the detector volumes and surfaces, dedicated `tracking_volume` and `tracking_surface` classes have been added. They are implemented separately from the `detector` class to be able to switch them out easily or to deploy multiple different, domain specific versions in the future.

These classes are constructed as a facade around a volume or surface descriptor and the detector object to which the respective descriptor belongs. The detector holds the global data containers, while the descriptor is used to fetch the correct data instances, such as the masks or transform. Based on these pieces of data, the stand-alone surface and volume classes offer the basic geometric functionality, as well as some extended tracking functionality which is otherwise not part of the detector implementation, such as the generation of the Jacobians that are needed for covariance transport or the projection of track parameters into the surface local coordinate system.

In the following, a few examples are shown for tracking volumes:

```
// Get an example detector
vecmem::host_memory_resource host_mr;
const auto [det, volume_names] =
    detray::build_toy_detector<algebra_t>(host_mr);

// Get a tracking volume facade for volume no. 3
const auto& vol_descr = det.volume(3);
const detray::tracking_volume volume{det, vol_descr};

// Example: Iterate through the volume's portal surfaces
for (const auto& pt_descr : volume.portals()) {
    std::cout << pt_descr.index() << std::endl;
}
```

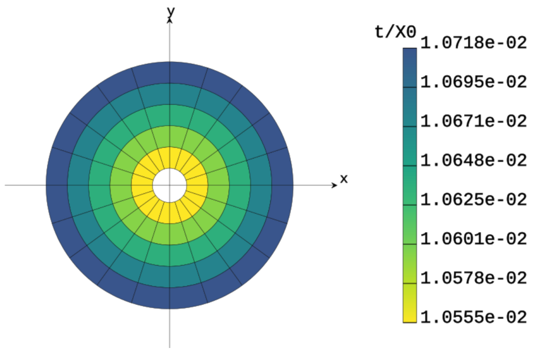
and for tracking surfaces, using the same example detector instance:

```
// Geometric context for data store access, e.g. for alignment
using detector_t = detray::detector<detray::toy_metadata>;
auto gctx = detector_t::geometry_context{};

// Get a tracking surface facade for surface no. 42
const auto sf_descr = det.surface(42);
const detray::tracking_surface surface{det, sf_descr};

// Do some geometry work
const point3_t centroid = surface.centroid(gctx);

const point3_t pos{4.f, 7.f, 4.f};
const vector3_t dir{0.f, 0.f, 1.f}; // < needed for line shapes
const point3_t local = surface.global_to_local(ctx, pos, dir);
const point3_t global = surface.local_to_global(ctx, local, dir);

// Do some tracking work
free_track_parameters<detector_t::algebra_type> free_param{
                pos,
                0.f * detray::unit<scalar_t>::s,
                10.f * detray::unit<scalar_t>::GeV * dir,
                -1.f * detray::unit<scalar_t>::e};

const auto jac = sf.free_to_bound_jacobian(gctx, free_param);
```

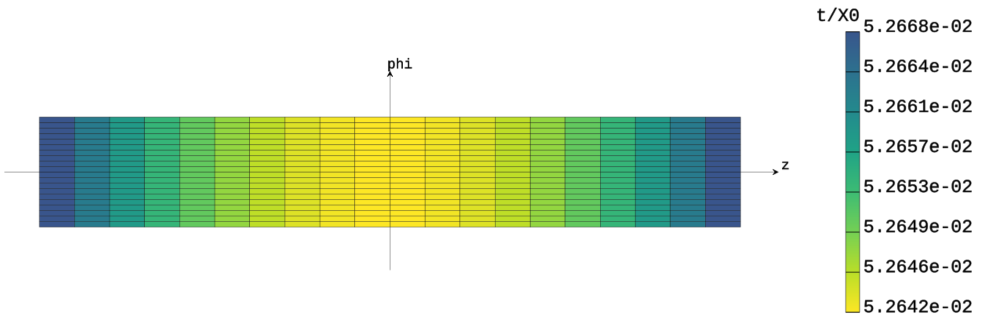
Some of these methods, especially those of the tracking surface, require shape or material specific information. This is the case, for example, for the `global_to_local` and `local_to_global` methods shown above. For this, the tracking surface internally uses the mask or material link that is part of the surface descriptor in order to execute the corresponding functor via the `multi_store` visit functionality. In order to run custom functionality that requires either the surface mask or material, dedicated `visit_mask` and `visit_material` methods were added to the tracking surface. These are also handy when multiple shape or material dependent calls need to be made on the surface, because these can then be bundled in a single functor for which the unwrapping of the `multi_store` tuple container needs to be performed only once.

6.1.5. Material Description

The initial homogeneous material description and interaction implementation in DETRAY was not developed as part of this thesis, however, it was used to later provide the material maps, which are based on the DETRAY grid implementation, which is detailed in Section 6.2. A piece of homogeneous material is described as either a *material slab* of a given thickness or a *material rod* of a given radius. The latter type is only used in the context of line surface shapes, which are outside the scope of this thesis. The material itself



(a) Endcap material map



(b) Barrel material map

Figure 6.11.: Examples of material grids in the toy detector, which are used to store material maps. Shown is the material thickness t per radiation length X_0 in every bin.

is parametrised by the relevant material properties, such as the charge number Z or radiation length X_0. A number of predefined materials exist, which can also be mixed to produce new homogeneous materials.

For the material maps, the material slabs are filled into grids that fit the mask of a material surface, which can be a portal, sensitive or passive surface. In every bin, there will be a single material slab of the averaged material and thickness that result from the ACTS material mapping process described in Chapter 4. An example material map taken from the DETRAY toy detector is shown in Figure 6.11a and 6.11b. Both the homogeneous material and the material maps are kept in the detector material store, from where they are recovered with the material link of a given surface descriptor.

In order to provide a consistent interface for both types of material description towards, for example, the material interactor implementation, a dedicated `material_accessor` exists, which either performs a grid lookup or fetches the material slab directly from the material store.

6.2. Acceleration Structures

During track reconstruction, nearby detector surfaces need to be searched for compatible measurements to be added to the track candidate. In order for this to proceed, the track candidate needs to be directed to the sensitive detector elements in its path. The DETRAY navigation will therefore need to access the detector and test compatible surfaces by intersection. Since it can be computationally costly to check every surface that is contained in a given volume, a mechanism has been provided by which the navigation can request surfaces restricted to a spatial neighbourhood around the current track position.

The deployment of *acceleration structures*, which are data structures that perform such a spatial search, is common in computer graphics and many different options are available. For example, in *Bounding Volume Hierarchies* (BVH) [206, 207], shapes are searched for by successively intersecting a ray with a hierarchy of respective minimal bounding boxes. This way, the search space for the navigation can be reduced efficiently by only testing the content (box or shape) of the box that is hit by the ray. Another popular choice for an acceleration structure in ray tracing is the *kd*-tree [208], which partitions a k-dimensional search space along one of its axes at every node and allows to reduce the number of candidate surfaces by performing a tree traversal.

In ACTS, an object or surface grid [209, 210] is used in the silicon tracker geometries, since it allows to exploit a number of geometric constraints for these detectors. For example, the sensors are grouped in layers and can therefore be registered in a two-dimensional grid instead of a three-dimensional one. This reduces complexity, not just in the way data is accessed in the grids, but also in the way compatible grid bins are found for a track during propagation. This way, it is not necessary to find all bins that were traversed by the track in a volume up to its current position, as is done in Ref. [210, 211], but it suffices to project the current track position onto the two-dimensional grid and include neighbouring bins. Furthermore, the sensor surfaces are in many cases of the same shape and positioned regularly in space with very little overlap, which simplifies their association to the grid bins. This also means that regular grids with a constant bin width are already a close fit to the detector geometry. Two examples are shown in Figures 6.12 and 6.13.

With the exception of misalignments, the detector geometry is static throughout track reconstruction, which means that the grids can be built ahead of time and will not need subsequent updates with IoV changes. In this setup, ACTS can even pool an entire spatial neighbourhood of surfaces in every bin, reducing the grid search to a projection of the track position onto the grid axes, followed by a single bin lookup. This could, however, increase memory consumption and, in addition, leads to a jagged memory layout for the grid bins, since bins at the rim will get fewer neighbouring surfaces registered during grid preparation.

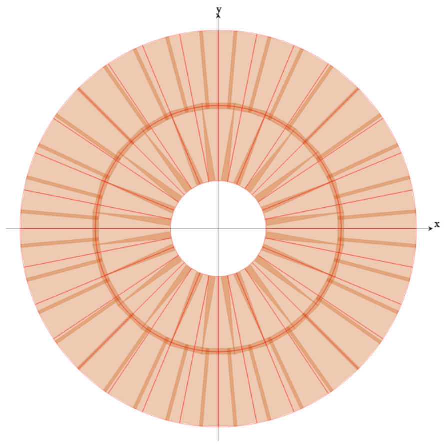


Figure 6.12.: Example of a disc surface grid structure, as found in the ODD endcap volumes.

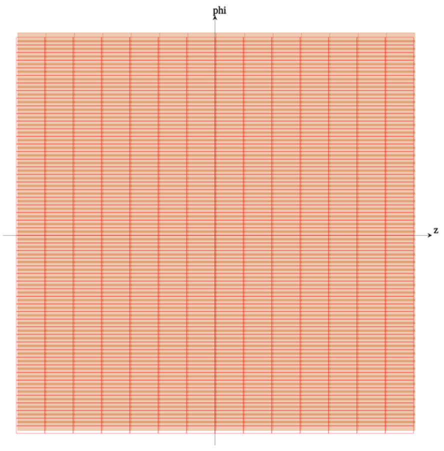


Figure 6.13.: A two-dimensional cylindrical surface grid from the ODD barrel section. The grid is unrolled along the ϕ-axis.

Much like the mask implementation on the surfaces, the volume descriptors in the DETRAY detector each link to a number of acceleration structures they can query for surface candidates during navigation. Every volume contains at least one default acceleration structure, which is currently a brute force method. The brute force method simply returns all surfaces that were registered with it to the navigator, which are usually just the volume portals. However, depending on how the detector was built, it can also contain sensitive or passive surfaces. For example, if no further acceleration structures are provided during detector construction for a volume that contains sensitive surfaces, the DETRAY detector IO will automatically register all surfaces in the default (brute force) method together with the portals and passive surfaces.

In contrast to the mask link, the accelerator link has multiple slots, so that in principle, a volume can be equipped with different ways of finding or producing surface candidates for the navigation. Every individual slot is associated with a surface category, such as portal or sensitive surfaces, which is governed by an `ID enum` in the volume that can be switched at compile time in the detector metadata. The `ID` must at least contain the portal, passive and sensitive surface categories (which may however point to the same slot), but can also be extended with custom categories if required by a specific detector setup. During the candidate search in the navigation, every acceleration structure that is registered in a slot of a given volume descriptor is queried in succession, starting with the respective default method, ensuring that the portals are always accessible to the navigator. To this end, the detector volume class that wraps the volume descriptor provides a `visit_neighbourhood` method, which calls the following code on each collection in the detector accelerator store with a number of extra arguments `args`:

```
// Get the specific acceleration structure from the store collection
decltype(auto) accel = collection[index];

// Run over the surfaces in a single acceleration data structure
for (const auto &sf : accel.search(det, volume, track, cfg, ctx)) {
    functor_t{}(sf, std::forward<Args>(args)...);
}
```

The `functor_t` is a piece of externally provided functionality that is called on each surface candidate that is returned by the acceleration structure instance named `accel`. This will, in particular, be the candidate search implementation of the navigator, as discussed in Section 6.3, where every surface candidate returned by the acceleration structure is tested via straight-line intersection. Different functors could be passed by other navigator implementations or any hypothetical parts of the code that might need access to a surface neighbourhood.

As a consequence of how the navigation candidate search is called in the code example above, every acceleration structure in DETRAY has to provide a `search` method that takes references to the detector instance, the volume descriptor, the current track state and the navigation configuration as arguments and returns an *iterable* range of surface candidates. Currently, this is not explicitly enforced in the code, since abstract interfaces as part of the acceleration structures cannot be used in kernel code, but a type concept could be added with `C++20` [212]. How the candidates are subsequently obtained by the search method is the sole responsibility of the acceleration structure. In principle, it is free to even generate candidate surfaces on the fly, for instance, if the memory restrictions are such that not all detector surfaces can be precomputed and held in memory at all times.

This design of the surface candidate access allows different detector setups to implement custom navigation schemes by defining and sorting distinct surface categories of a single volume into corresponding acceleration structures. It also allows to vary the acceleration structure per volume, so that different volumes in the same detector may apply a custom sensitive surface navigation. This becomes especially relevant once the detector contains subdetector systems with diverse geometries and hence navigation strategies, like silicon trackers vs. calorimeters. But already among the volumes of a silicon tracker, there will be volumes that contain a large or small number of sensitive surfaces and those that are empty gap volumes. It may be the case that only in volumes with a large number of sensitive surfaces, the deployment, and hence overhead of a surface grid lookup makes sense, while for the other two cases testing all contained surfaces for intersection could be the more efficient choice.

Since the acceleration structures are static throughout the lifetime of a detector instance, they are prepared by the host-side IO functionality and subsequently moved to the device. In case of the surface grids, for example, the data structures are already finalised during the tracking geometry construction in ACTS and later only converted to a DETRAY format. In order to move them to device efficiently, the data of the acceleration structures of multiple volumes therefore has to be pooled in collections as part of the detector, again similar to the storage of the surface masks. From these collections a specific acceleration structure is then accessed at runtime by its type id and collection index, which are stored in one of the slots of the accelerator link, as described above.

However, the data state of a single acceleration structure may be a lot more complex than that of a mask, which makes the definition of a custom VECMEM view type mandatory. A grid, for example, holds multiple vector containers internally. Since the accelerator store of the detector cannot a priori know which acceleration structure types will be added to it, the memory management remains the responsibility of the acceleration structure collection. This way, no internal details on the implementation of a particular acceleration structure or collection is coupled into the detector container that holds them. In fact, the same `detray::multi_store` container class is used for acceleration structures, as is used for the surface masks and detector material description.

6.2.1. Brute Force Surface Candidate Search

In order to have a reliable default candidate surface access for the navigation and to register the portals in a way that all of them are checked after every change to the track state, a brute force search method was added to the detector. It keeps a global record of all surface descriptors that were added to it, as well as an offset for each volume into that global surface descriptor container. Since currently every volume registers at least its portal surfaces with the brute force searcher, the correct offset into the surface container can be accessed through the volume index on a vector of offsets. Together with the offset of the following volume, the surface descriptor index range for a given volume can be formed. From this index range, an iterator pair (`detray::ranges::subrange`, see below) is generated on the surface descriptor record and returned to the candidate search to fulfil the interface requirements towards the navigator.

The memory layout of the brute force accelerator for the entire detector is therefore restricted to two vector containers, which keeps the memory management requirements for this class quite simple. When the detector accelerator store is moved to device, the brute force accelerator collection returns a multi-view of its two vector containers and later is constructed around this multi-view again in the device code.

6.2.2. The detray Grid Implementation

The DETRAY grid currently serves three main use cases and might also be used in TRACCC for seed finding in the future:

- **Surface grids:** Currently two-dimensional grids as navigation acceleration structures
- **Material grids:** Two- and three-dimensional grids that hold material map data
- **Volume grid:** Two- or three-dimensional grids that hold detector volume indices
- **Spacepoint grid (TRACCC):** Point search data structure to construct track seeds.

As such, the grids need to be two- and three-dimensional and provide a dynamic number of entries per bin for the surface and space point grids. The material grids will hold a single material slab per bin, but might share the particular material instance between multiple bins in order to save memory, as is proposed in ACTS. While the surface grids need to adjust their shape to the respective volume that contains them, material grids are used to implement material maps, which can be seen as a kind of material texture on a surface, and consequently have to be fit to any surface shape that ACTS can equip with material. For detectors like calorimeters or wire chambers, the material mapping might be done in the future for an entire volume and instead of projecting it onto e.g. the portal surfaces, a three-dimensional material grid could be used (*volume material*).

In order to test the efficiency of a surface grid neighbourhood lookup against assigning an entire neighbourhood to a single bin (in the following called *neighbourhood packing*) on the device in the future, the DETRAY grids should support the access of entries in neighbouring bins according to a range search on its axes. The size of these bin ranges will be given at runtime, so that it will be possible to set them dynamically, for example, according to the projected track covariances.

To be able to use different local navigation strategies in different volumes with the same navigator, the interface towards the navigation for surface candidate queries should be kept decoupled from any specific acceleration structure implementation. As explained above, the interface is therefore limited to a direct iteration over the surface candidates returned by the acceleration structure, which poses an additional requirement on the way the neighbourhood lookup returns its results.

Finally, since an a priori unknown number of volumes with sensitive surfaces and material maps per volume are needed for different detectors, the DETRAY detector needs to support containers that hold collections of different grid types with a dynamic number of entries. Like for the geometry implementation, the same restrictions to polymorphism, the use of pointer based data structures and memory layout/allocations apply. In particular, the number of containers that need to be moved to the device should be kept low in order to simplify host-device memory transfers.

From the beginning of the project, there has been a grid class in DETRAY, called `grid2`, which initially implemented a two-dimensional host-only grid. It was then later ported to device code by replacing the containers it used internally by either VECMEM host or device containers and adding a fairly involved VECMEM buffer and view implementation around it. The `grid2` type is currently used in TRACCC for seed finding, but has several shortcomings when it comes to the DETRAY use cases as outlined above.

For one, it does not support three-dimensional grids, which might hinder future development. More importantly, however, its neighbourhood lookup implementation depends on the allocation and filling of a result vector of candidate surface indices and was hence never ported to device code. Furthermore, since it relies on a jagged vector container internally for its dynamic bin capacity implementation, there is no easy way of forming collections of `grid2` instances that can be put into a DETRAY multi-store without forming a vector-of-vector data structure. An initial implementation used a static array of grids, where the number of grids in the collection had to be known at compile-time.

In the following, the current main DETRAY grid implementation will be discussed, which was developed in the scope of this thesis. It took some design aspects of the original `grid2` class, in particular the use of serializers and populators, and addressed the issues laid out above. Mainly, the new implementation allows to perform an in principle N-dimensional neighbourhood lookup in host and device code without dynamic memory allocations and can be transferred in large collections to device by just a few container copies. Furthermore, it does not only facilitate static and dynamic bin capacities, but also several bins sharing the same memory, as may be required by ACTS material maps in the future.

Finding a Grid Bin - Axes and Serializers

At the heart of the grid implementation is the question of how to retrieve the content of a bin given a global (track) position. This problem is usually solved by projecting the global position onto the local coordinate system that is spanned by the grid axes and then checking the value of every local coordinate against the bin edges of the corresponding axis. This will yield a bin index per axis, which together need to be translated into a position in memory to fetch the bin content. The latter task is solved by the *serializer*, which translates a local bin index, consisting of an index per axis, into a global index that can be used to access a bin container for the bin entries.

Following the grid design in ACTS, the DETRAY grid axes come with two different binning types, *regular* and *irregular* binning, as well as several axis boundary behaviours. The binning type of an axis determines the bin widths on the axes and how to calculate a local bin index from the corresponding coordinate of a position in space. The axis boundary type determines how to access the under- and overflow bins at the edges of the axis value range. Similar to ACTS, DETRAY provides three axis boundary behaviours:

- **Open:** specialised under- and overflow bins exist for queries outside of the axis span,
- **Closed:** no under- and overflow bins exist, instead all values that are outside of the axis span are mapped into the first and last bin, respectively,
- **Circular:** models periodic boundary conditions for ϕ-axes, where the under- and overflow bin indices wrap around back into the valid axis bin range.

Following the local bin index calculation that is done by the axis binning, the axis boundary implementation maps the local bin indices into the valid range for the respective axis. This ensures that a coordinate lookup on any type of axis always returns a valid bin index. The binning types provide the possibility to describe any of the above boundary behaviours together with either regular or irregular bin edges along the axis.

While regular binning is straightforward to achieve with just the axis span and the number of bins, irregular binning needs to define every bin edge individually. Therefore, for every lookup that uses irregular binning, the bin index has to be searched on the axis. This is why irregular binning, although allowing to fit grids very precisely to the data they contain, is in general discouraged due to inferior performance expectations.

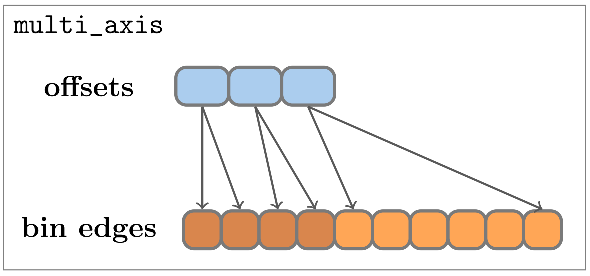


Figure 6.14.: The DETRAY `multi_axis` keeps a number of individual axes, each of which marks out a range of bin edges in a global bind edges container. For regular binning (dark) only two entries are required, while for irregular binning (light) every individual bin edges needs to be accessible.

Both axis binning and boundaries are defined as separate types that can be added to the DETRAY `axis` class in order to equip it with different combinations of behaviour. The concrete bin data, in particular, the number of bins and the bin edges, however, are not owned by the binning or even the individual axes, but by the class that contains all grid axes, the `multi_axis` shown in Figure 6.14. A single axis, and in particular the binning it uses, can access this data to perform their functionality, but none of them are concerned with the memory management, so that the data for all axes can be handled in a centralised way.

This also allows to use the same bin data containers for all axes of a grid, thus reducing the overall number of containers that need to be moved to device. The difficulty here is to allow any combination of regular and irregular binning per axis. An axis with irregular binning has to have access to the edges for a number of bins that is unknown at compile time, while the same is not the case for regular binning which always needs to access three values in memory, regardless of the number of bins on the axis.

Two containers have been defined as part of the `multi_axis`, one that holds all of the bin edges and one that holds the offsets into the bin edges container together with the number of bins for every individual axis. In the case of irregular binning, the bin edges range is obtained in a straightforward way from its offset into the bin edges container and then including the following bin edges according to the number of bins defined for it. For regular binning, the offset and number of bins are defined the same way, but the entries in the bin edges container always number exactly two, containing only the values of the total axis span.

The DETRAY `multi_axis` uses a variable number of fully defined single `axis` types, including all binning and boundary definitions. It can then retrieve any individual axis by instantiating it with the correct offset into the common bind edges store, if a concrete axis is requested by a client. This can be done by axis number (0, 1, 2, . . .), label (x, y, z, ϕ, . . .) or the concrete single axis type if it is known. Since the label is part of the axis type definition, every single axis type in a multi-axis is unique and the label can be resolved at compile time. In the following, several methods of accessing an individual axis are shown:

```
// Explicit definition of multi axis type
using cylindrical3D_axes_t =
    multi_axis<true, detray::cylindrical3D<algebra_t>,
               axis<closed<label::e_r>, regular<>>,
               axis<circular<label::e_phi>, regular<>>,
               axis<closed<label::e_z>, regular<>>>;

    // Lower bin edges for all axes
    vecmem::vector<scalar> bin_edges = {-10.f, 10.f, -20.f, 20.f, 0.f, 100.f};
    // Offsets into edges container and #bins for all axes
    vecmem::vector<dsized_index_range> edge_ranges = {
        {0u, 20u}, {2u, 40u}, {4u, 50u}};
    // Data-owning multi-axis
    cylindrical3D_axes_t axes(std::move(edge_ranges), std::move(bin_edges));

    // Total number of bins for the grid
    detray::dindex n = axes.nbins();

    // Access to the single axes
    auto r_axis   = axes.get_axis<detray::axis::label::e_r>();
    auto phi_axis = axes.get_axis<1>();
    auto z_axis   = axes.get_axis<axis<closed<label::e_z>, regular<>>>();
```

Apart from the memory handling, the `multi_axis` also defines the local coordinate system that corresponds to its collection of axes, so that the bin index lookup can be performed correctly for every coordinate. The grid instance can then use the local coordinate projection provided by its axes to transform a given global position into the correct basis before it hands it through to perform the bin lookup.

After the axes have returned the correct local bin index for a query point, consisting of a bin index per axis, it has to be translated into a memory location in order to fetch the actual bin content. This is done by the serializer of the grid, which thus defines the memory layout of the bin storage. For a one-dimensional grid this is a trivial task, as the local bin index of the only axis such a grid contains can directly be taken as a container index. With higher dimensional grids, the multi-dimensional data have to be serialized into the one-dimensional memory address space.

The simplest implementation of this, and the only one that is currently available in DETRAY, is to use same layout in memory that is commonly used for matrices. In DETRAY, this is done by the one-, two- or three-dimensional `simple_serializer` class, that orders the bin content such that the indices of the first axis are held together in memory. This, however, may lead to poor memory locality for bin neighbourhood lookups, as the bin content for neighbouring bins is spread out in multiple, potentially far distant blocks in memory. Serializers that transform the local bin index according to space-filling curves, such as in Refs. [213, 214], could help with this issue in the future.

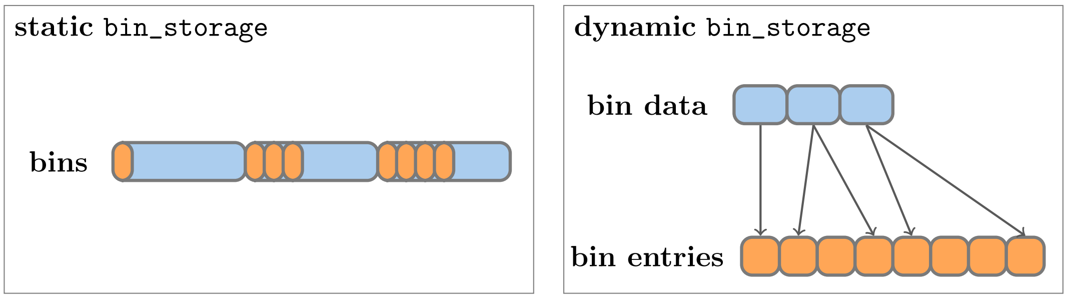


Figure 6.15.: Static (left) and dynamic (right) bin storage models. While the static bin storage can keep the bin entries as part of the bins directly, the dynamic storage requires the generation of subranges into a global bin entries container.

Accessing a Grid Bin - The Bin Storage

With the axes contained and managed by the `multi_axis` class, the next important component is the storage of the grid bin content. For some grids, like material grids or the volume search grid, the bin content will always consist of a single entry. For these cases, the `single` bin type was added, which wraps a single value. For the surface grids used in the navigation as acceleration data structures, the picture is a bit more complex. Depending on the extent of the detector surfaces, a single surface might get sorted into multiple bins and, if the grid binning is chosen to be sufficiently coarse, multiple surfaces may end up in the same bin. Either way, the surface grids have to be capable of storing multiple entries per bin. If no bin neighbourhood packing is applied for the grid, and the detector substructure that it holds is sufficiently regular, a static bin capacity can be set at compile time, since the number of bin entries can be made to be evenly distributed across the bins. In this case, the grid bins can be modelled as arrays with a fixed capacity, as done in the `static_array` bin class, and a bin storage container can hold them directly as a single memory block.

When bin neighbourhood packing is applied, and also for spacepoint grids in TRACCC, the bins can potentially hold a varying number of entries and setting a single compile-time storage capacity for all of them would result in the largest bin dictating the memory consumption globally. To avoid the resulting memory waste by this memory over-subscription, DETRAY provides an additional bin type with dynamic storage capacity per bin, the `dynamic_array` bin type. In order to prevent another instance of vector-of-vector memory layout, and to facilitate memory sharing between different bins, a bin storage implementation was added that keeps the bin instances and their data in separate containers, similar to how the axis containers are split between offsets and bin edges, as shown in Figure 6.15.

The indexing data for the `dynamic_array` bin type combines the offset into the bin entries container, the number of entries or size and the bin capacity. When a bin of a given global bin index is requested from the dynamic bin storage, it will create the corresponding instance of a `dynamic_array` bin on top of the bin entries container and return it to the caller. This way an a priori unknown number of bins with a dynamic storage capacity can be held in only two distinct containers that can be easily extended and shifted to fit into larger grid collections by adjusting the offsets. Furthermore, two bins with identical content can be given the same offset into the entries container and will hence reference the same bin entries in memory, thus reducing the memory consumption even further in case the bin entries require a lot of memory each.

Since the interface towards the DETRAY navigation requires the surface candidates to be iterable, both the bin storage service as well as the individual bins it contains need to be *ranges*, in other words, they have to provide an iterator pair that allows to traverse the bins and the bin content from beginning to end, respectively. The `single` bin achieves this by extending the `detray::views::single` type, as described down below, which emulates an iteration for a single value. The `static_array` bin type is a range by design, since it contains an array internally. However, the iteration has to be restricted to the valid entries that have been filled into the bin. Therefore, the `static_array` bin returns a `detray::ranges::subrange` as its range access, which is an iteration that is restricted to a subrange on the container. The `dynamic_array` bin implementation also returns a subrange that is restricted to the valid bin entries, however, the subrange is defined on the global bin entries container instead of the bin itself. As a result, any bin in any of the DETRAY grid bin types can be range-iterated to access its content:

```
// Iterate through the bins of a grid (bin storage range)
for (const auto& bin : grid.bins()) {
    // Iterate through the content of each bin
    for (const auto& entry : bin) {
        std::cout << entry << std::endl;
    }
}
```

Strapping it all together - The detray `grid`

Combining the bin storage with the axes and providing an interface to fill and retrieve the bin content is the task of the grid class. It takes template parameters on the multi-axis, bin and serializer types to be used, which determines the complete behaviour of the respective grid, as shown in Figure 6.16. Any of the types described above can be freely combined to produce a large number of different grid types. The only restriction is that the coordinate system given to the `multi_axis` class has to correspond to the axis types that have been passed along with it. A helper type, called `axes`, exists which infers the correct multi-axis type from a given DETRAY geometry shape plus some extra information on the axis boundaries and binning. This allows to easily fit a grid type onto a detector surface, for instance during the construction of material maps.

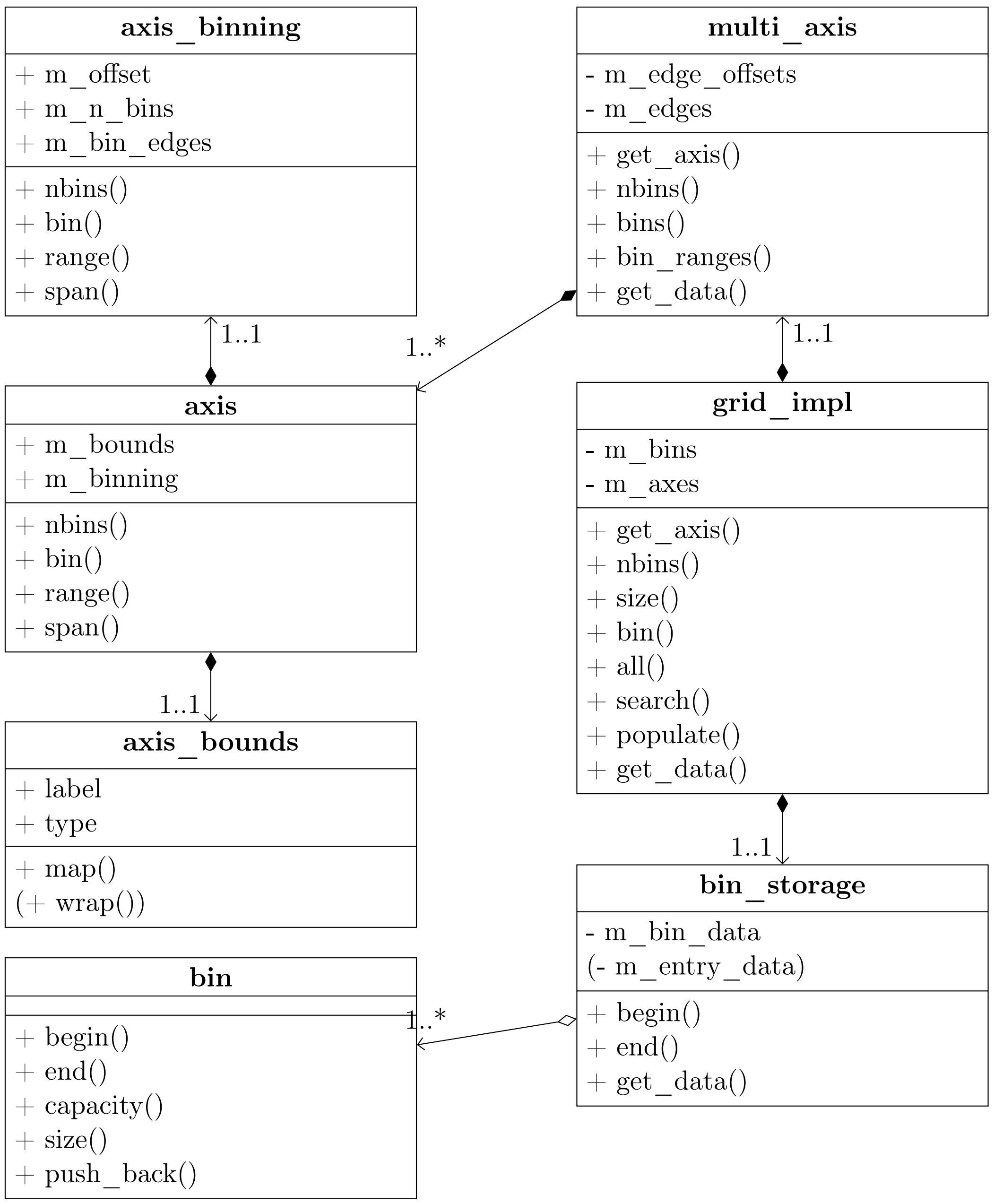


Figure 6.16.: Class diagram showing parts of the DETRAY grid implementation. The `axis_binning` stands for the regular and irregular types, the `axis_bounds` stands for the open, closed and circular types and the `bin` element stands for the `single`, `static_array` and `dynamic_array` types. The elements in parenthesis are only present in some of the concrete types.

The different grid shapes that can be generated and mapped to portals, passive or sensitive surfaces this way are shown in Figures 6.17, 6.18 and 6.19:

```
// A rectangular grid with regular, closed axes and single int entries
using rec_grid_t = grid<axes<rectangle2D>, bins::single<int>>;

// A disc shaped grid with irregular binning in r, regular binning in phi,
// capable of holding 4 doubles in every bin
using disc_grid_t =  grid<axes<ring2D, axis::bounds::e_closed,
                               axis::irregular, axis::regular>,
                          bins::static_array<double, 4>>;

// A 3D cylindrical grid with regular binning and open r- and z-axis,
// capable of holding an arbitrary number of points in every bin
// note: The phi axis is always circular, as inferred from the cylinder3D shape
using cyl3D_grid_t = grid<axes<cylinder3D, axis::bounds::e_open>,
                          bins::dynamic_array<point3_t>>;
```

In order to fill the grid bins after the grid has been created, a number of *populators* are provided, similar to the design of the `grid2` class. In this case, however, a populator can be applied to any type of grid bin and simply defines a prescription on how to add entries to the bin. Populators exist that replace, attach or fill a bin up with a given value. Currently though, a populator cannot increase the capacity of a `dynamic_array` bin type, the maximal capacity is instead given to the constructor during grid building. In addition to accessing the axes individually, inferring the total number of bins as well as the total number of bin entries, the grid also provides several ways to access the grid bins. Single access is possible by the local or global bin index, as well as using a point in the local coordinate system of the grid. Such a point with a number of coordinates corresponding to the grid dimension can, for example, be obtained by calling the `project` method with an additional transformation. Usually, this will be the placement transformation of the volume or surface the grid is associated with.

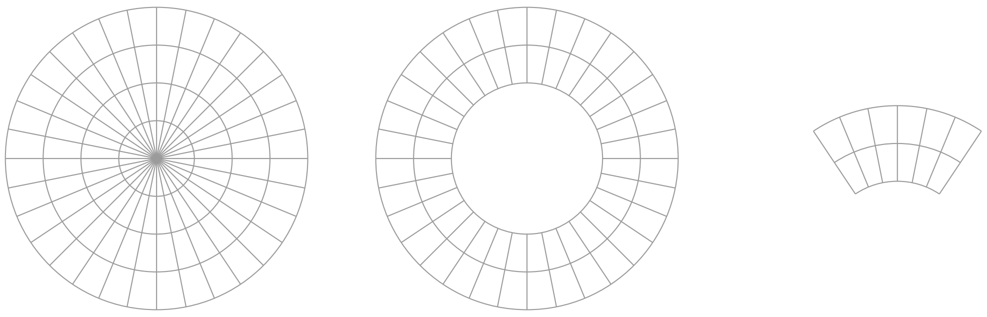


Figure 6.17.: Example polar grid shapes, corresponding to disc-like surfaces.

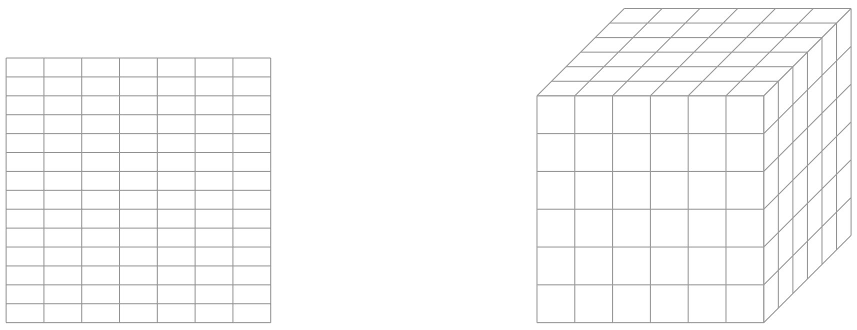


Figure 6.18.: Example grid shapes, mapping to rectangle surfaces and cuboid volumes.

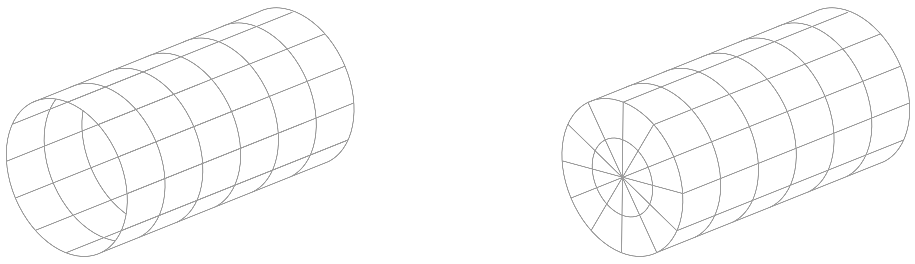


Figure 6.19.: Example grid shapes, corresponding to cylinder surfaces and volumes.

The bin access can also be flattened by calling the `all` or `search` methods, which facilitate the in-situ iteration over the content of multiple bins:

```
// Iterate through every entry that has been added to the grid
for (const auto& entry : grid.all()) {
    std::cout << entry << std::endl;
}

// Iterate through a neighbourhood of bins around the local point 'p'
for (const auto& entry : grid.search(p, search_window)) {
    std::cout << entry << std::endl;
}
```

The memory management of an individual grid consists of moving the data of the multi-axis and the bin storage to device. The grid VECMEM view type hence simply aggregates the view types of the contained `multi_axis` and `bin_storage` instances and hands them back to the respective constructors during the device side construction. These types in turn aggregate the view types of the edge offset and bin edges containers, as well as the bin data and bin entries containers, respectively. The same structure is applied for the grid buffer type. In other words, if the grid is not part of a grid collection, it then manages the data itself:

```
template <typename axes_t, typename bin_t,
          template <std::size_t> class serializer_t = simple_serializer>
class grid_impl {

    [...]

    /// Vecmem based grid view type
    using view_type = dmulti_view<typename bin_storage::view_type,
                                  typename axes_type::view_type>;

    [...]

    /// @returns view of a grid
    template <bool owning = is_owning>
    requires owning DETRAY_HOST view_type get_data() {
        return view_type{detray::get_data(m_bins), detray::get_data(m_axes)};
    }
};
```

6.2.3. Neighbourhood Iteration - Providing Ranges to detray

In order to provide the flattened bin entry view that is required by the navigation, the ranges concept [215, 216] has been introduced to DETRAY. Since the `std::ranges` implementation of the `C++` Standard Library only becomes available with the `C++20` standard [212] and DETRAY was initially implemented using only `C++17` due to CUDA restrictions, a very simplistic implementation of the ranges concept was added natively to DETRAY. It is in large parts based on the GNU implementation of the C++ Standard Library [217], but adding CUDA function qualifiers (e.g. `__global__` for kernel functions). Reimplementing some parts of ranges directly in DETRAY gives full control to guard against possible changes in other ranges implementations, for example, to ensure also complete SYCL compatibility.

A range is any type that provides a `begin()` and an `end()` method, which return an iterator at the start and end position of the range, respectively. The interesting aspect about ranges is the possibility to compose multiple ranges into various new iteration functionality and have them evaluate the result lazily, meaning only in the moment when it is requested by dereferencing the iterator. For DETRAY, this flexibility to compose iterators has allowed to develop a neighbourhood lookup that operates in-place and hence without dynamic memory allocations, which is especially important on the device. In the following, those ranges that are part of DETRAY will be presented, without going into the implementation details, as they just reimplement and follow known concepts.

In many places it has even been possible to adopt Standard Library functionality directly into `detray::ranges`, since for example functions like `std::begin` and `std::end`, which retrieve the begin and end iterator instances, are marked with the `C++` keyword `constexpr`. Together with the `-expt-relaxed-constexpr` flag in CUDA, this allows to reference functions that are completely defined at compile time in CUDA device functions, even though they have not been decorated with CUDA function qualifiers.

Furthermore, `detray::ranges` provides type traits and concepts for iterator and range categories based on the `std::ranges::range` concept, so that potential restrictions on the capabilities of a given iterator or range can be handled correctly by clients. For example, an *forward* iterator can only move forward in strict sequence, while a *random_ access* iterator must provide functionality to move forward and backward, as well as access any element randomly. This changes the way algorithms can be efficiently implemented on top of these iterators or ranges. The base class for all views in DETRAY is its implementation of the `view_interface`, which adds some basic functionality to all ranges. This includes inferring the size of the range from its start and end position, or adding access methods to the front and back of the range.

In the end, the goal is to replace `detray::ranges` with `std::ranges` or at least update it to adapt more of the Standard Library implementations directly.

Single

This very simple range wraps a single element and uses a pointer to that instance as iterator. The iterator points directly to the contained element and the iterator end position, or sometimes called sentinel, points at the memory location directly behind it. Thus, the size of this range is always one. The `single` range is useful when a single value needs to be used with an interface that expects a range, like the single bin type in the DETRAY grids.

Subrange

The most widely used range in DETRAY, which allows to view an arbitrary subrange of another range. It is constructed by passing a range and a pair of indices that define the subrange or directly a pair of iterators that operate on the range that should be viewed. The iterator type used by this range is the same as the range it provides a view on.

Iota

This is a factory type that generates a range which allows iterating through a sequence of values. Most commonly, this is an index sequence given by a start index and an end index, where the end index is exclusive in DETRAY. The iterator of the `detray::iota_view` keeps a value internally, that it increments in lockstep with the iterator until the end of the sequence is reached.

Enumerate

This range performs an enumeration of another range, that is, it returns every element of the range it views with a corresponding index that it increases as the iteration continues. An implementation of this has already been present in the initial version of DETRAY, since it is a very useful utility when a range element is required in conjunction with its position. It was subsequently refactored as part of `detray::ranges`.

Pick

The `detray::pick_view` will return elements of an underlying range if they match a list of indices. The list of indices must be given in a range of its own, as the iterator type of this view traverses the index range and then fetches the corresponding element from the range it views. The pick range is mostly useful in cases where an intermittent or unordered sequence of elements needs to be accessed, since DETRAY does not provide a more general `filter_view`, yet.

Cartesian Product

This range is intended to be introduced in the `C++23` standard [218] and it allows to combine multiple input ranges as a Cartesian product. Every element the view returns is a tuple of elements of the underlying ranges. This range can be useful to index multidimensional data structures.

Join

This view joins different ranges of the same type into a single iteration by switching its iterator form one range to the next once it reached the end of the previous range. There are two implementations present in DETRAY, one called `detray::static_join_view`, which takes a compile-time dependent number of ranges to join and hold them in a `tuple` internally, and one called `detray::join_view`, which takes the ranges to join from a range in turn. The `detray::join_view` thus takes a runtime defined number of ranges to join, but the implementation is slightly more involved as it has to handle the inner and outer range traversal correctly.

The implementation of the grid neighbourhood access, called the `bin_view`, makes use of several ranges composed together. It generates a local *bin indexer* from a number of `detray::iota_view` ranges that are initialised with the respective local bin index range per axis, and a `detray::cartesian_product_view`. The number of `iota` ranges has to match the number of axes in the grid, which is achieved by `C++17` fold expressions:

```
/// @returns the local bin indexer for the given @param search_window.
/// (cartesian product of the bin index ranges on the respective axes)
template <std::size_t... I>
DETRAY_HOST_DEVICE inline auto get_bin_indexer(
    const axis::multi_bin_range<sizeof...(I)> &search_window,
    std::index_sequence<I...>) {

    return detray::views::cartesian_product{
        detray::views::iota{detray::detail::get<I>(search_window)}...};
}
```

This range composition will generate the local bin indices of exactly those bins that form the neighbourhood around the lookup bin "on the fly", while iterating the neighbourhood. Thus the grid does not need to compute a collection of local bin indices beforehand, whose number is unknown at compile time and for which consequently a dynamic memory allocation might be needed.

The search window that is used to initialise the `detray::iota_view` ranges, is generated by the grid axes. Every axis will add a number of neighbouring bins around the respective coordinate of the query point, according to a runtime configuration. For grids that apply neighbourhood packing this parameter can be set to zero, so as to return just a single bin which already contains the entire neighbourhood.

The `bin_view` class is a range itself, as it implements an iterator type that allows to traverse the bins of the neighbourhood directly. This nested iterator holds a bin indexer that generates the correct local bin indices and then calls the grid to fetch the corresponding bin whenever the iterator is dereferenced. This way, iterating through the neighbourhood of a given bin with a dynamic search window size becomes as simple as:

```
// Index ranges of neighbouring bin indices per axis
axis::multi_bin_range<3> search_window{
        axis::bin_range{0, 10}, axis::bin_range{2, 4}, axis::bin_range{1, 5}};

// Neighbourhood iteration
for (const auto& bin : axis::detail::bin_view(grid_3D, search_window)) {
    for (const auto& entry : bin) {
        std::cout << entry << std::endl;
    }
}
```

The correct generation of local bin indices on a ϕ-axis needs some careful consideration though, since the bin ranges that the axes return get mapped according to the axis boundary behaviour. For *open* or *closed* axes this does not pose a problem, since the index ranges simply get clipped or shifted correctly, but on a *circular* axis an under- or overflow value will get wrapped around. This can lead to index ranges where the upper boundary contains a smaller value than the lower boundary, which cannot be handled correctly by the `detray::iota_view`. Therefore, the bin index range on a ϕ-axis is returned without performing a mapping and the local bin indices are mapped to the correct values after the generation by the bin indexer instead. This has the disadvantage though, that the mapping has to be done for every local bin index individually, while for the other axis types it needs to be done only once.

The last step that is needed now to implement the grid-based candidate search is to flatten out the iteration over the bin content, so that the navigator can traverse the neighbouring surfaces without knowing any specifics of the grid structure. This is achieved by composing the `bin_view` generated for a query point with a `detray::join_view`. That is possible, since every grid bin is itself a range, which consequently makes the `bin_view` a range of ranges and thus a correct input to a join view:

```cpp
/// @brief Return a neighbourhood of values from the grid
///
/// The lookup is done with a search window around the bin
///
/// @param p is point in the local frame
/// @param win_size size of the binned/scalar search window
///
/// @return the sequence of values
template <typename neighbor_t>
DETRAY_HOST_DEVICE auto search(
    const point_type &p, const darray<neighbor_t, 2> &win_size) const {

    // Return iterable over bins in the search window
    auto search_window = axes().bin_ranges(p, win_size);
    auto search_area = axis::detail::bin_view(*this, search_window);

    // Join the respective bins to a single iteration
    return detray::views::join(std::move(search_area));
}
```

The implementation of the DETRAY surface grid candidate search using ranges has allowed to avoid dynamic memory allocations both on the host and device. It makes use of generic tools that can be reused in other parts of the code, which also makes the client code easier to understand and maintain. On the flip side, the ranges implementation itself is not trivial and needs to be maintained within the DETRAY project. In some places the requirements for classes to become ranges has led to verbose and repetitive code, especially in the implementation of nested iterator types.

The computational efficiency of the ranges-based neighbourhood iteration on the device also has to be investigated, as it results in code with many branches and thus could lead to a de-synchronisation of the device threads.

6.2.4. Grid Collections in the Detector

The final challenge to make the grids available on device is to hold a runtime defined number of grids together in global collections as part of the detector acceleration structure store and ensuring efficient memory management. To this end, the grid, multi-axis and bin storage classes can be defined as either *data-owning*, which is the case for an individual grid that manages its data on its own, such as the volume or a spacepoint grid, and *non-owning*, where the grids are managed as part of the grid collection instead. Whether a grid is data-owning or not is already determined by its multi-axis type and then adopted by the grid and bin storage, ensuring that all three types agree on the mode of memory management.

All of the functionality described above works the same way with non-owning grids, with the exception of the VECMEM view generation and the corresponding grid constructors, which only exist for data-owning grids. Instead, the grid collections are responsible for providing a VECMEM view for the combined data of all grids. It consists of the view of a bin storage, the view of the container for the single axis offsets, the view of the axis bin edges, as well as the view of the container that holds the offsets of the individual grids into the collection. Except for the extra offset container, the other containers are of the same type as their counterparts in a data-owning grid, but this time they contain the data for all grids that are part of the collection. If the grid collection consists of grids with `dynamic_array` bins, the total number of containers that needs to be moved to device is five, otherwise four, since the other bin types require one container less for the bin storage. Depending on the particular detector, the number of containers that will be be moved to device for the different grid types it defines is of the order of a few tens. The ATLAS ITk, for example, needs four distinct grid types, namely disc and cylinder grids for both acceleration structures as well as material maps, which results in $2 \times 5 + 2 \times 4 = 18$ containers to hold an arbitrary number of actual surface and material grid instances.

In order to retrieve the data for an individual grid from the collection, the offset for this grid's bin range is fetched from the offset container of the collection. This is done using the index of the grid in the collection, which is kept either in the volume accelerator link or the surface or volume material link. The offset provides the information of where in the global bin storage the particular grid instance will find its data. Instead of the bin storage owning the bin container, the non-owning bin storage simply keeps a `detray::subrange` into the global bin storage as a member, which is initialised with the bin range offsets on the global bin container. This is decided at compile time by using the `std::conditional` trait internally:

```
using bin_range_t =
    std::conditional_t<is_owning, vector_t<bin_t>,
                        detray::ranges::subrange<vector_t<bin_t>>>;
[...]
```

```
private:
/// Container that holds all bin data when owning or a view into an
/// externally owned container
bin_range_t m_bin_data{};
```

As a result, a data-owning bin storage, as determined by the boolean flag `is_owning`, will directly hold a vector of bins, while a non-owning bin storage will hold a view on the grid collections global container. Since both the vector and the subrange are ranges, all downstream interfaces work the same. A similar thing is done for the `multi_axis`, where the non-owning type holds a subrange into the global container of the bin edges offsets per axis and a pointer to the global bin edges container. With the bin edges offsets and the number of bins per axis, the individual axes can retrieve their respective data in the same way as for a data-owning `multi_axis`. Since the axis and the binning implementations do not own data by design, the pointer to the bin edges storages is simply passed on, and the ownership of that data remains opaque to them.

Storing any kind of view as a class member, be it a pointer, an iterator or a range, is only possible if the data is never subsequently relocated in memory, which would invalidate the pointer or iterator. The reason the non-owning grid types work in the context of host-device memory transfers is, that they are instantiated on the fly by the grid collection when a specific grid is requested by a client. It is the grid collection, holding the persistent state of the grids, that is copied between host and device instead. However, the instantiation of a non-owning grid on the fly by the grid collection comes at the cost of fetching a number of offsets and creating the correct iterators, which becomes part of the overhead of querying this particular acceleration structure. Any potential impact on the performance of the navigation and material handling needs to be investigated in the future. Both, the queries to the accelerator grids, as well as the queries to the material grids, will not happen at every step of the propagation, but for the most part only when portals are encountered, which carry the material and induce a volume change in the navigation.

This design, however, also complicates the construction of a grid collection, since the offsets the grids are using internally, for instance to find the correct bin edges per axis, need to be shifted when the bin edges are appended to the global containers of the grid collection. A dedicated `push_back` method has been added to the grid collection that appends the data of a data-owning grid to the grid collection and sets all shifts and offsets correctly. Furthermore, the addition of new grid data containers to the collection has to take into account that the grid bin storage consists of two internal containers for the `dynamic_array` bin type, but only holds only one for the other bin types.

6.2.5. Summary

Being multi-dimensional data structures that are used in many different contexts, the addition of grids that are constructed on the host and then copied to the device in large collections has been a challenging task. Nevertheless, DETRAY has been equipped with a grid utility that can handle any use case the ACTS grid implementation currently provides as well.

DETRAY is capable of modelling N-dimensional regular and irregular grids, if provided with a corresponding local coordinate definition and serializer, as the main implementation is fully generic with regards to the number of axes a grid can contain. The grids can have bins that hold only a single entry or a collection of entries, with and without a compile time bin capacity limit, depending on the use case of the particular grid type. The grid bin type with dynamic capacity was designed to also facilitate the sharing of memory if their bin content is identical, thus reducing memory consumption. A demonstration of this is yet to come, however. The neighbourhood lookup has been added such that it can adapt the search area at runtime and can be deployed more efficiently in device code, since it does not rely on dynamic memory allocations. First timing measurements are showing hints that the in-situ iteration is also faster on the host than the corresponding zone finding implementation of the `grid2` class, which uses dynamic allocations to return its results to the caller. In order to provide this functionality, DETRAY was equipped with a minimalistic ranges implementation that can be reused in other parts of the code and is based on and adopts parts of the GNU C++ Standard Library implementation.

A major issue, next to the complexity of the implementation itself, is its *const-correctness*, that is, to keep data that should not be changed immutable. In the case of the non-owning grid types, the `const`-correctness is currently broken, since instantiating a constant grid instance propagates "constness" to the pointer or range members in a very inconvenient way. It will render the pointer or range member itself constant, but not necessarily the data it is pointing to. In some cases, this was solved by defining all pointer members to be always pointing to constant data by default and allowing the change of the underlying data in the non-constant methods of the grid, like `populate`, by using a `const_cast` to remove "constness" in this instance. On a whole, the issue should obtain a more stringent fix in the future.

6.3. Navigation through the detray Detector Geometry

6.3.1. The Propagation Flow

The track parameter and covariance transport for DETRAY was developed outside the scope of this thesis, as was the material interaction implementation. However, the navigation which was reimplemented from its initial version to work with the detector geometry described in Section 6.1, is an integral part of the overall propagation workflow, which is inspired by the ATLAS tracking chain [219], so a brief overview of the propagation will be given here for context. A description of the theoretical background of the track parameter and covariance transport can be found in Section 3.2.

The propagation module is responsible for the transport of a set of initial track parameters and their covariances through a detector geometry to their final values at the end of the track, including all intermediate track states at every detector surface that was encountered by the track in between. The main components needed to achieve this, are the *stepper*, the *navigator* and a number of *actors*, which are all steered by the *propagator*. All of these classes are implemented in an inherently thread-safe way that has been adopted from ACTS, where the data the respective algorithm operates on is separate from its implementation and is held as part of a *state* object that exists once for every thread. The state objects are given to the algorithmic code along with any other arguments, such as their configuration.

The integration of the equation of motion, as well as the computation of the transport Jacobian, which was described in Section 3.2, is done in the stepper. Two stepper implementations exist, one that models a straight line (in case there is no magnetic field) and one that applies an implementation of the Runge-Kutta-Nyström (RKN) algorithm [145, 146]. The transport Jacobian is accumulated as part of the stepper state and is determined according to the semi-analytical Bugge-Myrheim-Method [147, 148]. The Runge-Kutta stepper can thus transport a track parametrization through an inhomogeneous magnetic field, which is sampled a number of times during the computation of the stages of the RKN algorithm by interpolating a magnetic field map that contains the field strength vectors at discrete lattice positions. The magnetic field description in DETRAY is provided by the COVFIE library, which in general models vector fields on device [181].

The DETRAY track parameter definition is identical to the one in ACTS, in particular, it includes a time parameter which will allow to transport timing information on the measurements, but is currently still a placeholder. There are *free track parameters* **F**, defined in the global coordinate frame, as well as *bound track parameters* **B**, defined in the local (bound) coordinate system of a given detector surface, as detailed in Section 3.2:

$$\begin{aligned}\mathbf{F} &= \left[x,\, y,\, z,\, t,\, T_x,\, T_y,\, T_z,\, \lambda\right]^{\mathrm{T}}, \\ \mathbf{B} &= \left[l_0,\, l_1,\, \phi,\, \theta,\, \lambda,\, t\right]^{\mathrm{T}}. \end{aligned} \tag{6.2}$$

The track propagation can be started with a set of free or bound track parameters and will from there commence with transporting the parameters and their covariances through the magnetic field to the closest detector surface reachable by the track.

The track covariance transport, as well as the projection to bound track parameters at each encountered detector surface that is either a sensitive or material surface, is done as part of the chain of actors that is called at every step during the propagation. In particular, the actor chain will call every actor type that has been registered at compile-time in sequence and provide it with its respective state. Actors can also be composed, meaning a list of observing actors can be registered with a subject actor and will be handed their subject's state in addition to their own. This results in a compile-time resolved call tree of actors, where, however, direct message passing outside of the actor states is not possible. All actors additionally have access to the propagator's state, in order to be able to receive updates on the latest changes of the track and navigation states and to update the current track state in turn.

Compositional actors allow to model dependencies between actors, where one actor relies on the results of another, but can otherwise not verify if that actors is actually running as part of the actor chain and running in the correct order. Actors that, for example, depend on the initialisation of the bound track parameters at every surface in the stepper state can fail or produce incorrect results, if the parameter transporter is did not run before them. Currently, no observing actors are implemented in DETRAY or TRACCC, however, this could help improve safety and readability in the future.

A number of actors have been implemented besides those for the covariance transport, some of which are part of the TRACCC library and handle, for example, the Kalman filter state updates, as conceptually explained in Section 3.3. Notable actors in the DETRAY project include the *material interactor*, as well as the *aborters*, which monitor different termination criteria for a track. The material interactor handles the material induced energy loss according to Bethe (equation 1.5) or optionally Landau (equation 1.6), as well as scattering related updates to the track state covariance, which for most particle types is calculated using equation 1.7). Aborters exist, for example, to terminate the propagation flow for either runaway or looping tracks with a path length that has exceeded a predetermined threshold or to stop tracks that have reached a particular target surface.

After any update to the track state, either by the stepper or by any of the actors, the track state position and direction has to be put into relation to the tracking geometry through which it is being propagated. This is done by the navigator, which, specifically, will determine the distance to the next most likely detector surface to be encountered by the track. This next surface candidate, as well as the straight-line distance to it, can then be used by the stepper as an initial new step size after reaching a surface. From there, the adaptive RKN algorithm will scale the step size automatically again, but never exceeding the distance to the next surface provided by the navigator.

Furthermore, using the distance to the next surface, the navigator determines whether a track state has been transported to a position in space that can be considered as lying on that surface. This information will be available to the stepper and all actors from the navigation state, and will trigger certain actors to run and perform their respective tasks, such as covariance transport to the local coordinate system of the surface. The navigator also provides additional information on the type of surface that was reached by the track, as well as whether or not it holds any kind of material. The latter is, for example, crucial to trigger the projection of the free track parameters onto portals which carry material and to signal the material interactor to run. As there is currently no direct communication between the different participants in the propagation flow, the state objects present the main public interface to receive updates from another component. The navigator will, for example, access the current track position and direction directly on the stepper state. Stepper and navigator are essential to the propagation and therefore not run as actors. It is critical that they, as well as the actor chain, are called in the correct order and that their states are passed on correctly. This happens in the propagation loop inside the propagator class, which runs until the track has reached the end of the detector or another custom abort criterion has triggered. As a strongly simplified code example, the propagation loop performs the following conceptual steps:

```
DETRAY_HOST_DEVICE bool propagate(
    state &propagation,
    actor_states_t actor_state_refs = dtuple<>{}) const {
        // Initialize the navigation
        m_navigator.init(propagation, m_cfg.navigation);

        // Run all registered actors/aborters after init
        run_actors(actor_states, propagation);

        // Find next candidate
        m_navigator.update(propagation, m_cfg.navigation);

        // Run while there is a heartbeat
        while (propagation._heartbeat) {
            // Take the step
            m_stepper.step(propagation, m_cfg.stepping);

            // Find next candidate
            m_navigator.update(propagation, m_cfg.navigation);

            // Run all registered actors/aborters after update
            run_actors(actor_states, propagation);

            // And check the status
            m_navigator.update(propagation, m_cfg.navigation);
        }

        // Pass on the whether the propagation was successful
        return propagation._navigation.is_complete();
```

The reason why the navigator has to run again after the call to the actors is that, in principle, any actor can change the track state, as is done in the filter step of a Kalman filter.

6.3.2. The Caching Navigator Implementation

In principle, any kind of navigation may be implemented in DETRAY, as long as it provides the information to the propagation workflow that was described in the previous section. However, the main navigator implementation in DETRAY closely follows the ACTS navigation model, which was introduced in Chapter 4, where the navigation moves between detector volumes and performs a *local* navigation over the surfaces contained inside a given volume by querying acceleration data structures that are registered with that volume. The acceleration data structures currently available in DETRAY, which are the brute force method and the surface grids, and their implementation as part of the navigation flow, have been presented in Section 6.2.

The navigation flow switches between detector volumes by exiting and entering through dedicated *portal* surfaces, which are the boundary surfaces of the detector volumes and carry the volume index of neighbouring volumes as part of their masks (navigation link). The portal surfaces are always tested for reachability as part of a brute force search method rather than a surface grid, so that the exit portal of a given volume can always be resolved. In light of the fact that the portal surfaces are expected to be relatively few in number compared to the sensitive surfaces in the detector, namely for the most common case of a cylindrical detector setup two disc and two cylinder portals per volume, the reliability of the portal search has been prioritised over ultimate performance by simply checking all of them.

Once the next detector volume has been found, which happens when the track state reaches an exit portal, the navigator will set the current tracking volume index, which it keeps as part of its state, to the navigation link provided by the portal. This also automatically switches the access to all other information in the detector to the new volume descriptor and its corresponding set of links. Specifically, the accelerator link of the new volume descriptor will point the navigator to a new set of acceleration data structures which will return the portals and neighbouring surfaces of the new navigation domain. From these, it will determine the next target surface and its distance to the track state, starting with the entry portal that the track is currently on.

The exit portal of the preceding volume is automatically skipped and will not be communicated as a valid candidate to the actor chain, since it contains identical information to the entry portal of the new volume and would thus trigger duplicate processing in the actors. For example, it could lead to counting material that has been mapped to the portals twice. The two adjacent portals must carry identical information, however, since during backwards navigation along the track, which can be useful for smoothing of the track parameters after track finding, the former entry portal becomes an exit portal. With symmetric portals, the navigation logic can remain the same, even if the stepping direction is reversed.

In DETRAY, all types of detector surfaces are part of the same navigation flow, instead of switching between portal and sensitive/passive surface navigation modes, which reduces the code complexity and simplifies the memory allocation scheme for the navigation cache. To illustrate this principle, the navigation flow can conceptually be thought of as following a graph, where the detector volumes represent the nodes and the masks of the detector surfaces the directed edges. The surface masks connect their mother-volume with another volume, with the direction defined as pointing from the mother-volume to the target volume. The target volume does not have to be a distinct volume, however. A portal connects its mother-volume to one of its neighbours with which it shares the corresponding boundary surface, while any other surface types contained in the volume constitute a loop. The navigator will follow the navigation links that are part of the surface masks whenever the track states reaches a surface and either stay in the current volume or initiate a volume switch, if the link differs from the current volume index. This way, no deeper insight about the surface type is needed for the navigation flow. Once a DETRAY detector has been built, the volumes and surfaces it contains can be iterated and a `detray::volume_graph` object can be generated from them by parsing the navigation links on every mask. The volume graph internally produces an adjacency matrix, which is a quadratic matrix with a dimension that equals the number of detector volumes plus the conceptual *world volume* in which the detector itself is contained. The latter is indicated with an invalid navigation link (all bits of the index set to one) and signals to the navigator that the end of the detector was reached and the navigation should be terminated successfully. The connections in the volume graph can be printed and offer a first way to check, whether a detector instance has proper navigation linking and can be navigated correctly. Alternatively, the graph can be visualised using the `dot` format [220], as is shown for a small toy detector configuration in Figure 6.20. Each edge should have connections in both directions on the respective node, which means that the portals can be traversed in both directions. The exception is the *world* node which is a sink, as all tracks exit the detector geometry at this node. The graph visualisation can, however, quickly become confusing for larger detector geometries.

Fast Navigation using Ray-Surface Intersections

As is the case in ACTS, DETRAY performs a fast navigation with additional candidate caching in order to find viable candidates that are compatible with the path of the track among the surfaces that are returned by the acceleration data structures. For this, the distance to the candidate surfaces is approximated by a straight-line intersection using the tangential of the track, which is given by the direction of the global track state at the current position. Those surfaces that were successfully intersected, meaning where the intersection position lies inside one of the surface masks, are saved in the internal cache of the navigator to be re-evaluated at subsequent steps.

This approach has the advantage, that intersections between a straight-line or ray and various different detector surface shapes can be calculated analytically and are hence, in general, more efficient and reliable than numerical approaches. Furthermore, the caching allows to limit the number of calls to the acceleration data structure and to keep updating, for example, only the next closest surface candidate in the cache. This will be detailed in the following section. In order to dispatch the correct intersection algorithm for every surface shape, a compile-time specialisation of the `ray_intersector` is called, based on the local coordinate frame of the shape and the linear algebra implementation used in the detector.

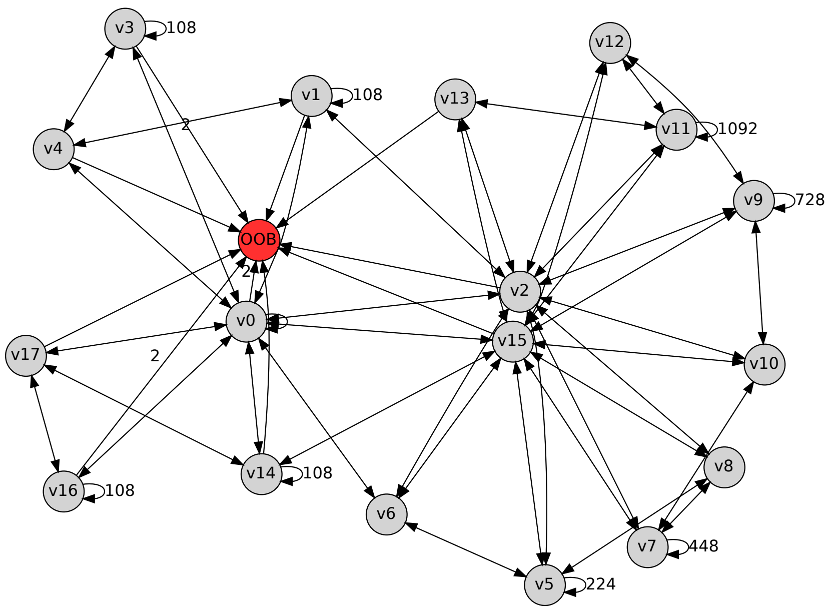
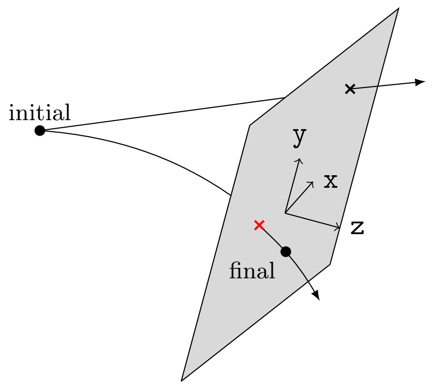


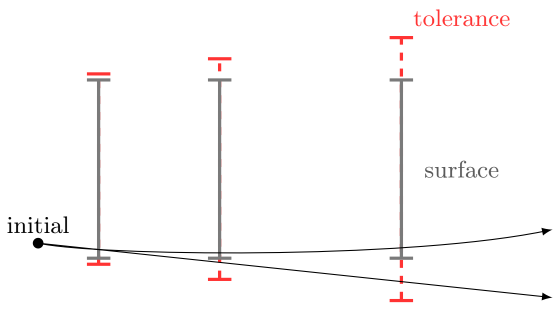
Figure 6.20.: The volume graph for a toy detector geometry in a configuration with four barrel layers and two endcap layers on either side, the same setup as in Figure 6.8. The labels on the nodes match the volumes indices, for example, $v0$ describes the beampipe volume. The nodes on the right are the barrel layers, containing between 224 and 1092 sensitive surfaces (represented by the loops), and their intermediate gap volumes. The nodes with 108 loops correspond to endcap layers.

The premise behind the fast navigation is, that over short distances and for sufficiently high momenta, the trajectory of a particle traversing the detector is approximately a straight line. However, if one of these two conditions is not met, inaccuracies in the navigation can occur, either by misjudging the distance to the next surface or by passing it altogether at the mask edge. These effects are expected to be aggravated for either low momentum particles, with a strong track curvature, or by long stepping distances performed by the Runge-Kutta stepper, over which even a reasonably high momentum track can exhibit a notable deviation from a straight line.

The latter issue can be improved either by providing step size constraints that limit the distance by which the stepper can advance the track state by to a certain predefined length, or by carefully designing the tracking geometry. Each surface reached during the navigation always triggers a re-evaluation of the track position and direction and a new step to be taken by the stepper. Consequently, the initial stepping distance to reach the first surface can be kept low, and issues introduced by the track curvature can be reduced, by placing the volume portals in close proximity to the contained sensitive surfaces. In



(a) Overstepping happens when a surface is tilted towards the track direction, so that the distance is overestimated.



(b) Mask tolerance scaling compensates for the displacement of a curved trajectory from the straight-line at large distances.

Figure 6.21.: Visualisation of problems that can arise due to the linear approximation of the track direction during candidate search in the navigator.

between the volumes that contain sensitive surfaces, gap volumes are placed in order to keep the detector geometry fully connected by portal crossings. These gap volumes can be traversed with less navigation accuracy, since they do not contain surfaces that need to be resolved during local navigation.

The first problem to mention with regard to the track curvature is what is called *overstepping* in ACTS. The distance between the track position and a ray intersection point on a candidate surface could at first glance seem to be a lower bound for the stepping distance that needs to be taken along the curved track in order to reach the surface. However, if there is any point on the surface closer to the track position than the intersection point of the ray, either because the ray is not parallel to the surface normal or because the surface shape is not convex from the point of view of the track, the distance given by the ray can be an overestimation of the true stepping distance, as shown in Figure 6.21.

This will become a problem for the navigation once the stepper advances the track state beyond the true intersection point, leaving the surface behind the next track state position. A navigator that exclusively searches for surface candidates in the forward direction along the track momentum will then miss the surface as a candidate in the next update and mark it as unreachable. This could lead to the track "teleporting" through a valid candidate, unless the navigator is allowed to look for candidates a certain distance behind the current track position. This *overstepping tolerance* has to be limited though, so that a track is able to leave a surface candidate in case the adaptive RKN algorithm mandates a valid but small first step away from the surface. In the event that overstepping occurs and the navigator finds the valid candidate behind the track within the overstepping tolerance, the distance to the surface will be negative, prompting the stepper to automatically move backwards onto the surface, so that the navigator can properly communicate the surface encounter to the actor chain. Another issue that has led to missed surfaces during fast navigation, happens during the boundary check of the surface mask and is also related to the small distance between the ray intersection point and the true intersection point on the surface. It is possible, that the navigation ray misses the surface by only a small margin, while the curved track actually bends towards the surface center and would intersect it successfully. Adding a scalable *mask tolerance* to the mask's boundary check can help reducing this effect.

The idea here is to set a comparatively large mask tolerance if the track position is still far away from the candidate, so as to compensate for the deviation from the track path from a straight line over large stepping distances. This allows the navigator to keep more candidates in the cache for a while, until it can be determined whether the track actually encounters the surface. This is done using a tight mask tolerance when the track is very close to the surface, ensuring that not too many "false positive" candidates are being communicated to the actor chain, which could increase computing resource consumption. To achieve this behaviour for the mask tolerance, an initial tolerance value is scaled with the straight line distance to the surface in between a minimal and a maximal mask tolerance. This is shown in Figure 6.21. During track finding, when there is still a large uncertainty on the track position and direction, the minimal mask tolerance can be set higher in order to account for this uncertainty.

Updating the Navigation Cache - Trust Levels

Already in the initial implementation of DETRAY, the navigator held an internal candidate cache that could be updated in different modes, as determined by a newly introduced *trust level* flag. This cache is filled with candidate surfaces that have been returned by the acceleration data structures of the current volume and were then intersected successfully, as discussed in the previous section. The cache is sorted by the intersection distance and the closest candidate is chosen as the next target surface for the track to reach. It uses a fixed sized array with a compile-time capacity, which means that excess surface candidates cannot always be included. In this case, the cache will be filled from the acceleration structure again once it is empty and no portal has been reached.

The cache has been implemented as a `detray::range` over the surface candidates, which means it can be iterated between the first and the last reachable candidate. Since the cache is sorted by distance, the beginning of the range is always the next target surface for the track and the end is the furthest candidate in that volume that is still compatible with the track direction. Surfaces that have already been encountered and handled by the actor chain stay, in general, before the "next" surface iterator position as it gets advanced to the following candidates. However, if the entire cache is re-evaluated, candidates that have been determined to be unreachable will be set to an infinite distance explicitly and will consequently be put behind the last reachable candidate during subsequent candidate sorting. An example cache state is shown in Figure 6.22.

After the initial setup of the cache, the stepper and any of the actors can make changes to the track state and the candidate information in the cache needs to be updated. Both the stepper, as well as the actors, can flag to the navigator how severe the change is that

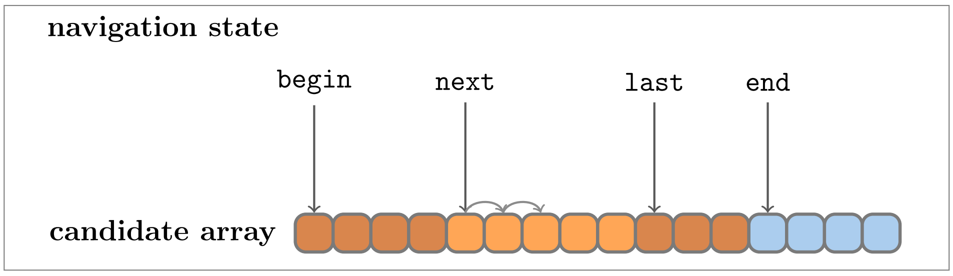


Figure 6.22.: The cache of the navigator keeps those intersection results in an internal array that are found to be compatible with the current track state by ray-surface intersections. The dark nodes represent candidates that have already been visited by the track or have otherwise become unreachable. The light orange nodes are those candidates that are actively being navigated through (comprising all types of surfaces) and the blue nodes are empty slots in the array.

was made to the track state, if any change was made at all. These are the aforementioned trust levels, which communicate to the navigator how much update work has to be done in order to restore a consistent navigation state. Since the trust level models how reliable the information on the next candidate still is, it can only ever be lowered by the individual actors, never raised, until the navigator runs and resets it to *full trust*. It should be noted that only the trust in the next candidate is restored after a navigation update. It depends on the trust level, whether the other candidates in the cache will be updated or not. In general, the data in the cache should therefore only be accessed through dedicated methods on the navigation state, since they can interpret the current navigation state correctly. In the following, a brief overview over the different trust levels is given:

- ***Full trust:*** Nothing has been changed by the actors and the navigator will therefore not perform any updates. After the update steps for any of the other trust levels, the navigator will reset the trust level to *full trust*.

- ***High trust:*** Only update the next target surface of the track. This is the most efficient cache update mode, since it is restricted to a single ray surface intersection, but should only be applied if the track has advanced along the expected path without big changes to its direction. If the target surface, or the succeeding surface, is no longer reachable, the trust level will be escalated [221].

- ***Fair trust:*** All candidates will be re-evaluated and the cache sorted again afterwards. Unreachable candidates will be sorted to the back of the internal array and the end of the candidate range will be set appropriately. If the cache is exhausted after this update, meaning that all candidates have become unreachable, the trust level is escalated to *no trust*.

- ***No trust:*** Full initialisation of the navigation state in the current volume. This update queries the acceleration data structures of the volume, fills the cache with new candidates and sorts them.

How to actually determine the appropriate trust level after a change to the track state is up to the actor that made the change. The DETRAY steppers can be equipped at compile-time with different methods to evaluate the trust level after an update. These are called *navigation policies* and with them the behaviour of the navigator can be adapted to different modes. For example, an *always init* policy will force a complete re-initialisation of the current volume after every step, while the default policy only reduces the trust level to *fair trust* if the step size that a stepper tries to take hits one of the potentially configured step size constraints. A dedicated Runge-Kutta policy exists for the corresponding stepper, that will determine the trust level according to the amount by which the step size was reduced.

Both the default, as well as the Runge-Kutta policy, operate on the idea, that the constraint or alteration of the step size is indicative of changing conditions in the propagation, for example, in the geometry or the magnetic field. This, in turn, might warrant a more thorough re-evaluation of the navigation state, as well. On the other hand, running the full initialisation of a volume often, is expected to be computationally costly and should ideally be kept to a minimum, since it has to query the acceleration data structures, evaluate all resulting candidate surfaces and perform the sorting of the cache.

6.3.3. Summary and Outlook

The DETRAY caching navigator follows many of the design principles of the ACTS navigation, such as using straight-line intersections for candidate evaluation and candidate caching to minimise queries to the acceleration structures. Inaccuracies that result from the linear approximation to the particle trajectory are compensated by adding a number of tolerances to the candidate search. In DETRAY, the navigation flow does not distinguish between surface types and strictly only moves along navigation links between volumes in the detector. The scrutiny with which the navigator updates the cache can be influenced via trust levels, which can be used to define navigation policies through which the stepper can steer the behaviour of the navigator. The combined handling of surfaces and portals, as well as trust-level based state updates, are being considered to be re-integrated into ACTS.

There are a number of open issues with the DETRAY navigation at the time of writing, to a large degree due to missing features. Currently missing, for example, is a volume grid in the detector, which would allow to associate a track state position with a detector volume. This is required in case the propagation does not start in the volume with index 0, which is currently where the navigator will start by default. Consequently, this has to be the volume that contains the Interaction Point, where the majority of particles originate in a collision experiment.

A volume grid is also needed to recover navigation flows that got lost in the detector, for example, by overstepping beyond the overstepping tolerance through a portal into an adjacent volume. In this event, the volume switch will be missed and no surfaces from the new volume will be available to the navigator. When the navigator is unable to find any candidates, it currently tries to intersect surfaces again with a very large overstepping tolerance and aborts the propagation if still no surfaces can be found. In the future, it could query the volume grid in order to determine the current volume and access the corresponding surfaces in the detector again.

A number of smaller issues remain, such as the question of how to determine the next target candidate in case of overlapping surfaces in the detector and how to handle convex surface shapes that return more than one intersection point with a ray. In the category of performance related questions, an interesting study will be the optimisation of the number of portal intersections. The computation of the intersection between a ray and a cylinder portal is comparatively complex and hence expected to be costly. Yet, due to the way the volumes need to index their surfaces in the flat DETRAY detector containers as contiguous ranges, every portal has to be registered with both adjacent volumes. Therefore, both exit and entry portals will be intersected during a volume switch, even though the surfaces are identical. By associating the portals to their respective neighbours, for instance by adding the corresponding identifier to the masks, the second intersection might be skipped and the number of ray-cylinder intersections could be reduced. Similarly, portals that are part of the same volume, and lie side by side on the same surface, are intersected independently from each other at the moment. In the future, the surface should be intersected once and then the masks should be checked one after the other.

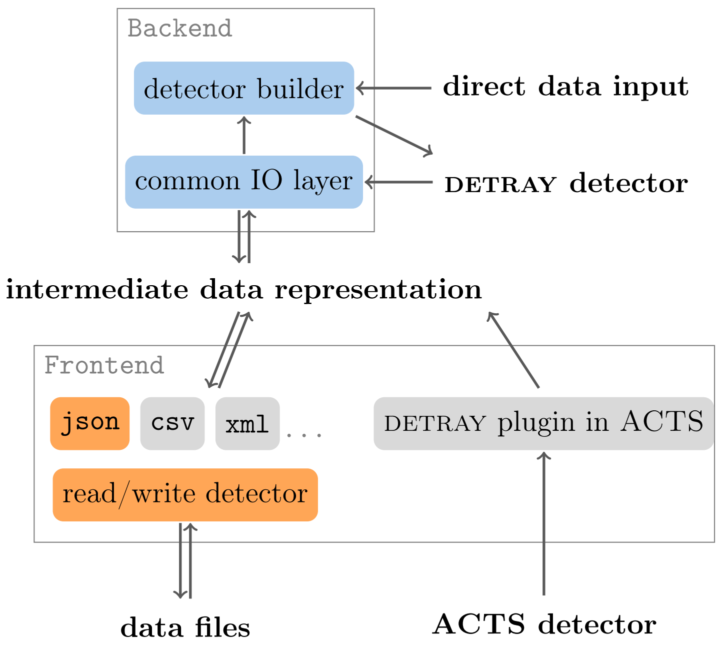


Figure 6.23.: Internal structure of the DETRAY detector IO implementation. The greyed parts are not (yet) implemented or only partially available. Of these, the plugin in ACTS is intended to become the main IO road for DETRAY, while the json file IO is currently used to read the detector data in the TRACCC project. The `csv` and `xml` modules were added for completion to highlight the design intention of the file IO implementation.

6.4. Detector Building

The geometry building and IO functionality in DETRAY has to cover multiple different use cases and was hence designed in a highly flexible and modular way. It covers both the internal construction of a detector instance from input data (backend), as well as the reading and writing of data from and to file (frontend), as shown in Figure 6.23. The backend and frontend are decoupled, so that the frontend can be switched between, for example, different file formats. A custom frontend is also meant to be used for the DETRAY plugin in ACTS, where the geometry data will be converted directly from ACTS to DETRAY without the use of intermediate data files. In summary, the IO infrastructure needs to support:

- a native detector construction within DETRAY (e.g. for the testbed detectors),
- a general detector construction and IO from and to data files,
- a direct detector conversion from ACTS to DETRAY.

The frontend contains reader and writer functionality for the given data sources to and from an intermediate data representation of the respective detector components.

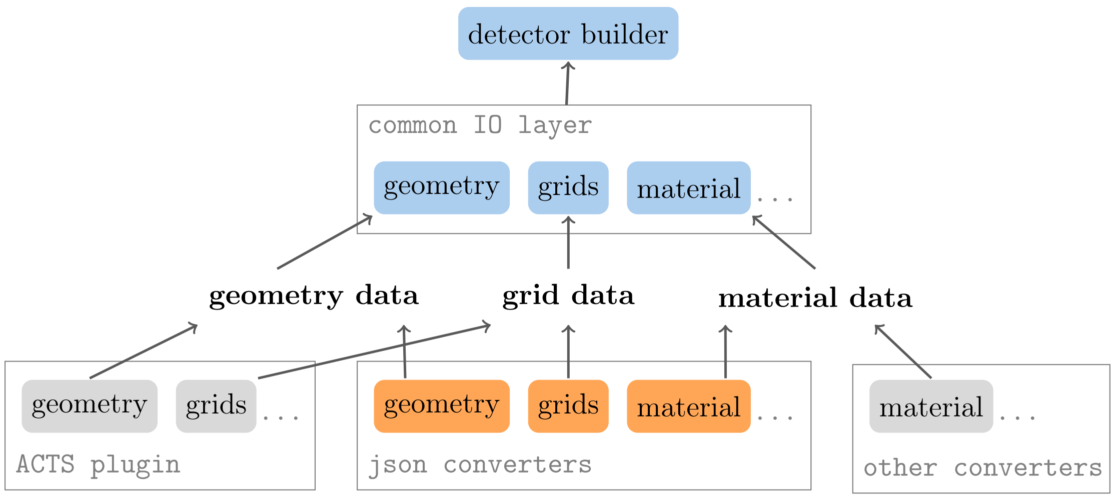


Figure 6.24.: Each detector component has a separate data path and matching data files within the respective frontend implementation, so that they can be built independently from each other. For every component, there exists a corresponding backend implementation to translate the data representation for the detector builder. The greyed out frontends are not yet (fully) implemented.

The file IO module in the frontend is made up of several converter plugins that each handle a specific data source and are managed by the detector reader and writer functions, which in turn provide the main user interface. The intermediate data representation can then be fed to the backend, which will aggregate and pre-process the data and afterwards assemble it to a complete detector instance in a final building step. For detector geometries that are built natively in DETRAY, like the toy detector, the frontend is simply skipped and the data is added directly to the detector builder. The detector builder is responsible for the construction of a concrete detector instance and is connected to the frontend via a common IO layer that will correctly fill the detector data into the builder from the intermediate data representation.

In addition to the possibility to exchange the frontend, the building process is divided between the different detector components, like material or acceleration data structures, which allows to include and remove these components in the building process at runtime, depending on whether or not the corresponding data file is given to the detector reader. This makes it possible for a detector to be built, for example, with different kinds of material descriptions, or with no material at all, by simply replacing the material file with another one or leaving it out entirely. All the while, the other data files remain untouched and the detector configuration can be changed and tested in various setups by simply composing different input files. In principle, it would even be possible to combine different file formats for separate components of the same detector, which may be useful in case a component was only available in a specialised data format. In this case, only a small, dedicated portion of the IO would need to be reimplemented for the specific component, as shown in Figure 6.24. Only the geometry file is required at all times, since it contains essential information on the detector setup that is needed in all following building steps. The separate data files also facilitated the gradual rollout of new IO features during the principal development process of the project, so that basic functionality like the geometry IO could already be tested downstream, while more specialised functionality like the material maps IO was still work in progress.

The building of the detector instance proceeds volume-by-volume and makes no further assumptions on the detector geometry beyond the requirement that at least one volume is present. Consequently, the data in each of the different component files are grouped by volume, using the index of the volume in the detector volume container as an identifier. Any given file will contain a header in addition to the data section, which is used to check the file version and register the content type, as well as giving some statistics on the contained data. An example in `json` format is shown below:

```json
{
    "header": {
        "common": {
            "version": "detray - 0.100.1",
            "detector": "toy_detector",
            "tag": "geometry"
        },
        "volume_count": 38,
        "surface_count": 4158
    },
    "data": {
        "volumes": [
            [...]
        ]
    }
}
```

So far, a complete IO chain has been implemented for the `json` format (*JavaScript Object Notation*), using the *nlohmann* library [188] to output `json` objects from `C++`. In particular, adopting the `json` format in the data files has the advantage that it is human-readable, thus simplifying debugging the IO implementation by allowing direct inspection and also manipulation of the data content (for example, removing or editing entries). The correctness of the data file format can be checked using a `json` schema definition [222] that exists for every detector component file, for example:

```
python3 detray/test/validation/python/file_checker --geometry_file [file]
```

The corresponding `json` file export for the ACTS project has been implemented outside the scope of this thesis as part of the ACTS geometry revision. Any ACTS tracking geometry can be converted to the DETRAY `json` format, read into DETRAY and then the GPU-ready detector instance can be constructed from it. It is expected that the direct conversion from ACTS without intermediate data files will become the principal IO mode for DETRAY. The main reason for this is, that no infrastructure exists in DETRAY to convert a full geometry description from, for instance, TGeo [192] or DD4hep [168] to a DETRAY detector. Instead, the sophisticated ACTS tracking geometry building infrastructure is used to obtain a valid tracking detector instance which can then be converted to the DETRAY format in a second step. This way, the logic to convert a full geometry description to a tracking geometry description does not need to be re-implemented and maintained separately in DETRAY, while still providing a mechanism to make the geometry of every experiment that has a representation ready in ACTS also available in DETRAY without further changes to existing setups.

A first implementation for the ACTS DETRAY plugin [223] is available in ACTS for the first ACTS tracking geometry revision (called *Gen 2*) and is in the process of being adapted for the final ACTS tracking geometry version (*Gen 3*) [224], which will make it available for most experiments in ACTS.

6.4.1. The IO Backend

The main work of building the detector instance is handled by the IO backend, which is split into two layers. The detector builder layer is implemented by the `detector_builder` class and a number of *volume builder* implementations for different detector components with factory classes to fill them. Apart from orchestrating the volume builders, the detector builder will also handle the construction of the volume search data structure on the volumes it builds in the future, which in most cases will be a spatial grid. The second layer, which can optionally be on top of the detector builder, is the *common IO layer*. The common IO layer consists of a reader and writer class for every detector component and acts as a converter between the detector builder and the IO frontend.

Building a Detector Instance

The detector builder accepts a template argument for a detector type and holds a volume builder instance for every volume that should be added to the final detector instance. New volumes can be added to a detector by initialising a new volume builder in the detector builder. The collection of volume builders is kept in an internal data structure that reflects how the geometry setup in the detector should be built. Currently, only a simple vector of base class pointers to the `volume_builder_interface` is implemented, which represents the simplest detector builder setup possible and shows that no interconnections between the volumes are to be handled by the detector builder. A graph or tree data structure could be used in the future to model neighbouring volumes in the geometry, for which a portal-linking should be performed, or a volume hierarchy, where some volumes can exist as nodes of other volumes. Any of these more sophisticated builder modes only becomes relevant, however, for DETRAY-native detectors. In the current main IO mode, where complete tracking geometry instances are read from ACTS with all relevant interconnections between the volumes are already resolved, a flat vector of volume builders is completely sufficient.

Moreover, the detector builder provides a template parameter for a basic volume builder type which is responsible for building the geometric components in a given detector volume, such as portals and any additional surfaces. The idea here is, that different basic volume builders could at some point be used, for example, to build different detector memory layouts, like *Struct-of-Array* (SoA) vs. *Array-of-Struct* (AoS).

If additional detector components should be added to a particular volume, the basic volume builder for that volume can be *decorated* with another volume builder which is then responsible for constructing the respective component, such as material or acceleration data structures. A dedicated member function exists as part of the detector builder class interface to construct and then decorate a volume:

```
// Build a detector of type metadata_t
detray::detector_builder<metadata_t, detray::volume_builder> db{};

// New cylindrical volume
auto vb = db.new_volume(detray::volume_id::e_cylinder);

// Add an acceleration structure to the volume
using grid_builder_t = detray::grid_builder<detector_t, grid_t>;
auto vb_with_grid = db.template decorate<grid_builder_t>(vb);
```

Once the data in all volume decorators and for the volume search data structure has been gathered, the detector instance is built with a given VECMEM memory resource by calling the respective `build` member function of every registered volume builder in sequence. The volume builders will then make sure that all components are constructed correctly, add them to the detector global data stores and update the relevant links to global indices. After all volumes have been built, the volume search data structure will be constructed in the same step and the detector instance is returned.

Building Detector Volumes

The volume builders are implemented in a *decorator pattern* [225], which was chosen, since it allows to expand an instance of a given class at runtime with additional functionality, in this case to build optional detector components. It provides the necessary flexibility when dealing with IO data read from file, as mentioned before, but it is also highly convenient when building detectors directly in DETRAY. Not every detector setup will require all available components and it should be possible to express the building of a simple detector in a simple way. For example, a tracking geometry like that of a telescope detector, which is essentially just an array of sensors, should not become coupled to IO requirements of a calorimeter, such as the construction of volume material, unless specifically requested.

To facilitate this, all volume builders in DETRAY implement a common abstract volume builder interface towards the detector builder, as shown in Figure 6.25. Volume builders that decorate another volume builder inherit it through the `volume_decorator` class, which holds a pointer to a parent volume builder as a member and forwards all calls of its member functions to this wrapped volume builder. Any class that inherits from the volume builder decorator can then selectively overwrite those member functions that need to be expanded with new functionality, while still being able to call the respective implementation of the parent builder. Calling the `build` method on a volume builder will then result in a chained call through all layers of decoration, if any, making sure that every component is added to the volume correctly. It is up to the decorating volume builder, however, whether it will call its respective parent before or after it completes its own building task. This becomes important, since in the final detector instance, all components need to be addressable and linked to each other correctly, thus in some cases imposing interdependencies between the building processes of these components. For example, a builder for an acceleration structure might need to access the complete geometry of the volume in its final configuration as part of the global detector stores, whereas a material builder might want to inject extra information before the geometry building.

Geometry Volume Builder This is the basic volume builder implementation that is used throughout the building of all detector setups. As mentioned before, it can in principle come in different variations, for instance according to special memory layout requirements that should be observed for the detector.

The `volume_builder` collects the data from all kinds of surface factories in no particular order, as described in the following section. When the `build` method is called, it will proceed to add the entire geometry of the volume to the detector in a single iteration. For this, the volume index is automatically set according to how many volumes are already present in the detector and the volume's placement transform is added to the detector transform store. Afterwards, the index based data links of the surface descriptors are updated. The linking update needs to happen, since the global detector stores might already be filled with the data of previous building steps, which imposes an offset between the indices the volume builder holds locally and the final position of its data in the detector. On the other hand, volume-local indices are the only way to reliably communicate linking of detector components between DETRAY and ACTS, since the specific procedure to fill a detector instance (including possible data deduplication and sorting steps) is as of now inaccessible to ACTS when it exports its tracking geometry.

When everything is correctly updated, the surfaces are added to the detector by appending the transforms, surfaces descriptors and masks. This can proceed in a straightforward way, since the volume builder internally keeps the same kind of transform and mask stores as the detector. Depending on whether or not the volume builder was decorated with a builder for an acceleration structure, it will either add only its portals or all surfaces to a new *brute force searcher*, as discussed in Section 6.2, which it appends to the detector acceleration store. Once the surfaces are added to the detector stores,

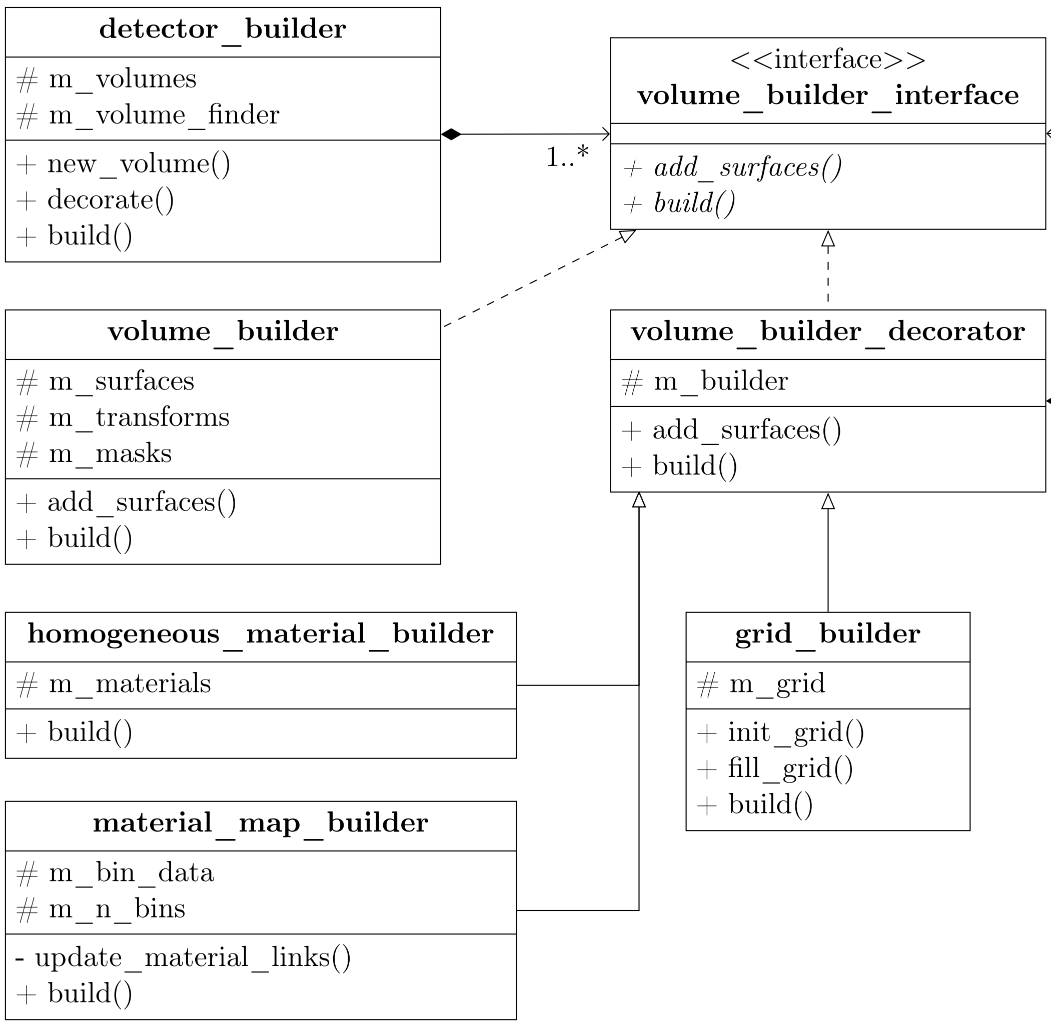


Figure 6.25.: Class diagram showing the relevant parts of the DETRAY volume builder design: The detector builder consists of volume builders that can either be a basic volume builder or a decorated version of it. The decoration happens via extending the `volume_builder_decorator` class, which wraps an implementation of the `volume_builder_interface`.

the index ranges of the volume descriptor are updated to be able to reference the correct surface descriptors later on.

As a final step, the geometry volume builder will then pass on a pointer to the current volume under construction in the detector to the following volume builder decorators, which can use it to add further data for any optional detector components.

Surface Grid Builder This builder is tasked with the creation of the surface search grids that were described in Section 6.2. It uses a grid factory instance, which will be introduced in greater detail below, to obtain an instance of a particular grid type. The grid is subsequently filled either in an automated way according to a prescription given by a `bin_filler` class or by hand through the public class interface of the grid itself.

The main difficulty for the grid builder is that the grids register the surface descriptors directly, so that the linking they carry needs to remain synchronised with the surface descriptors that are registered in the detector surface lookup. For this reason, the basic volume builder has to build the geometry with all linking offsets already resolved before the grid builder can finalise the grid for that volume in a second step. Depending on whether the grid is still empty when the build method is called, it will either dispatch its bin filler to insert the surface descriptors into the grid or, alternatively, it will complement the surface descriptors that were pre-filled by the common IO layer with the final surface linking information.

Several bin filler implementations are available, which can either fill a surface using the position of its centroid, a given local bin index or by matching the surface contour onto the grid binning. The default for the grid builder is the filling by centroid position, which is used, for example, for the building of the toy detector, while the `json` based file IO populates the grids directly.

The builder finishes the grid construction by copying the grid into the corresponding grid collection of the detector and registering it in the volume descriptors accelerator link. Throughout the building process, a memory owning grid type is used to fill the bin data. With the copy of the grid to the detector, the management responsibility of the grid's memory is passed on to the grid collection, which will subsequently only instantiate non-owning grid types from its global storage, as explained in Section 6.2.

Homogeneous Material Builder If a detector is equipped with a homogeneous material description, as detailed in Section 6.1.5, those volumes that contain material surfaces will need to have their builder be decorated by a `homogeneous_material_builder` in order to add the material slabs or rods to the surfaces.

Since its corresponding material factory will resolve the material links of a given surface that the builder holds, the builder later simply needs to update the links according to the global detector data store offsets and add the material to the detector.

Material Map Builder The material maps in ACTS are based on grids that hold a material slab per bin, as discussed in Section 6.1.5, and hence for every volume the material map builder has to construct a range of different grid shapes that fit onto their respective surfaces. Depending on whether it is supposed to build a surface or volume material representation it might need to have to switch between two or three dimensional grids.

Since the surface grid builder that was described above is tailored to constructing and updating grids that hold surface descriptors for navigation, and because decorating a grid builder for every material map in a volume could become computationally inefficient, the material map builder was implemented as a dedicated class that does not wrap single grid objects of various types to be filled. Instead, it keeps the binning data for all material maps for a specific volume in an `std::map` container. The data in the map container is associated by the index of the geometry object the material belongs to. From this data collection it will then proceed to build all material maps for the volume in a single call.

At first, the material map builder needs to pre-calculate the correct material links for all surface descriptors and update them before any other builder is called to ensure consistent linking. Using the grid factory, it can afterwards directly fit the material grid onto the respective surfaces by their masks, with only the number of bins needed as additional information. Once the grid factory has constructed an empty grid that fits the surface, the material map builder will populate it from the material data and their local bin indices that it holds internally. Finally, it will add the material grid to the global material grid collection of the detector and perform a last check, that ensures its pre-calculated material link matches the position the map was eventually copied to.

The internal data stores of the builder should be reusable to, in the future, build three-dimensional volume material maps, but this time registering the data under the volume index instead of the surface indices. A separate build interface could then be called to register the three dimensional material grid with a volume instead of the detector surfaces.

Data Factories

The detector components are created and added to a detector instance with correct linking by the volume builders from an internal data store, as outlined above. In order to fill these internal data collections of the different volume builders with consistent, pre-processed data, dedicated data factory classes are used in DETRAY. Different factories exist for the different detector components. They can be broadly categorised into those that aggregate data from a data source, for example from the common IO layer, and those that generate their product on the fly according to a given prescription.

To this end, most data factories possess a call operator that takes the respective data stores of a volume builder and fills them with coherent data that already resemble the data layout of a detector. For instance, surface data are already split into transforms, mask boundary values and any linking information needed by the geometry volume builder. From this, the respective volume builder can assemble higher level data objects or simply insert the data into the global detector data stores with correctly updated linking.

Even though the data factories will either accumulate or generate the data close to the way they need to be added to the detector data stores later on, those factories that are themselves filled by an outside data source expect the data chunks to come in a well defined and more object-oriented way. Each such factory will therefore define a data type that holds all necessary components for a single instance of the factories product, so that a convenient and intuitive data handling becomes possible when filling the detector data by hand into the IO backend. These data types are akin to the payloads of the intermediate data representation that links the IO frontend with the backend, but were developed independently and already contain higher-level objects instead of raw data. The payloads are also independent of any concrete DETRAY types, like the linear algebra implementation, to facilitate an easier integration into other projects, while the factory data can be filled conveniently using built-in types when working inside DETRAY.

The factories that produce surface data for a geometry volume builder implement the `surface_factory_interface` so that they can be bound to the `add_surfaces` method of all volume builders. Similar to the way the volume builders are implemented, these surface factories can be decorated with extra functionality, for instance, to build material for their surfaces. The exception from this is the grid factory, since grids are frequently not associated to a surface but instead to a volume. In the following, the most relevant data factory implementations will be discussed briefly:

Surface Factory This is the main factory that aggregates geometry data from either the IO frontend or by getting filled by hand. From the object-oriented surface data that it receives, it will sort the transforms, mask boundaries etc. into separate collections, making sure to keep the data consistent between the collections for the user automatically. The surface factory can hold data for portals, sensitives or passive surfaces, but they all need to be of the same shape, so that the data store they will be added to can be determined at compile-time. As usual, this is done by templating the surface factory on the corresponding shape type, for example:

```
using factory_t = detray::surface_factory<detector_t, detray::rectangle2D>;

auto rec_factory = std::make_shared<factory_t>();

// Add data for a 10x8mm rectangle at position (0, 0, 0), linking to volume 1
rec_factory->push_back({detray::surface_id::e_sensitive,
                        transform3_t(point3_t{0.f, 0.f, 0.f}),
                        1u,
                        std::vector<scalar_t>{10.f, 8.f});
```

When its contents are to be added to a volume builder, the surface factory distributes them correctly into the different data collections of the volume builder, including building the appropriate mask instances. If a specific ordering of the surfaces is passed to the factory via a collection of index positions, it will observe this ordering when filling the volume builder data stores. The material link of a surface is left uninitialised at this point, as this is handled by the dedicated material factories.

Homogeneous Material Factory As a surface factory decorator implementation, the homogeneous material factory will add either a material slab or rod to a given surface, depending on the shape of the surface mask. It has a dedicated method through which material data can be added for a specific surface via the surface volume-local index. This is necessary, since the detector components are read independently from each other during file IO and the surface data cannot always be passed in conjunction with its material data. Instead, a pre-filled surface factory can be given to the material factories to then add material to the surfaces later. Similar to the surface factory described above, an index based ordering can be given to the homogeneous material factory which will be observed when filling the data stores of the homogeneous material builder. Since there may already be surfaces in the homogeneous material builder present from a previous factory run, and because not every surface is guaranteed to have material, an index offset between the factory material data and the builder surface store has to be observed. The factory then needs to facilitate that only certain surfaces will be selected to get their material link set, all the while making sure that the ordering of the material is kept in case it was requested. If no concrete ordering is passed, the material is added to the trailing surfaces in the homogeneous material builder data store. Surfaces can be built with material, as shown for trapezoids in the following example:

```cpp
using tpz_factory_t = detray::surface_factory<detector_t, detray::trapezoid2D>;
using mat_factory_t = detray::homogeneous_material_factory<detector_t>;

auto tpz_factory = std::make_unique<tpz_factory_t>();
auto mat_tpz_factory = std::make_shared<mat_factory_t>(std::move(tpz_factory));

// Add data for a trapezoid passive surface, linking to volume 15
mat_tpz_factory->push_back({detray::surface_id::e_passive,
                            transform3_t(point3_t{0.f, 0.f, 1000.f}), 15u,
                            std::vector<scalar_t>{1.f, 3.f, 2.f, 0.25f}});
// Add data for a silicon slab with 1mm thickness for the last surface
mat_tpz_factory->add_material(detector_t::materials::id::e_slab,
                              {1.f * detray::unit<scalar_t>::mm,
                              detray::silicon<scalar_t>{}});
```

Grid Factory This factory does not implement the surface factory interface, as it is uniquely used to produce stand-alone grid instances. Given a bin and a serializer type, the grid factory can produce grids of any shape, which it can deduce from the mask of a surface. This can be especially handy when building the grids needed for material maps, as they have to be fit onto the surfaces of a given volume. The mask boundary values will, in this case, determine the spans of the grid axes, and only the number of bins per axis needs to be passed as additional information.

The grid factory can, however, also be used to build more complex types, where the axis boundary and binning types are given explicitly. In principle at least, the building of N-dimensional grids would be possible, if a corresponding N-dimensional coordinate frame and serializer were available. In the following example, the construction of a disc-shaped grid using the grid factory is shown:

```cpp
// Generate grids a with single integer value per bin
vecmem::host_memory_resource host_mr;
auto gr_factory = detray::grid_factory<detray::bins::single<int>,
                                       detray::simple_serializer,
                                       algebra_t>{host_mr};
```

```
// Disc surface mask with 20mm radius
scalar_t r{20.f * detray::unit<scalar_t>::mm};
detray::mask<detray::ring2D, algebra_t> disc{0u, 0.f, r};

// Matching disc grid with 5 bins on the r- and 10 bins on the phi-axis
auto disc_grid = gr_factory.new_grid(disc, {5u, 10u});

// Put value '42' into the bin with global index 3
disc_grid.template populate<detray::replace<>>(3u, 42)
```

Material Map Factory Since the access to the fully built surfaces can only happen after the geometry volume builder has run, the material map factory collects the data needed to build the different material grids in common containers until the material map builder fits them onto the respective surface using a grid factory. In particular, this includes the number of bins per axis, the parameters and thickness for the material slabs that will be inserted into the bins of the material grids, as well as a mapping of the material data to the local bin indices of the respective grid. The data are associated to the surface by the volume-local surface index, in a similar fashion to the homogeneous material factory.

When called by the corresponding material map builder, the factory will first add any surface data that it collected and then continue with the construction of the material slabs from their components and combine them in a small data type with their respective local bin index. This happens for every grid separately and the resulting material data collection is then registered with the corresponding surface index to the internal data container of the material map builder.

In a last step, the type identifier for the respective material map is added to the surface descriptor, so that the material map builder can proceed with the pre-calculation of the material links.

A number of surface factories besides the ones that were presented above have been developed, which are mostly surface and material generators used to construct the testbed detectors. For instance, the telescope test detector creation relies on a telescope surface generator that places a number of surfaces of a given shape at specific positions along a pilot track. These are then enclosed by portal surfaces using the `cuboid_portal_generator`, which automatically fits an axis-aligned bounding box around a group of surfaces and constructs rectangular portals according to the dimensions of the box. Concluding, a minimal example from the DETRAY tutorials of the construction of a detector with a single cuboid volume filled with square surfaces is given:

```
// Build a detector of type metadata_t with a cuboid volume
detray::detector_builder<metadata_t, detray::volume_builder> db{};
auto vb = db.new_volume(detray::volume_id::e_cuboid);
```

```
// Generate 10 10x10mm square surfaces
auto sq_generator =
    std::make_shared<detray::tutorial::square_surface_generator>(
        10, 10.f * detray::unit<scalar_t>::mm);

// Add a portal box around the surfaces with a min envelope of 'env'
constexpr auto env{0.1f * detray::unit<scalar_t>::mm};

auto portal_generator =
    std::make_shared<detray::cuboid_portal_generator<detector_t>>(env);

// Add everything to the volume
vb->add_surfaces(sq_generator);
vb->add_surfaces(portal_generator);

// Build the detector
vecmem::host_memory_resource host_mr;
detray::detector<metadata_t> det = db.build(host_mr);
```

The Common IO Layer

Filling the data factories and detector builder by hand is not expected to be a frequent occasion and will most likely remain limited to the construction of DETRAY-internal detectors for testing or R&D. For the other two main IO modes, meaning file IO and direct conversion of an ACTS detector, an automated handling of the detector builder was deployed instead. This is done by the readers and writers of the common IO layer, which fill the data factories associated with the various volume builders from the payloads of the intermediate data representation. A dedicated pair consisting of a reader and a writer exists for every detector component, which will either convert a DETRAY detector into the intermediate data representation and hand it off to the IO frontend, or take the data from the IO frontend in a volume-by-volume fashion and add it to the detector builder.

The geometry reader, for example, will register a new volume builder in the detector builder for every volume it discovers in the input data. It will then add all associated surfaces to it by instantiating a surface factory for every surface shape that is part of the input data of that particular volume. Readers that handle optional detector components will proceed to decorate those volume builders in the detector builder for which the detector component was requested.

By the end of the data reading, the detector builder will hold a volume builder for every volume in the detector which is decorated and filled with exactly that functionality and data that is needed for its respective construction. The different readers and writers of the common IO layer will not be presented in further detail, however, a few common challenges in their implementation will be discussed in the following with a few examples.

The main challenge of this part of the detector building process is to relate the runtime input data to the detector data stores, which are by design only addressable using compile time type identifiers. In principle, this can be done by checking the runtime encoded input type identifier information against all of the possible code branches for the different detector types. However, since a detector should be easily extendable with new types at compile time, such as with new shapes or new acceleration data structure types, the code paths for these different types are generated from generic code by the compiler. This reduces the amount of boilerplate code that needs to be written and increases maintainability, but it also leads to a situation where the builder code cannot use a given type or type identifier explicitly and rely on its presence in the detector type that is under construction. For example, a telescope detector type is unlikely to define the stereo annulus shape, which is specific to the ATLAS ITk strip endcaps, so trying to compile against this shape in the IO code would fail for this particular detector type. Furthermore, the type identifier values a detector uses will differ between different detector types, since they need to be contiguous in order to correctly map to the position of the data collections in the tuple-based multi-store.

This was solved on the one hand by introducing a global type identifier scheme in the IO for all shapes, material types and acceleration data structure types that are part of DETRAY which is also shared with ACTS for consistent communication between the two projects. On the other hand, DETRAY makes heavy use of a `C++` rule called *Substitution Failure is Not An Error (SFINAE)*, which states roughly that a class or function will be discarded by the compiler without throwing an error if a failure happens during the resolution of its template parameters. This way, spurious compiler errors can be avoided when referencing a type or an identifier that is not present in a given detector, falling back on an empty default implementation. These detector type traits can then be used in the IO code to identify which code paths, for instance for the different surface shapes, can be safely compiled for a given detector type. This also enables to read in the same input data with the same common type identifiers into instances of different detector types as long as the detector type encompasses all required types.

For the construction of the grid types from global file based type identifiers, where every grid type can potentially be assembled in many different ways (for example, with different number of axes, axis boundary behaviour or binning), a more general approach was taken. The combinatorics of the number of types is so large that it becomes infeasible to write a type trait for every one of them. In this case, the compiler will in fact automatically generate code paths for all possible grid types by using a recursive conversion function that calls itself with updated template parameters, depending on the comparison of the runtime type identifier.

In the base case, when the list of identifiers to compare is empty, one of its template parameters will contain, for example, all of the binning types of the grid, which can then be used to conveniently assemble the requested grid type. The filling of the grid will only commence, if the fully assembled grid type is part of the detector type under construction. While this approach is more generic and very concise to write, it comes at the cost of the complier actually having to instantiate code for every possible grid type, even if that grid type does not exist in the detector in the end, which can increase the memory and CPU resources needed during compilation by a lot.

6.4.2. The IO Frontend

The main customisation point in the DETRAY IO chain is the IO frontend, which is designed as a thin layer on top of the common, DETRAY-internal backend that handles the specifics on how to securely construct a detector instance. As such, it is expected that multiple frontends may be added to the project over time, covering file IO in various formats but also direct data input, for example, as part of an ACTS plugin.

The key to decoupling the frontend from the backend is an intermediate data description that collects the data of detector components in an intuitive, hierarchical way, regardless of how the data will be stored in the detector later. This also helps decoupling the detector implementation itself from the IO. A first implementation of this was already part of the initial `json` IO implementation and has now been expanded and connected to the IO backend. For every detector component, a corresponding detector-level data payload type is available, which will hold the respective intermediate data representation in a per volume collection. For example, the `detector_payload`, which holds the geometry data, will contain a collection of `volume_payloads` for every volume and an additional payload for the volume search data structure. The volume payloads, in turn, will contain a collection of `surface_payloads`, along with further data like its name and index and so forth. The `detector_grids_payload`, as a another example, contains the data for either surface or material grids for every volume.

The main task of a particular frontend implementation is then to provide conversion functions for the IO data format to and from the intermediate data representation. With the nlohman `json` library, mentioned previously, this is already formalised as the `from_json` and `to_json` function overloads, which will be picked up automatically for custom types during the `json` IO stream. In order to connect to the IO backend, the DETRAY `json_converter` has been designed as class template that takes a backend reader or writer implementation as a parameter, which were described in the previous section. Consequently, a concrete `json` reader or writer for a particular detector component can be gained by simply plugging one of the backend IO types into the `json` frontend, as shown below:

```
using namespace detray::io;

// Read the tracking geometry from file in json format
using json_geometry_reader = json_converter<detector_t, geometry_reader>;

// Read a homogeneous material description from file in json format
using json_homogeneous_material_reader =
    json_converter<detector_t, homogeneous_material_reader>;

// Write the tracking geometry to file in json format
using json_geometry_writer = json_converter<detector_t, geometry_writer>;
```

```
// Write a homogeneous material description to file in json format
using json_homogeneous_material_writer =
    json_converter<detector_t, homogeneous_material_writer>;
```

The `json` reader for a given detector component will, in its implementation, call the correct conversion function overload on the `json` object it reads from its specific file, and pass the resulting payload on to its backend reader, as shown in Figure 6.26. The `json` writer, on the other hand, will call the backend writer first to obtain the respective payload for a particular detector component and subsequently pass it on to the matching `json` converter for file output. A similar solution that connects conversion functionality with a call to the IO backend will need to be provided for any additional IO frontend that is to be added to DETRAY, reducing the workload of the implementation of new IO plugins mainly to the conversion functions themselves.

In addition, implementing abstract IO interfaces allows any concrete reader and writer types to be picked up by the user-facing detector reader and writer in case the corresponding file format was requested for IO. While the detector writer function can infer from the detector type which detector components need to be written to file, the detector reader registers a detector component reader only if the corresponding input file was passed to it. It does so by peeking at the file header, which will tell it which kind of data is contained in the data section of the file. The final writing and reading of a DETRAY detector can be done by calling the corresponding IO function with a given configuration like this:

```
// Read the detector 'det' from a number of input data files
detray::io::detector_reader_config reader_cfg{};
reader_cfg.add_file("detector_geometry.json");      // required
          .add_file("detector_surface_grids.json"); // optional
          .add_file("detector_material_maps.json"); // optional

vecmem::host_memory_resource host_mr;
auto [det, names] =
        detray::io::read_detector<detector_t>(host_mr, reader_cfg);

// Write the detector back to a different set of json files
detray::io::detector_writer_config writer_cfg{};
writer_cfg.format(detray::io::format::json)
          .replace_files(false);

detray::io::write_detector(det, names, writer_cfg);
```

6.4.3. Summary and Discussion

The IO module of a geometry library plays a pivotal role, as it sets the stage and potentially imposes restrictions for all geometries that may be built with it. It could potentially

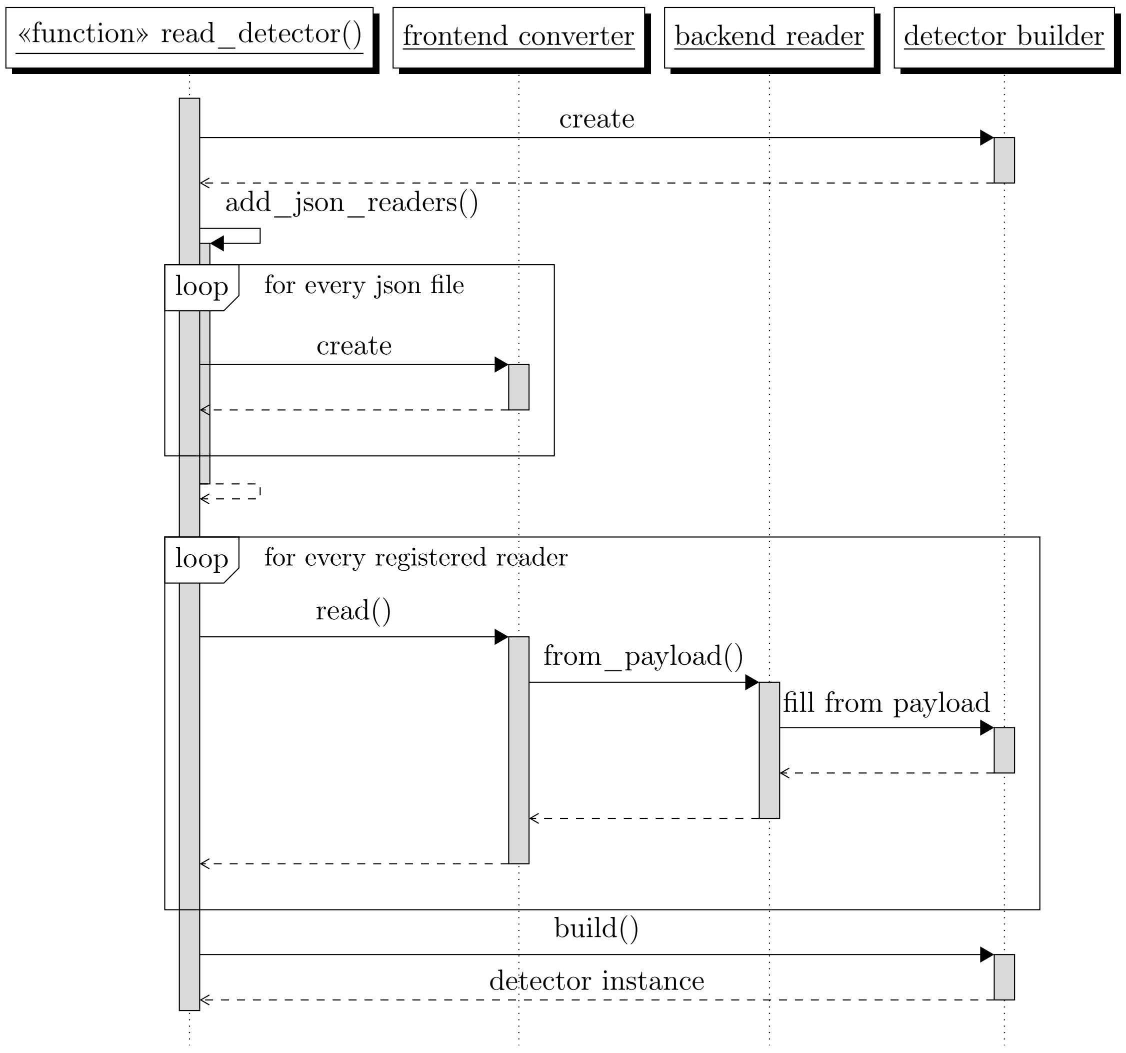


Figure 6.26.: Reading a detector from input files: First a detector builder is created and the concrete readers are registered according to the input data files that were passed (currently only `json` files). Then, the readers are called in sequence and the detector builder is filled with data. With the final `build` call, the detector instance will be assembled and returned to the caller.

even have an influence on later computing performance, for instance, due to the way the data is ordered or laid out in memory. Apart from that, it needs to be flexible enough to adapt to different setups, as geometries may need to be constructed in many different contexts, such as file IO or manual construction. The complexity of a versatile IO implementation can become challenging quickly and a careful design is needed to keep it maintainable and extendible to new use cases.

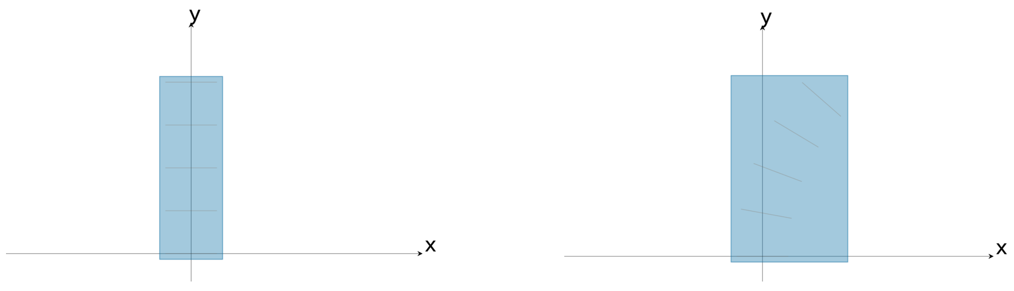
The DETRAY project has been equipped with a generic and highly modular IO implementation that meets its current and future needs. It allows for an intuitive description of a manual detector construction, as well as being able to interface to different external data sources. The composition of data files at runtime provides a straight-forward way of controlling the complexity of a detector setup during development, testing and validation, despite internally relying on a detector description that is fully compile time polymorphic. New functionality in the IO chain can be added easily by decoration and existing modules can be swapped out without further code changes to other modules.

This same flexibility, however, also comes with a few downsides, chief among them are the complexity of the implementation and the correct interconnection of data between the separate components. In order to link data reliably, a large number of indices and type identifiers is needed, which can be error prone. Index offsets and data ordering side effects can lead to data inconsistencies that are difficult to detect. To counter this, a number of unit and integration tests has been setup for each of the detector components as part of the project's Continuous Integration system, including round trip IO tests for all testbed detector types. Additionally, a detector consistency check can optionally be run following detector construction.

A number of IO features are, however, still missing, most of them performance optimisations. The current volume builder implementations, for instance, cannot perform data deduplication and will instead add the same information to a detector instance as many times as it is specified by their respective data source. This can lead to excessive memory usage in the face of tens of thousands of surfaces and hundreds of surface grids per detector. As an example, for most surfaces in a given volume, the mask or material slab in a homogeneous material description will be the same and can be put in memory once, while being linked to by many surface descriptors. Thoughts about deduplicating the index based surface grids in the endcap volumes of many silicon tracker geometries have been discussed in ACTS and might also be an interesting option for DETRAY.

Apart from surfaces and surface grids, there is also a discrepancy between ACTS and DETRAY material grids. In ACTS a material grid can span the boundaries of multiple volumes, associated to a single large boundary surface that is shared by all volumes. In DETRAY, each volume will need its own descriptor for that boundary surface, which then all link to the same material grid instance. This is in principle possible, but requires a mechanism to discover that a particular material grid has already been added to the detector by another volume builder. This makes the pre-computing of the material links in the material map builder dependant on what data has been added by other volume builders, which is information that is at the moment not easily accessible. Instead, such a material map is added to the detector by each volume builder separately and thus is duplicated multiple times.

Another issue that should be investigated in the future is the sorting of data within the global detector data stores, which currently more or less reflects the order in which the data was read from file, unless specifically instructed otherwise. For instance, no dedicated algorithms have been deployed at this time which would allow to lay out some data according to their spatial neighbourhood, thus potentially providing better memory locality when dealing with detector components in a given geometric region.



(a) Modules placed along y-direction (b) Modules placed along a helix

Figure 6.27.: Telescope test detectors with different module placement.

6.5. Detector and Navigation Validation

Bringing the components described in the previous sections together, the DETRAY validation tools allow to test the geometry, material budget and navigation of a detector read in from `json` files on host and device. The validation tools are compiled into stand-alone executables as part of the test library, but are comprised of distinct functions that can also be re-used in other contexts, like the DETRAY Continuous Integration or client projects such as TRACCC or, in the future, ACTS. In the following, the validation tools will be introduced using the DETRAY toy detector as an example and will subsequently be applied to the ACTS Open Data Detector (ODD) [166], as well as the ATLAS Inner Tracker (ITk) [128, 129], which were exported from ACTS into the DETRAY `json` format.

6.5.1. Testbed Detector Definitions

In order to be used in unit and integration tests or for rapid prototyping development, DETRAY defines a number of detector geometries that are constructed using the builder and factory classes for the detector directly. These detectors are hence independent from any file-based IO and are themselves subject to unit- and integration tests, ensuring their correctness. In the scope of this thesis, two testbed detector were developed, the toy detector, which is the pixel section of the ACTS Generic Detector [187], and the telescope detector, which provides an array of surfaces. The testbed detectors can be generated using simple function calls and a specific configuration, as detailed in the following. All of them can be written to `json` files using the `detray::detector_writer` and thus exported to other projects.

Telescope Detector The telescope detector allows to define a very simple geometry for a low complexity test setup, which defines a single volume and no surface grids. The sensitive surfaces are arranged along a *pilot track*, which can be a ray or a helix. This way, the telescope can be easily oriented with different global coordinate axes and is also capable of accommodating bent tracks in a strong magnetic field or with low momenta. Two example configurations are shown in Figure 6.27.

The sensitive surfaces can be constructed with different mask shapes by a dedicated `detray::telescope_generator` surface factory, that will place a number of surfaces of the same shape in a geometric array within a volume along the pilot track. The distances between the surfaces are either automatically determined (evenly distributed) or can be given explicitly.

The portals are placed onto an axis aligned bounding box and can therefore induce large stepping distances, if the pilot trajectory is not well aligned with the global coordinate axes. The bounding box is automatically adjusted to the dimensions of any collection of sensitive surfaces contained in the volume, by first calculating the axis aligned bounding box around each surface and then constructing a global bounding box around the individual boxes. This means that the portal box is not a minimum bounding box by construction and it can be given an additional minimal envelope around the surfaces as part of the configuration. The construction of the portals using bounding boxes is done by the `detray::cuboid_portal_generator`, which therefore has to be added to a volume builder after the sensitive surfaces have been constructed. The following example demonstrates the construction of a telescope detector with rectangular sensitive surfaces along a helical pilot track and automatic surface placement:

```cpp
#include "detray/detectors/build_telescope_detector.hpp"

// host memory resource
vecmem::host_memory_resource host_mr;

// Mask with a rectangular shape (20x20 mm) linking to volume 0
detray::mask<detray::rectangle2D, algebra_t> rectangle{
    0u, 20.f * detray::unit<scalar_t>::mm,
    20.f * detray::unit<scalar_t>::mm};

// Pilot track in x-direction, starting from the origin
detray::free_track_parameters<algebra_t> x_track{
    {0.f, 0.f, 0.f}, 0.f, {1.f, 0.f, 0.f}, -1.f};

// Helix in a constant B-field of 2T in z-direction
dvector3D<algebra_t> B_z{0.f, 0.f, 2.f * detray::unit<scalar_t>::T};
detray::detail::helix<algebra_t> h(x_track, B_z);

detray::tel_det_config tel_cfg{rectangle};
// Automatically place 15 surfaces, regularly spaced
// along a 2000mm path according to the helix 'h'
tel_cfg.pilot_track(h).n_surfaces(15)
       .length(2000.f * detray::unit<scalar_t>::mm);

// Build the telescope detector
const auto [tel_det, names] =
    detray::build_telescope_detector<algebra_t>(host_mr, tel_cfg);
```

Toy Detector This detector was the main testbed used in DETRAY for the initial development of the geometry description that was presented in Section 6.1. It is designed as an hermetic cylindrical detector as can be found in many collider experiments. The code used to generate the sensitive modules in the barrel and endcap layers was taken from the pixel section of the ACTS Generic Detector, with the exception that the rings that make up the endcap layers contain trapezoid surfaces instead of rectangles. The number of layers in the geometry can be configured, which is very useful during debugging and testing.

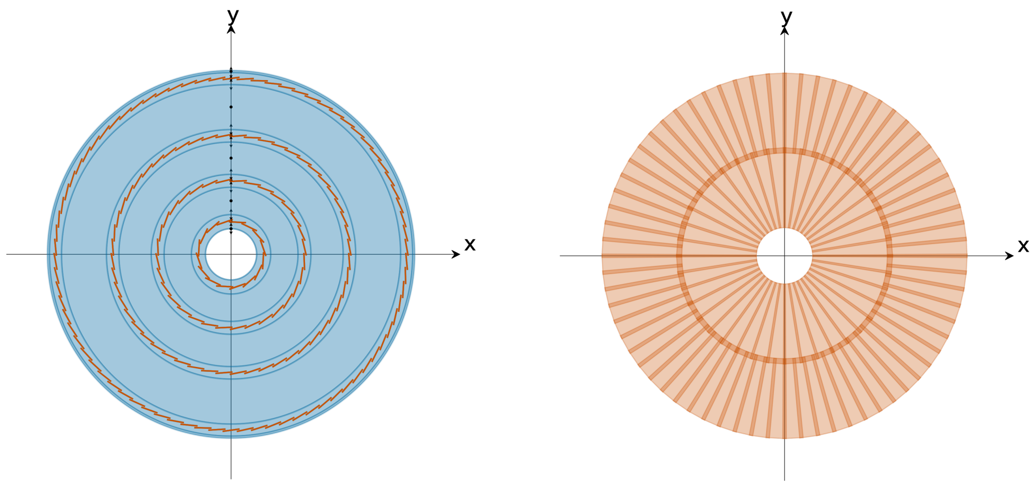
The minimal setup just comprises the beampipe material surface and the innermost volume that contains it. In addition to the beampipe volume, up to four barrel layers can be added to the detector, each of which contains a generic silicon tracker layer of rectangular shaped sensitive surfaces that represent pixel detector sensors. The number of sensitive surfaces scales with the radius of the barrel layer and concentric cylinder surface grids are added to each layer with a custom number of bins.

If the complete barrel section is built, and the detector thus reaches its nominal radius, between zero and seven endcap layers can be added on either side. Each endcap layer is conceptually identical, just placed at different positions in the positive and negative half-spaces along the z-axis, and contains two rings of trapezoidal sensitive surfaces which are registered in a disc shaped surface grid. For both the barrel and endcap sections, there exist dedicated surface generators that can be added to any volume builder in order to generate the sensitive surfaces for a layer with specific parameters, such as the sensor size and layer radius (called `detray::barrel_generator` and `detray::endcap_generator`). The barrel section, as well as the layout of an endcap layer, are shown in Figure 6.28.

The portals around the cylindrical volumes that contain the barrel and endcap layers are generated automatically in a similar fashion as the portal box around the telescope detector volume is constructed. The radii of the inner and outer cylinder portals are estimated by first computing a mean radius from the centre positions of all surfaces in the layer. Then the distance to the maximal half length of the global bounding box in either x or y is calculated and subtracted from the mean radius to form the inner radius of the volume, while the maximal half length is taken as the outer radius.

In between the volumes that contain layers of sensitive surfaces, empty gap volumes are placed. Since the exact extent of the layer volumes is only known after their construction, the gap volumes can only be built after the two respective neighbouring layer volumes have finished their portal generation. The navigation links of the portals that interconnect all of these volumes are currently calculated explicitly and handed to the `detray::cylinder_portal_generator` as additional arguments.

The initial homogeneous material description was not developed as part of this thesis. It puts a silicon material slab of a predefined thickness onto each sensitive surface and a slab of tungsten material onto the beampipe surface. Since the detectors exported from ACTS will, however, hold their material in material maps on the portal surfaces, a material maps generation has been implemented and tested for the toy detector. For this, a `detray::material_maps_generator` was added, which can decorate a class that inherits from the `detray::surface_factory_interface`. It will then generate a material



(a) xy-view of the barrel section (b) Sensitive surfaces of an endcap layer

Figure 6.28.: The *toy detector* allows to build a silicon tracker geometry similar to the pixel section of the ACTS Generic Detector directly in DETRAY. The portals are shown in blue, the sensitive surfaces in orange.

map for any surface that it finds has been generated or otherwise built as part of its underlying surface factory instance and adds it to the volume builder of the volume under construction. This of course, mandates that the volume builder has been decorated with a material maps builder beforehand.

The material maps are generated according to a *profile function*, two of which have been implemented so far: a function that increases the material thickness linearly along the bins on the r-axis of a disc shaped map and one that increases the material thickness quadratically along the bins of the z-axis of a cylinder map. The binning for the material grids needs to be given as part of the toy detector configuration, as is the case for the type of material that should be filled into the bins. The rest of the material grids is automatically fitted onto the surfaces of the underlying surface factory by virtue of the `detray::grid_factory`, as was explained in Section 6.4. Example material maps of the toy detector are shown in Figures 6.11b and 6.11a.

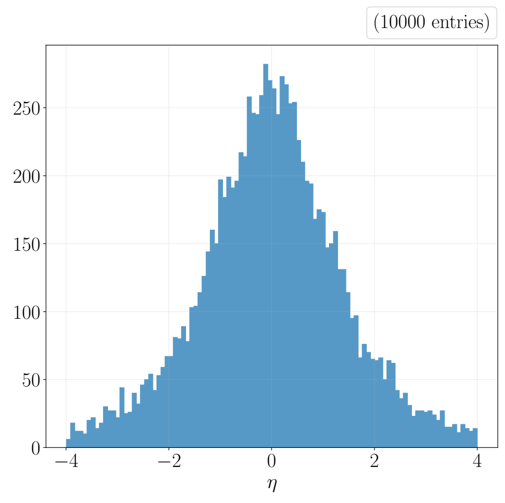
The toy detector is thus a complete demonstrator of the features of a DETRAY detector, albeit smaller than, for example, the ODD and with less realistic detail in its passive surfaces and hence material description. The toy detector and all of its internal linking has been explicitly unit tested in its configuration with three endcap layers. It is also registered for integration tests that run the geometry, navigation and material validation which will be presented in the following sections. The code example below shows the construction of a toy detector with four barrel and seven endcap layers, as well as material maps on all of its portals and their respective binning:

```
#include "detray/test/common/build_toy_detector.hpp"

// host memory resource
vecmem::host_memory_resource host_mr;

// Create toy detector configuration
detray::toy_det_config<scalar_t> toy_cfg{};
// Number of barrel layers (4), endcap layers on either side (7)
toy_cfg.n_brl_layers(4u).n_edc_layers(7u);
// Material map configuration, including binning in (phi, z) and (r, phi)
toy_cfg.use_material_maps(true).cyl_map_bins(20, 20).disc_map_bins(5, 20);

// Build the toy detector
const auto [toy_det, names] =
    detray::build_toy_detector<algebra_t>(host_mr, toy_cfg);
```

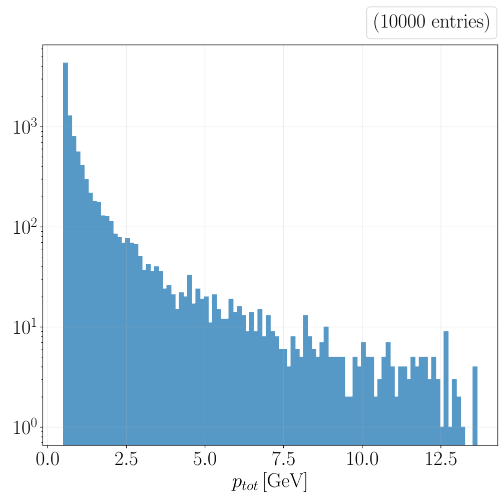
6.5.2. Track State Generators

In order to be able to run a meaningful validation, a number of test tracks have to be generated first, which can then be run through the navigation. Similar to the ACTS *particle gun*, which generates the track parameters for single test particles that can be propagated through a tracking geometry, DETRAY provides track generators based on `detray:ranges`. These can therefore simply be run in a ranged for loop to generate test tracks, for example:

```
#include "detray/simulation/event_generator/random_track_generator.hpp"

using trk_generator_t =
    detray::random_track_generator<detray::free_track_parameters<algebra_t>>;

trk_generator_t::configuration trk_gen_cfg{};
trk_gen_cfg.n_tracks(10000u)
           .eta_range(-4.f, 4.f)
           .randomize_charge(true)
           .p_T(0.5f * unit<scalar_t>::GeV);

// Vector for the constant magnetic field
vector3_t B{0.f, 0.f, 2.f * unit<scalar_t>::T};

// Generate the track state while iterating
for (const auto& track_param : trk_generator_t{trk_gen_cfg}) {
    detray::detail::helix h(track_param, B);

    std::cout << h << std::endl;
}
```



(a)



(b)

Figure 6.29.: Distribution of some example track properties for test helices with $p_T = 0.5\,\text{GeV}$ generated by the `detray::random_track_generator` for the detector scan and navigation validation.

This will generate 10 000 track parameter instances in a pseudorapidity range of $|\eta| \leq 4$ with directions generated by drawing the φ and θ values from a uniform random number generator. In conjunction with a constant magnetic field vector of 2 T these can then be used to define helical test trajectories, as described in Section 3.2. The track generation can also produce dedicated straight-line rays instead, in case of vanishing magnetic fields.

Two different modes of track generators exist, one that generates tracks in uniform steps through φ and θ (`detray::uniform_track_generator`) and one that uses a pseudorandom number generator to determine the direction (`detray::random_track_generator`). The uniform track generator is helpful in scans like the one that is performed for the material validation, where each direction bin should be sampled only once, while the random track generator is used to produce randomly distributed test cases for the navigation validation. The distribution of track properties, such as direction or transverse momentum, is shown for the sample that was generated for the navigation validation on the toy detector in Figure 6.29. Since the random number seed was fixed, all of the following validations for the different detector geometries were run with identical test tracks. The rest of the track properties, such as the charge distribution, as well as the samples for the other momenta are shown in the Appendices in Figures A.1 to A.7.

6.5.3. Geometry and Navigation Validation

In this section, the tools will be described with which a detector can be validated and tested for correct navigation and material description. This generally works in multiple steps:

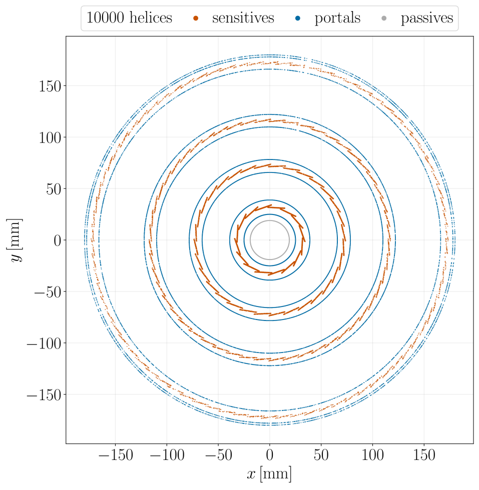
- **Consistency check:** Run a number of logical checks on the detector and its components to ensure internal consistency
- **Ray/helix scan:** Generate parametrised test trajectories (ray or helix) and run checks on the trace of detector surfaces they intersected
- **Navigation validation:** Run the navigator with the same initial direction as the test trajectory from the scan and compare traces.
- **Material validation:** Generate equally spaced rays through the detector and accumulate material from intersection points. Later, compare against a material trace collected during navigation.

The first test after the construction of a detector, be it after explicit construction or after reading in data files, is to run the consistency checks. These comprise many different tests on general properties of the detector data structure, ranging from simple checks that there are any volumes in the detector to checking that every surface that is part of the surface range of a given volume is properly registered in one of its acceleration data structures and can be retrieved correctly from the detector by its identifier. Furthermore, the self-consistency checks on every single volume and surface are called during this test, checking that the index based links of the given descriptors have valid values. The consistency checks of the masks, which are called by their respective surfaces, additionally make sure that the mask boundaries conform with shape specific restrictions, for example, that all radii have positive values.

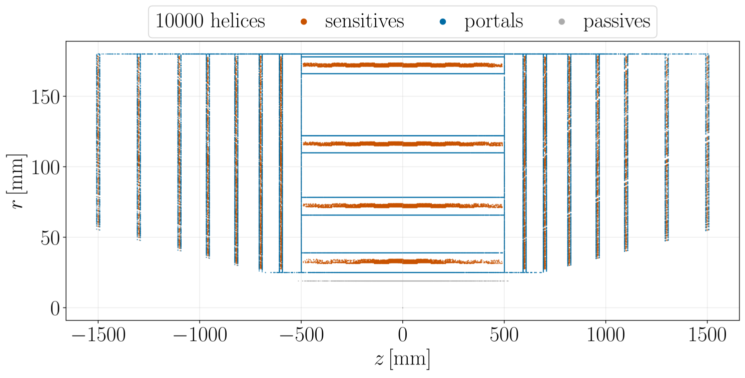
Ray and Helix Scans

Once the consistency check is successful, it can be tested whether the detector is suitable for navigation by generating test trajectories and intersecting them with every surface in the detector. Those that are intersected successfully will be recorded as part of an *intersection trace* per test trajectory, which can subsequently be checked for consistency. The intersection trace contains the intersection data at every encountered surface, as well as a snapshot of the track parameters at this position. An empty record is added at the beginning of every trace which is placed at the origin of the track, capturing its initial track parameters.

Sorted by distance from the trajectory origin, the intersections must follow a sequence of an arbitrary number of sensitive and passive surfaces in between the portal surfaces that lead to neighbouring volumes. These portals have to come in pairs at the same distance, one for either of the two neighbouring volumes, and have to carry the correct navigation links. No surface intersection must be found outside the scope of the entry and exit portal of its respective volume, since this would mean that the surfaces would not be correctly contained in its mother-volume. Finally, it is checked that the last position in the trace contains a portal that allows to exits the detector. For detectors exported from ACTS, the portal masks are not always clipped to the extent of the volume, leading to multiple overlapping portals on the neighbouring volumes. Resulting duplicate portal intersections need to be removed before the intersection trace consistency check is run.



(a) xy-view (z-range of $\pm 50\,\mathrm{mm}$)



(b) rz-view

Figure 6.30.: Helix scan of a toy detector geometry using the track generation presented in Section 6.5.2. It shows the scatter plot of the intersection points of the helices with the detector surfaces, colored by the surface type.

This test can be performed for both straight-line rays, as well as helices [226] in a default $2\,\mathrm{T}$ constant magnetic field in global z direction. The resulting intersection traces, together with the corresponding track parameters, can be written to text files to be used as truth reference for following tests. Figure 6.30 shows the intersection positions of a helix scan with $p_T = 0.5\,\mathrm{GeV}$ of the toy detector.

For the straight-line rays, the intersections can be calculated analytically, the same way as it is done for the navigation rays during the track parameter propagation. The same is not true for the helical tracks, however, as there is no general analytical solution for intersections with arbitrarily aligned surfaces. Instead, the intersection has to be calculated numerically, for example, using a root finding algorithm, as explained in Ref. [134]. For instance, any position $\mathbf{r}(s)$ and direction $\mathbf{T}(s) = \mathrm{d}\mathbf{r}/\mathrm{d}s$ along a helix that fulfils the following equations [227]:

$$\begin{aligned} f(s) &= ((\mathbf{r}(s) - \mathbf{c}) \times \mathbf{z})^2 - r^2 = 0\,, \\ f'(s) &= 2\,((\mathbf{r}(s) - \mathbf{c}) \times \mathbf{z}) \cdot (\mathbf{T}(s) \times \mathbf{z})\,, \end{aligned} \tag{6.3}$$

lies on a cylinder surface with radius r, local z-axis $\mathbf{z}$ and centre $\mathbf{c}$. The goal is then, to find a parameter s so that the helix position becomes compliant with the cylinder condition. Similar equations are formulated for any other surface geometry that might be encountered as part of the detector. The root finding procedure for these equations that is applied in the helix scan follows algorithms from Ref. [228], in particular, `zbrac` and `rtsafe`. As a first step, the approximate root positions are estimated by a straight-line intersection of the surface with a ray that has the same origin as the helix and the direction of the helix at half the distance between its origin and the surface centroid or radius. In most cases, this is expected to constitute a more accurate direction estimation for the entire path to the surface than the direction at the helix origin.

Since the derivative of f is known for every surface type, the *Newton-Raphson* algorithm can be used to find the precise position of the root. It approximates the root by intersecting the x-axis with the tangent at the current position and subsequently using that intersection as the new position:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}\,. \tag{6.4}$$

The quality of the starting position can determine whether or not the algorithm converges to any root and if so, to the correct one. For cylinders, for example, both straight line intersection points need to be tested. Other behaviour of the function f and its derivatives can throw the Newton algorithm also off course, such as a vanishing first derivative or symmetries in the function that can cause the iteration to oscillate between a few iteration positions. In order to safeguard against failures in the Newton method, a second algorithm is applied, which is guaranteed to converge if a *bracket* can be found around the root. For this, a small interval is put around the initial estimate and the sign of $f(s)$ is checked at the boundaries of the interval. If the sign of the two function values is different, then the interval constitutes a bracket around a root that is contained within the interval bounds. If on the other hand the sign is equal, the interval is widened iteratively until a root can be bracketed or the maximum number of interval expansions is exceeded. It remains to be noted, that a root is only guaranteed to lie within the bracket if the function $f(s)$ is continuous over the interval. Furthermore, a sign change may not be detected if, for example, an even number of roots is contained in the interval or if the root is at the same time an extremum of f.

If a root was successfully confined in a bracket, it can be found iteratively by a *Bisection* prescription. This entails shrinking the bracket at every step by taking the middle point and evaluating the function f at this point. Depending on the sign, the corresponding boundary of the bracket is set to the new point. If therefore a step by the Newton algorithm would carry the iteration out of the bracket or if the Newton step would actually converge slower than the Bisection step (motivated in B.1), the Bisection step is taken instead. If a bracket lies completely outside a given maximal path length at any point, the iteration is aborted and the intersection is reported as having missed the surface, since it is expected to be discovered outside the detector space. If the initial bracketing fails, the Newton algorithm is run on its own, without any bisection steps.

After an intersection point has been found, the surface mask is evaluated in order to find out if the detector surface was actually hit by the helix. Since the iteration terminates with a convergence tolerance of $1\,\mu$m in the step size, it comes with a certain approximation error, which is handled by adjusting the mask tolerance. For this, the final step size $\mathrm{d}x$ of the iteration before it terminates is projected onto the surface using the incidence angle of the helix. This is to make sure that the decision on whether or not the surface was hit is taken as accurately as possible, so that it can serve as a ground truth for the comparison against the navigation trace.

Comparison with the Navigator

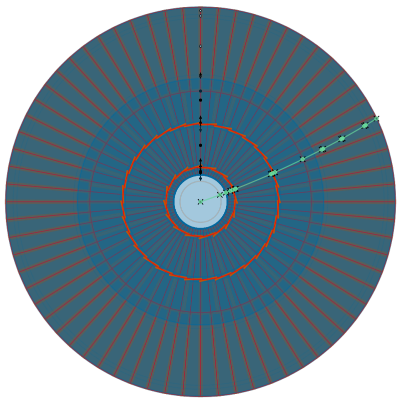
After the ray or helix scan finishes, the truth intersection traces and corresponding track positions can be read from a number of files for a subsequent test, which takes the initial track parameters from the first trace entry and lets the navigation follow the track. All surfaces that are encountered by the track are recorded using an *object tracer* navigation inspector, which is capable of recording internal navigation states, such as the exit portal of a volume. This would otherwise not be captured if the tracer would run as part of the actor chain.

This test can be run for rays, using the straight line stepper, as well as for helices, using the Runge-Kutta stepper in a $2\,\mathrm{T}$ constant magnetic field. Furthermore, both a host and a CUDA implementation of this test exist, so that the GPU-based navigation can be validated against the CPU version. For the device test, the number of truth intersections in the trace from the ray or helix scan can be taken as a lower bound for the device-side memory allocation that needs to be made for the object tracer.

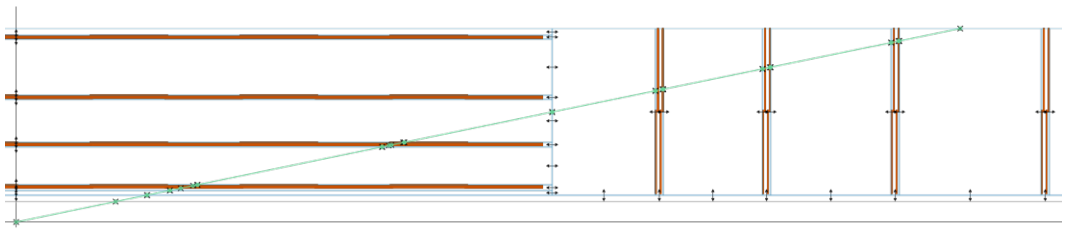
After the navigation finishes and the object tracer has recorded all intersections for the track, the navigation trace and the truth trace from the scan are compared to make sure they contain identical surface sequences[1]. Dummy records will be inserted into either the navigation or the truth trace, if either is missing intersections compared to the other. This makes following comparisons easier, as the traces will have the same length and missed intersections become apparent in residual calculations as outliers, for example. If more than one surface is missing in sequence in a single trace, a dedicated message is printed as warning that something might be seriously wrong, for instance with the acceleration data structure.

In case of any failure, a detailed error log is printed for the corresponding traces, which contains the first intersection that differs, a printout of the entire trace, as well as a detailed debugging record of the navigation process. The latter will contain in-depth information about the navigation states that were recorded during the propagation of

[1]The order of portals that lie at the same distance may be swapped



(a) xy-view



(b) rz-view

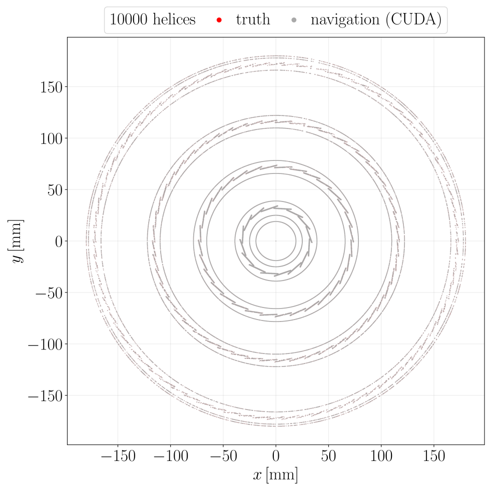
Figure 6.31.: Detector visualisation containing a faulty test trajectory: The black crosses are the truth positions and the green circles correspond to the positions reached by the navigator. The green line is the truth helix.

the test track. It will, for example, show all surface candidates in the navigation cache after any type of update. Since the collection of such a log is costly, and also not possible in general on the device, it is only done during host-side testing.

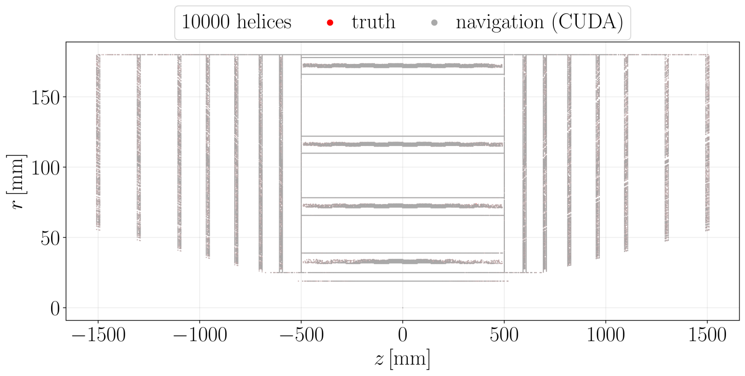
Additionally, any faulty trace is automatically dumped as an `SVG` for visual inspection using the DETRAY detector visualisation tool. The `SVG` will show those parts of the detector geometry that were traversed by the track, as well as an overlay of the recorded and the truth intersections. An example of this is shown in Figure 6.31.

Since the detector is built from input files in the navigation validation tool, this test can be run with and without acceleration structures, by simply leaving out or swapping the grid file, for example. This can be useful for investigating the cause of missing surfaces in the intersection trace of the navigator. It should be noted that the detector scan, which is used as a reference for this test, runs brute force intersection trials which are thus independent from any acceleration structures.

An overlay plot of the track positions found between the host helix scan and the device navigation tracing in the toy detector is shown in Figure 6.32. The resolution of the plot usually does not allow to spot minute differences, but larger discrepancies, such as surfaces that are not found by the acceleration structures, might become visible as



(a) xy-view (z-range of $\pm 50\,\mathrm{mm}$)

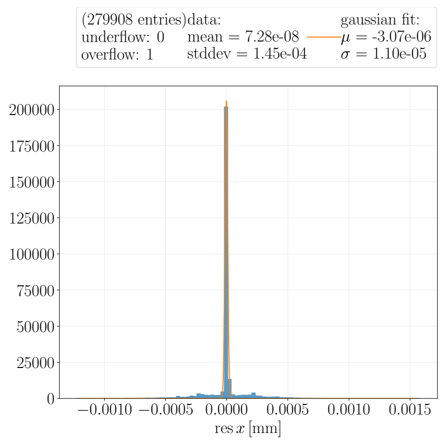


(b) rz-view (entire ϕ-range overlayed)

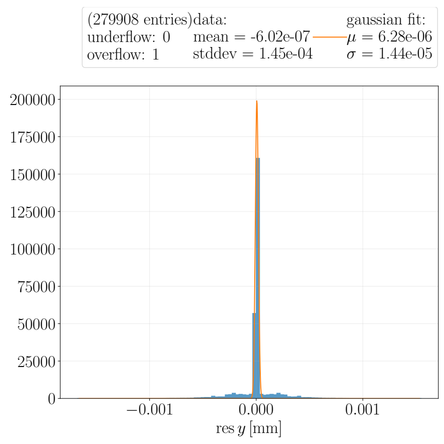
Figure 6.32.: Overlay of track positions found during a helix scan of the toy detector on the host (red), which is a brute force intersection method, and the track position trace recorded during navigation on the device (grey).

red patches where the navigation positions are missing. A similar plot can be made to compare host and device navigation traces directly, as shown in Figure C.2.

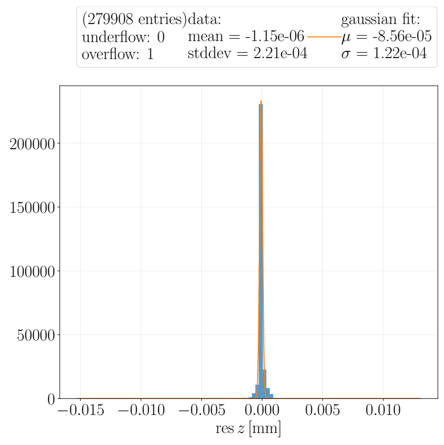
It is also possible to plot the residuals between the intersections found by the detector scan and the navigator, as is shown in Figure 6.33. In general, this residual will also depend on the floating point precision that DETRAY was compiled with. In order to avoid floating point issues as much as possible, the detector scan is always run in double precision and then read in from file for the single precision navigation validation runs. The `csv` file IO can, in principle, also influence the floating point error, but this is expected to be much less severe. Missed or additional surfaces will appear as outliers in the residual plots, as they are calculated against the dummy records that are inserted into the scan and navigation traces during the trace checking.



(a) Residual x



(b) Residual y



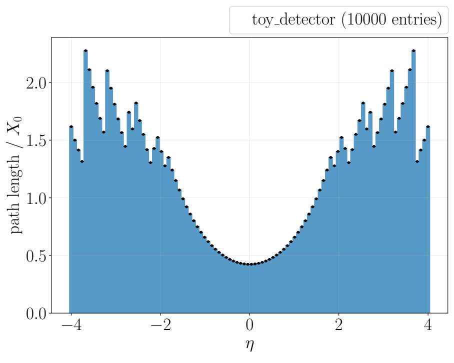
(c) Residual z

Figure 6.33.: Track position residuals between the helix scan and device (CUDA) navigation traces for the toy detector with $p_T = 0.5\,\text{GeV}$.

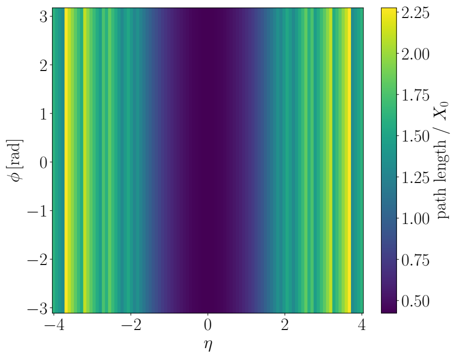
The residuals between host and device track positions have been investigated as well and were found to be small, generally with a standard deviation in the range of $0.1\,\mu m$ to $0.01\,\mu m$ for all tested detectors. The respective plots can be found in Figures C.9, C.10 and C.11 for the toy detector. The GPU used for the tests is an *NVIDIA RTX A5000* together with an *AMD EPYC 7413* 24-Core CPU.

6.5.4. Material Validation

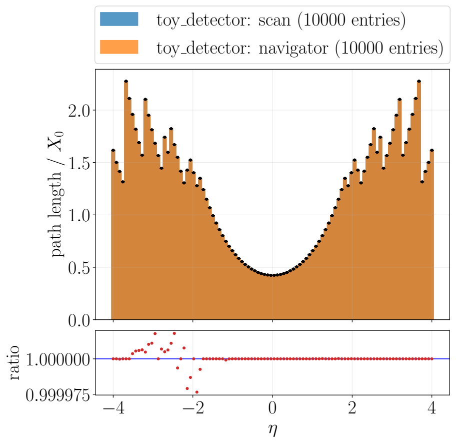
The material description in the detector can be validated in a similar way as the navigation. An initial ray scan will collect the integrated amount of material along the ray that was discovered on the detector surfaces that were intersected successfully. This is a similar concept to the material budget scans done in Geant4 with test particles called



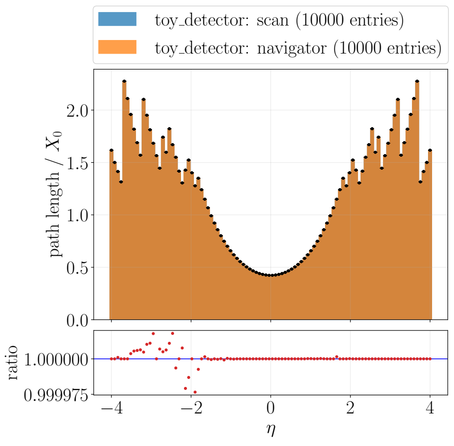
(a) Scan profile



(b) Scan map



(c) Host trace



(d) Device trace

Figure 6.34.: Material scan and material navigation traces of the toy detector with auto-generated material maps and seven endcap layers are practically identical.

Geantinos [164]. The path length of the ray through any material slab is collected both in units of radiation X_0 and interaction length λ. In order to obtain regular binning for the comparison, the `detray::uniform_track_generator` is used for this test. For the toy detector, this is displayed in Figures 6.34a and 6.34b, where the mean material budget as path length in units of radiation length X_0 is shown against η. The η-bin content is the mean taken along the φ-bins of the material scan, which each contain the same material budget in the toy detector material maps, as can be seen in Figure 6.34b. The errors in the material profile plot in Figure 6.34a are the standard deviation of the mean.

In a second step, the navigator is run along the ray with a special *material tracer* as part of the actor chain. The goal here is to record the amount of material in the same way it is reported to the actors, since the material interactor runs as part of the actor chain. The material tracer works the same way on the device as on the host, as it only accumulates the total material budget instead of recording each intersection. This can be done with a fixed number of variables, instead of having to allocate memory for a container of potentially unknown size.

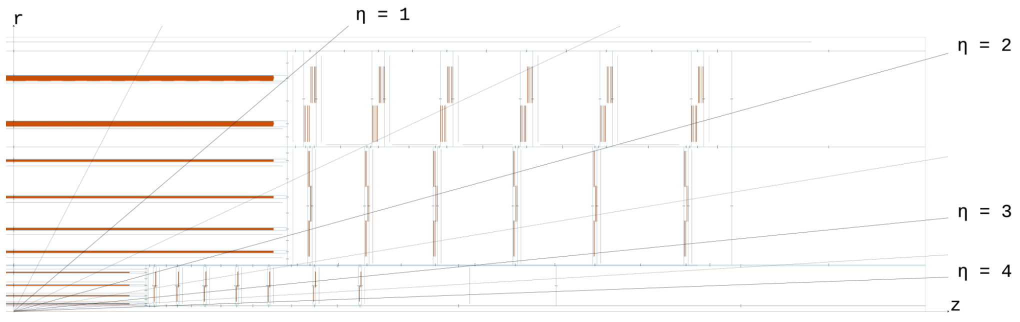
Because the ray material scan would record the material it finds on both the exit and the entry portal, and thus see too much material, a correction has been added where it skips the exit portal the same way the navigator would. With this correction, the material found by the ray scan and the material recorded by the material tracer should match. This is tested automatically per ray during the test and is also run for the DETRAY testbed detectors as part of the Continuous Integration system. It has been found, however, that this is not always the case and that single material rays can differ sometimes substantially in the material they find compared to the navigation. The material recorded during host and device propagation for the toy detector is shown in Figures 6.34c and 6.34d as a function of η, respectively. The error on the ratio plot is obtained from Gaussian error propagation of the profile plots. For the toy detector and the 100×100 rays in ϕ and θ directions that were used in the test in Figure 6.34, no major discrepancies between the material scan and either the material navigation trace on host or device were found. The differences found in some bins are minor and were not reported by the material test as faulty rays, which means that their relative error is below 1%.

However, in the negative endcap region, between $\eta \sim -4$ and $\eta \sim -2$, there seems to be a more systematic fault that hints at a problem with this specific part of the toy detector material maps. It is still a small deviation, but appears consistently in the host and device material validation of the toy detector and needs to be followed up. An additional thing to consider is, that at the moment the material scan is run in single precision, instead of double precision like the detector scan. Especially during the accumulation of the collected material this could lead to floating point errors which might render the truth material budget noisy.

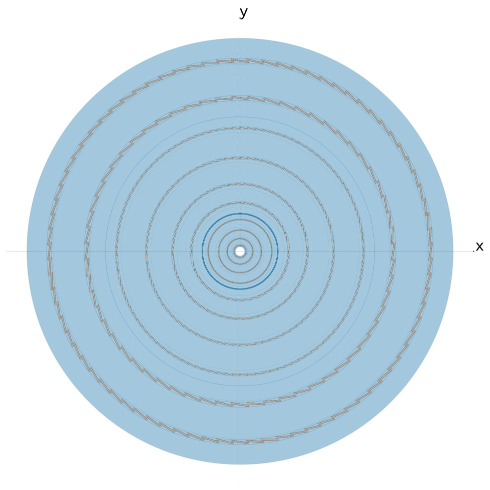
6.5.5. Results

After establishing the tools on the well understood DETRAY toy detector, the validation can also be run on tracking geometries that were exported from ACTS using the new "layerless" tracking geometry description. In particular, these are the *Open Data Detector* (ODD), which is constructed in ACTS from a DD4hep based simulation geometry, and the *ATLAS Inner Tracker* (ITk), which is generated from the same GeoModel [235] description that is used in the ATLAS Athena framework [236]. After the ACTS tracking geometry is constructed, it is converted into DETRAY format and dumped into `json` files that can be read by the DETRAY IO [237]. A visualisation of these geometries in DETRAY is shown in Figures 6.35 and 6.36.

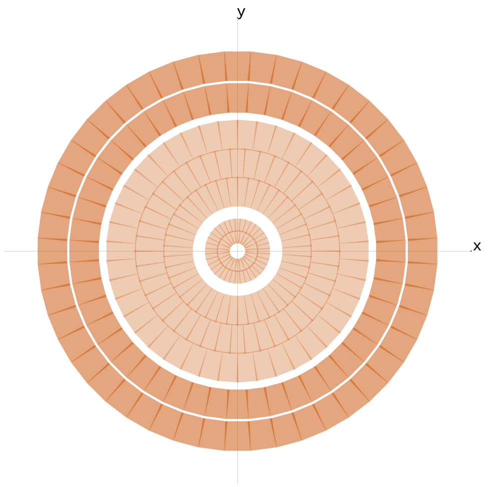
Since the ODD and ITk detectors were exported from ACTS before DETRAY was capable of holding multiple masks per surface, a dedicated python script was developed to perform the surface merging directly on the `json` files. In the end, this will be fixed in the ACTS DETRAY plugin for the final version of the ACTS tracking geometry implementation. The surface grid and material maps files also had to be updated, since the surface



(a) rz-view



(b) xy-view of the barrel section

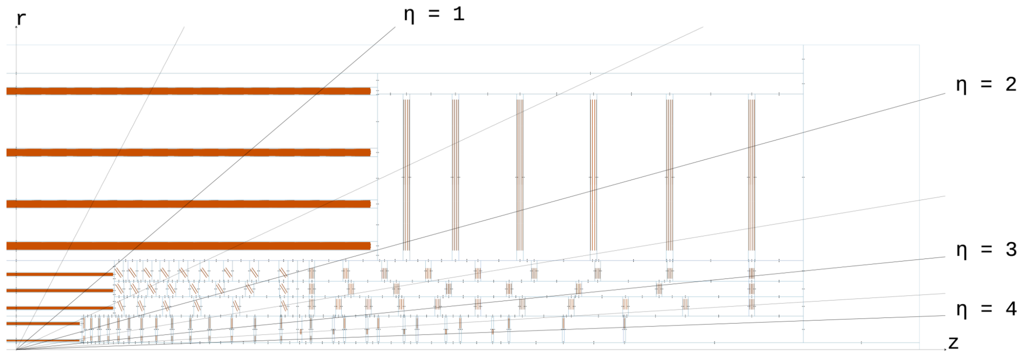


(c) Sensitive surfaces of an endcap layer.

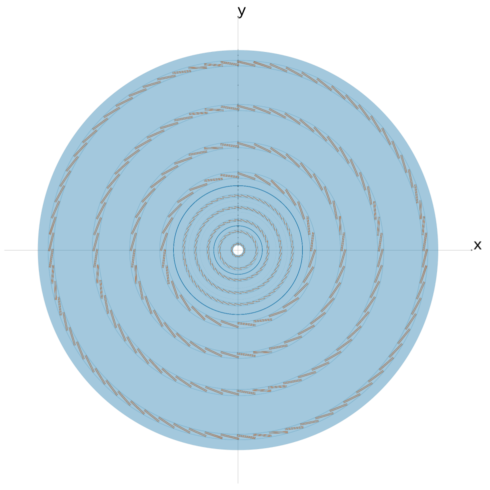
Figure 6.35.: The ODD tracking geometry in DETRAY.

indices change due to the merging. In the case of the ODD, this is straight-forward to do, as the material maps reside on dedicated passive material surfaces. For the ITk, this was more involved, due to overlapping bins of the material grids of neighbouring portal surfaces. The material in these bins was averaged on a best effort basis between the two maps, according to the ACTS material averaging prescription that is also used during the material mapping process [238]. The ITk material maps are therefore an approximation to the material maps held in ACTS, potentially showing discrepancies in the material content of some of the cylinder maps.

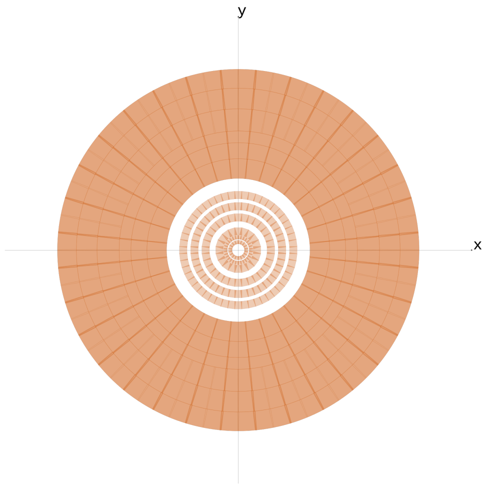
From the `json` geometry, grid and material files that are provided by the ACTS project, the validation tools that were developed as part of this thesis can be run in the workflow that was presented above. The validation workflow is run for 10 000 tracks with $p_T = 0.5\,\mathrm{GeV}$, $p_T = 10\,\mathrm{GeV}$ and $p_T = 100\,\mathrm{GeV}$, respectively. The first momentum that is tested corresponds to the minimum p_T cut that was used for the Run-3 ATLAS tracking performance evaluation [133]. The other momenta are included, because it is



(a) rz-view



(b) xy-view of the barrel section

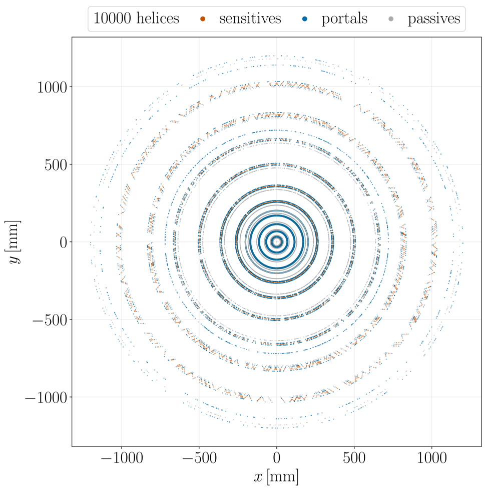


(c) Sensitive surfaces from the pixel inner system and outer endcap, as well as the strip endcap.

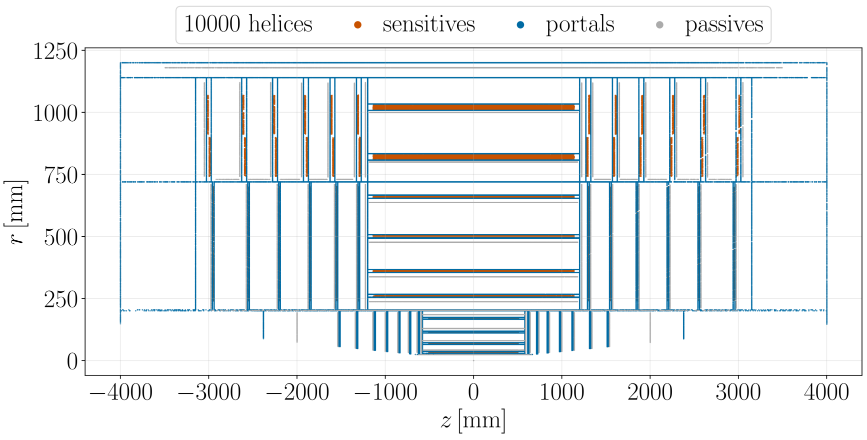
Figure 6.36.: The ITk tracking geometry in DETRAY.

expected that the navigation will be more accurate for higher momentum tracks, as motivated in Section 6.3. Therefore, if many missed surfaces by the navigator are found in the low momentum sample, it is expected that an improvement can be seen for the higher momentum samples.

Also, since the truth trace generation for the helices can actually miss surfaces itself, as discussed before, the number of surfaces that are found by the navigation in addition to the truth trace are counted as well. These are expected to be close to zero if the helix-surface intersection code works correctly and the mask tolerances are adjusted correctly. As the ray scan and straight-line navigation are fairly trivial, and have been found to always pass the tests, the plots included in the following focus on the helix based validation. Example helix scans of the ODD and the ITk from these validation runs are shown in Figures 6.37 and 6.38.



(a) xy-view (z-range of $\pm 50\,\mathrm{mm}$)



(b) rz-view

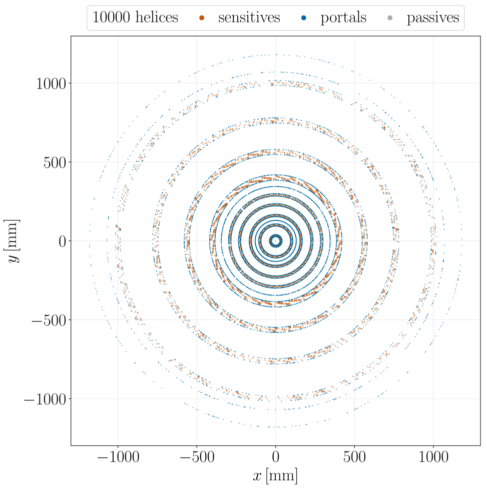
Figure 6.37.: Helix scan of the ODD at $p_T = 0.5\,\mathrm{GeV}$ and $|\eta| \leq 4$. It shows the scatter plot of the intersection points of the helices with the detector surfaces, colored by the surface type.

The number of missed and additional surfaces for the different momenta and detectors are shown in Table 6.1. The truth data from the detector scans is always produced in double precision and subsequently read in from `csv` data files for the single precision test, as already mentioned. The results for the ray scan are listed in Table C.1.

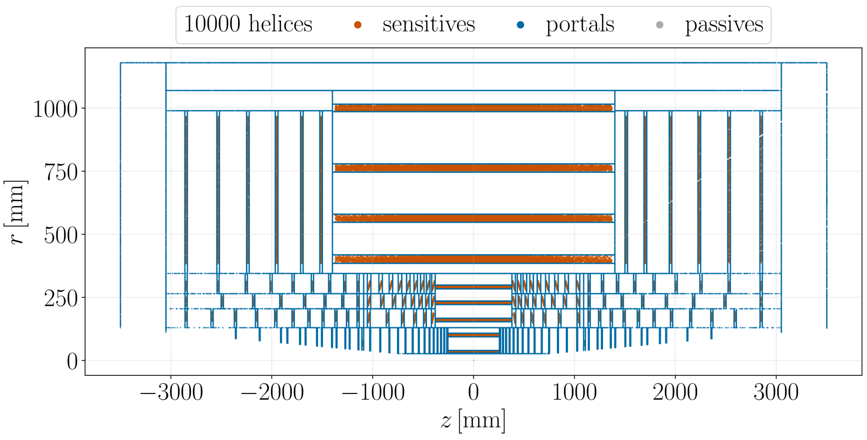
As can be seen, for data samples that were generated with 10 000 rays and helices respectively, the agreement between the detector scan and navigation trace is very good, with hardly any missed surfaces (portals, passives and sensitives counted together), even down to a transverse momentum of $p_T = 0.5\,\mathrm{GeV}$. Only one additional surface was found by the navigation in the ODD test for $p_T = 10\,\mathrm{GeV}$, but for now, additional surfaces are not counted as an error of the navigation. The largest number of missed surfaces

Host					
Detector	p_T [GeV]	**miss.**	**add.**	**total**	$\epsilon_{\text{surfaces}}$
Toy Detector	0.5	1	0	279909	99.9996 ± 0.0006
Toy Detector	10	1	0	279584	99.9996 ± 0.0006
Toy Detector	100	0	0	279653	100.000 ± 0.0004
ODD	0.5	0	0	827286	100.000 ± 0.0002
ODD	10	0	0	820128	100.000 ± 0.0002
ODD	100	0	1	820180	100.000 ± 0.0002
ITk	0.5	1	0	776926	99.9999 ± 0.0002
ITk	10	2	0	771987	99.9997 ± 0.0003
ITk	100	0	0	771923	100.000 ± 0.0002
Device (CUDA)					
Toy Detector	0.5	1	0	279909	99.9996 ± 0.0006
Toy Detector	10	1	0	279584	99.9996 ± 0.0006
Toy Detector	100	0	0	279653	100.000 ± 0.0004
ODD	0.5	0	0	827286	100.000 ± 0.0002
ODD	10	0	0	820128	100.000 ± 0.0002
ODD	100	0	1	820180	100.000 ± 0.0002
ITk	0.5	1	0	776926	99.9999 ± 0.0002
ITk	10	2	0	771987	99.9997 ± 0.0003
ITk	100	0	0	771923	100.000 ± 0.0002

Table 6.1.: Surface finding efficiency of the DETRAY navigator on host and device (CUDA) compared to a helix scan with 10 000 helices. The errors on the efficiency is a statistical error obtained by a Bayesian approach [239–241].



(a) xy-view (z-range of $\pm 50\,\text{mm}$)

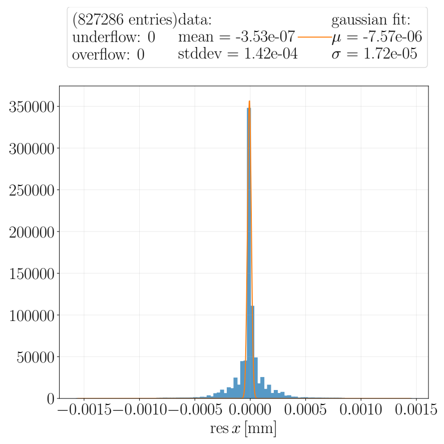


(b) rz-view

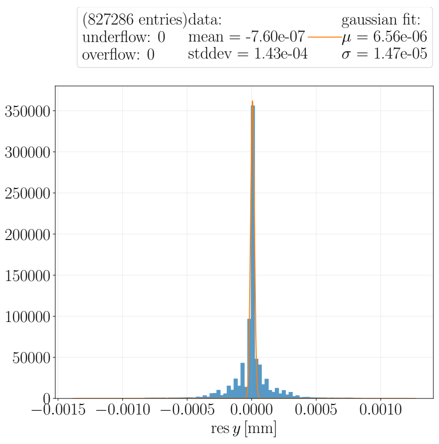
Figure 6.38.: Helix scan of the ITk at $p_T = 0.5\,\text{GeV}$ and $|\eta| \leq 4$. It shows the scatter plot of the intersection points of the helices with the detector surfaces, colored by the surface type.

is found for the ITk geometry at $p_T = 10\,\text{GeV}$, however, for $p_T = 100\,\text{GeV}$ no surfaces were missed for any detector. All of the intersection points of the missed and additional surfaces have been found to lie very close to the respective mask boundary. This can be additionally tested by re-evaluating the mask for the intersection position, but without a mask tolerance. If the intersection is then found to be outside of the mask, it was lying in the tolerance band around the nominal mask boundaries before. This is the case for about half the missed and additional surfaces.

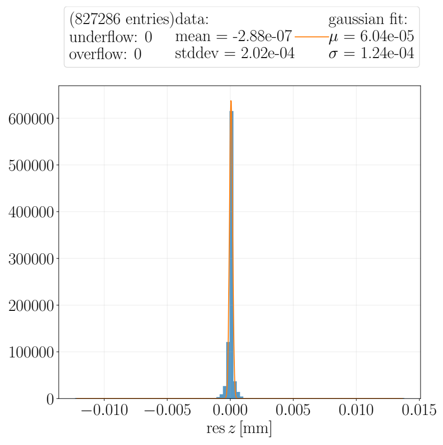
The residuals between the truth and device track positions on the successfully intersected surfaces are small and thus consistent with expectations. In particular, assuming a detector resolution of around $10\,\mu\text{m}$, like that of the ATLAS ID pixel detector which



(a) Residual x



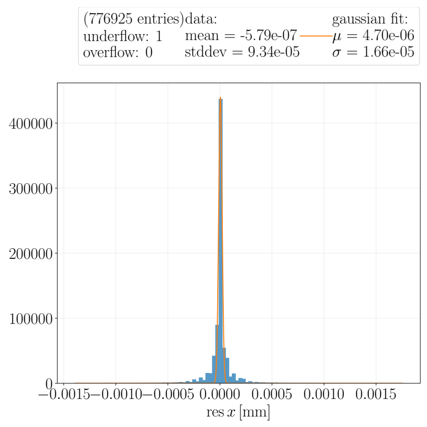
(b) Residual y



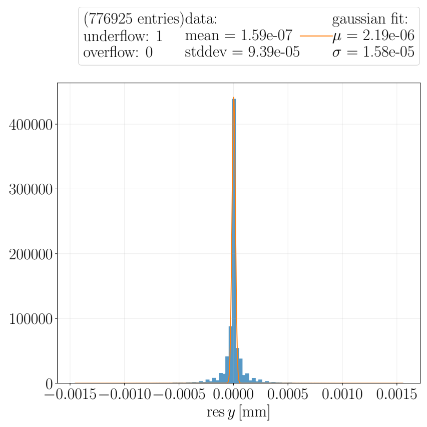
(c) Residual z

Figure 6.39.: Track position residuals between the helix scan and device (CUDA) navigation traces for the ODD with $p_T = 0.5\,\text{GeV}$.

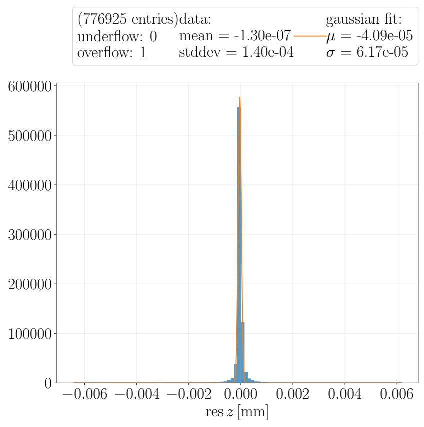
was outlined in Section 2, the standard deviations found in these plots are almost two orders of magnitude smaller. With the caveat, however, that the presented residuals are with regard to global coordinates, while the residuals in local coordinates for the different sub-detector systems would be needed to compare against the detector resolution. Missed surfaces in the trace comparison appear as outliers ($> 1\,\text{mm}$) in the over and underflow bins. In x and y directions, the residuals seem to be larger and less consistent with a Gaussian distribution than in global z-direction, which is parallel to the constant magnetic field. The residuals for the ODD and ITk are shown for $p_T = 0.5\,\text{GeV}$ in Figures 6.39 and 6.40, respectively. The plots for the other momenta can be found for the ODD in Figures C.12 and C.13 and for the ITk in Figures C.17 and C.18. The 2D overlay plots of the track positions that show the comparisons between the truth traces and the



(a) Residual x



(b) Residual y

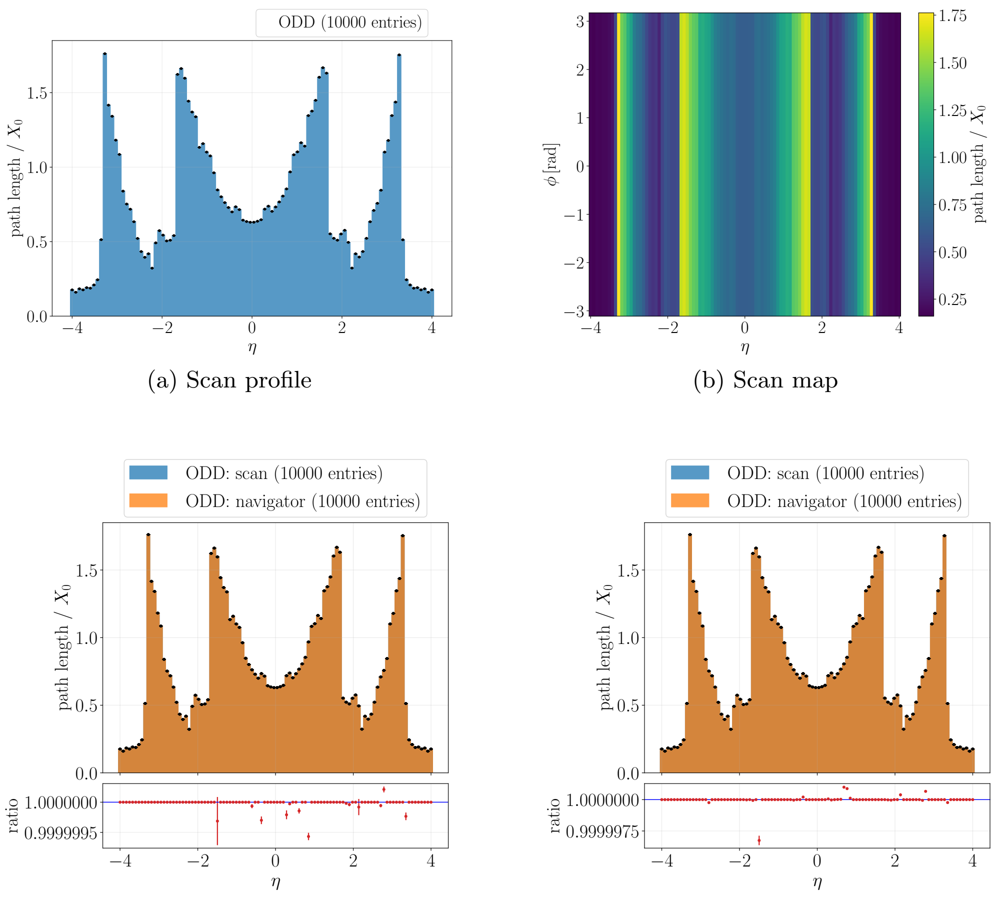


(c) Residual z

Figure 6.40.: Track position residuals between the helix scan and device navigation traces for the ITk with $p_T = 0.5\,\text{GeV}$. The residuals for the other momenta can be found in Figures C.17 and C.18.

device navigation trace, as well as the comparison between host and device navigation traces can be found in Figures C.3 to C.6.

The material validation plots for the ODD and ITk are shown in Figures 6.41 and 6.42. The ODD geometry that was used here has the material mapped onto dedicated disc and cylinder material surfaces in the gap volumes, with a homogeneous material distribution along the φ bins. The expectation here is therefore similar to the material plots of the toy detector that were shown in Figure 6.34. And, indeed, only very few bins show a slight deviation between the material scan and the material navigation trace. The ITk detector, on the other hand, comes with more detailed material maps, which show additional substructure along the φ direction, as can be seen in Figure 6.42b. The material comparison in Figures 6.42c and 6.42d is consistent, however, the ITk was the



(a) Scan profile

(b) Scan map

(c) Host trace

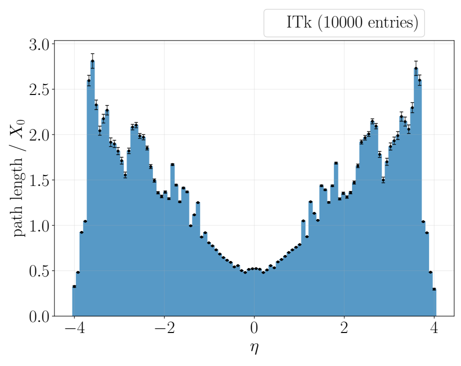
(d) Device trace

Figure 6.41.: Material scan and material navigation traces of the ODD with material maps on special material surfaces.

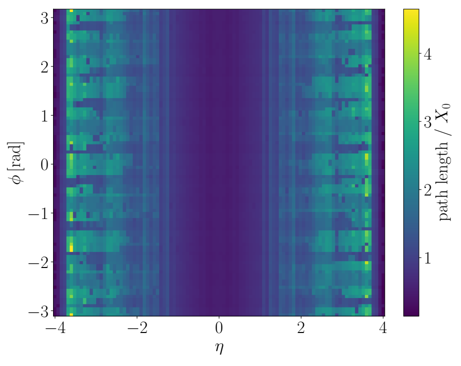
only detector where the material test flagged a number of rays with substantial differences in the accumulated material compared to the device material trace.

6.5.6. Summary and Discussion

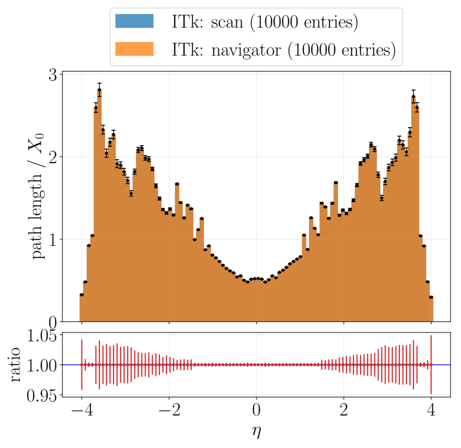
Powerful tools were implemented for the DETRAY detector and navigation validation, which allow precise checks ranging from the verification of general consistency conditions of the detector data structure and index links, to fine grained comparisons of intersection and track positions during host- and device-side navigation runs. The main validation functionality is implemented in a number of free functions that can be re-used outside the current validation tests. This allows future clients of the detray library to repeat the outlined validation steps in their respective experiment's software framework. The validation is also available as stand-alone executables and can be run automatically on



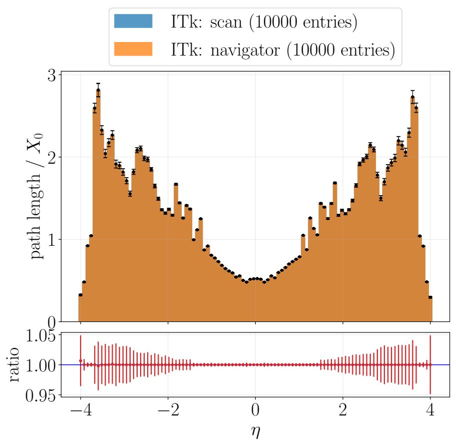
(a) Scan profile



(b) Scan map



(c) Host trace



(d) Device trace

Figure 6.42.: Material scan and material navigation traces of the ITk with material maps.

any detector geometry that was read in from input files. They provide a rich printout of debugging information that can help commissioning a new detector in DETRAY, including a visualisation of any problematic tracks as `SVG`s. For the DETRAY testbed detectors, the full chain of validation functionality is additionally run as part of the Continuous Integration in the source code repository, thus providing strong integration tests.

As was mentioned in Section 6.5.3, no guarantee for convergence of the helix-surface intersections exists at this point. In case the iteration does not converge, a failure of the algorithm does not immediately follow, however. In case no bracket was initially found, it may be the case that the bracketing algorithm failed, as discussed before, but it may also be that there simply is no valid intersection. On the other hand, if a bracket was found, the algorithm is guaranteed to converge. Consequently, the algorithm was written such that it throws an exception and the validation terminates, if the the maximal number of iteration steps is exceeded despite a correct bracket.

If a root of an intersection function such as equation 6.3 was found, it does not have to be the correct one, however. Therefore, the intersection point that was found might wrongfully be assumed to lie outside the surface mask and might not be included in the intersection trace. The portals are the only surfaces where this particular problem can be securely identified, since they have to form a consistent sequence of volume switches in the intersection traces and can consequently be checked explicitly to have been intersected successfully. Up to this point, no errors during the helix scans of the three detectors that have been investigated here were observed.

In the future, it might be interesting to investigate whether the helix equation in equation 3.2 can be approximated well enough by a polynomial, which might allow to use different root finding methods or characterise the roots better. Once understood, approximation errors arising from this might be compensated by adjusting the mask tolerances for the navigator in such experiments. Once the integration of DETRAY into ACTS is finished, the DETRAY propagation can additionally be validated against existing track extrapolation methods.

The results of the navigation validation that were obtained so far show very good results, both for host and device, with the navigator being able to recover almost all surfaces that were also found by the detector scans. The results are stable over the three sampled momenta, ranging from $p_T = 0.5\,\text{GeV}$ to $p_T = 100\,\text{GeV}$. The missed surfaces were found close to the mask boundary edges, many of them not lying within the mask if it were not for the application of boundary tolerances. These can therefore likely be attributed to a navigation configuration that is not yet optimal for the specific detector. Up to this point, the three different detectors were all tested with the same configuration, which is listed in Table C.2. However, there may simply not be a single configuration that works well to recover all of these surfaces. If, for example, the minimal mask tolerance is raised in order to capture one of the missed surfaces, the navigator might start to find additional surfaces elsewhere. Another thing to consider is the low track statistic of only 10 000 which has to be expanded in the future, when the truth generation can be made to run more efficiently.

Another effect that might play a role in the discrepancy between the detector scan results and the navigation trace is floating point errors. Since floating point numbers cannot be encoded precisely in a computer, cut-off and rounding errors occur in the floating point number representation and operations. Different algorithms will, in general, come with different behaviour concerning floating point errors, potentially even exacerbating the problem in certain situations (for example, through *catastrophic cancellation* [242]). A careful analysis of the DETRAY algorithms[2] concerning numerical stability will have to follow in the future to make sure results are reliable, especially when running in single precision. As a first measure, the tests in this chapter could be repeated in double precision to see whether some of the discrepancies will disappear.

The number of additional surfaces found by the navigator was very low, hinting that the root finding during the helix scan did not fail in the majority of cases. Due to the unreliability of the truth information, the additional surfaces are currently not counted

[2]in this particular case the root finding and Runge-Kutta algorithms

as an error. However, if they are really found in addition, and not because the helix intersection failed to converge on a result or due to tolerance effects at the sensor edge, they could have an influence on hole searches at later stages of the tracking chain. If the CKF is lead to a wrong surface by the navigator and does not find a compatible measurement there, this could be wrongfully flagged as a hole along the track.

The current navigation validation is run against the detector scan, either with rays or helices, and thus restricted to homogeneous magnetic fields. The setup using helices is already a fairly detailed test using the Runge-Kutta stepper and relying on the navigator to resolve bent tracks correctly. In order to test the propagation in a more realistic setup, for example, using the ATLAS magnetic field map made available through COVFIE, one could perform a first navigation run without the acceleration structures and with a very small step size constraint, as well as the `always_init` navigation policy for the stepper. This will force the navigator to make frequent updates along the track, while refraining from doing any candidate caching, thus maximising navigation accuracy. In a second run, the nominal navigation can be performed and later compared.

The navigation validation that was performed in this section is, however, not fully representative of the navigation that will be run during track finding. Due the comparatively large uncertainties on the current track state during initial track finding, the navigation will likely have to be run with much more loose tolerances and will additionally have to be robust against updates of the track state stemming from the Kalman filter. This will be investigated in the upcoming physics performance study of the full chain demonstrator TRACCC, which will yield results on, for example, the track finding efficiency, and thus also constitute an indirect test of the navigation accuracy of DETRAY in a realistic setup.

In order to make the conversion of the material maps work between ACTS and DETRAY, the possibility to add material maps that are larger than the owning surface had to be added for the tests in this section. This is due to the different portal description in ACTS and DETRAY, where portals are either part of a single large surface that is shared between adjacent volumes (ACTS) or individual surfaces that need to be registered with every volume separately (DETRAY). This has also lead to the situation, where the material maps are currently duplicated in memory and data files, once per DETRAY-portal surface. This will be resolved in the future by adapting the DETRAY material maps IO module.

The material validation has yielded overall quite consistent results, except for the toy detector, where the material maps generated in the negative endcap might be slightly faulty. For the other detectors, only a few tracks show discrepancies between the material ray scan and the material accumulated during propagation, especially on the ITk, where the discrepancies can be larger. The reason for this will have to be investigated in the future.

The material scan uses the ray's intersection position to fetch the material slab from the map, while the material tracer uses the local position of the bound track parameters. A slight discrepancy here could potentially push the material lookup into an adjacent bin. The final evaluation on the quality of the material description will be made once it has been compared to the fully detailed material description using Geant4. For this, the position of the material along the track also needs to be validated, instead of just looking at the accumulated material budget.

6.6. Performance Measurements

One of the main outcomes of the parallelisation project will be the measurement of the tracking throughput, that is, the number of reconstructed events per unit of time. Beyond this, it is instructive to also examine the performance of the respective algorithms within the tracking chain. In the following, the throughput as tracks propagated per unit time, as well as the scaling behaviour of the navigation and track parameter propagation step will be investigated for different detector geometries.

However, the benchmark implementation that is used in this section is mainly designed to provide a tool for developers to facilitate a quick code optimisation cycle and lightweight automated performance monitoring. It is therefore a synthetic benchmark, meaning it does not measure realistic workloads, but rather trajectories that are generated within the benchmark tool itself for this specific purpose. To this end, it uses the same track generators that were presented in the previous section 6.5.2. This allows for a short turnaround time between performance measurement and code adjustment, in particular, when investigating the impact of certain optimisations or testing specific track populations (e.g. varying the momentum or angular distribution) to understand the performance limitations.

6.6.1. Methodology

In order to receive reliable results, it is crucial to prepare the performance measurements accurately. This starts at the definition of the workload that is going to be measured all the way to the implementation of the measuring code itself, since spurious effects can otherwise influence the results.

In the samples used here, all generated tracks are muons of randomised charge, with the transverse momentum taken from a flat distribution between 0.5 GeV and 100 GeV. In order to equalise the workload between the threads, the tracks are sorted by the absolute value of the polar angle of the track direction θ for the throughput measurements [243]. Sorting the threads during data sample preparation allows threads to be assigned to tracks that will propagate through a similar region in the geometry, such that thread divergence can be minimised on the GPU. Thread divergence describes an effect where threads of the same work unit (called a *warp*) encounter different workloads or instructions in the program. This leads to a de-synchronisation of the thread execution and incurs a performance penalty on the GPU. For the scaling measurements on the CPU, no sorting is applied as each thread works on a continuous block of the tracks.

Depending on the benchmark case, for example, if the number of tracks needs to be varied for a given measurement, separate track samples are prepared to keep the measurements independent of each other. For the GPU benchmarks, the track samples are copied to the device before the timing measurement starts, thus ruling out overheads from the data movement. This is based on the assumption that the DETRAY propagation will run in the middle of the tracking chain, where the data already reside on the device, coming from previous and going into subsequent GPU algorithms.

The DETRAY propagation can be set up in different configurations. In particular, the benchmarks can be run with just the navigation and numeric integration in a constant magnetic field as a baseline or including the track parameter error transport that was described in Section 3.2.2. Additionally, the use of navigation acceleration data structures[3], as well as the inclusion of material in the detector, and hence the handling of material effects during propagation, can be steered by passing the respective data files to the benchmark tool. This will trigger the construction of the corresponding detector components, as detailed in Section 6.4, which will automatically prompt DETRAY to make use of these components during propagation.

The propagation configurations that are investigated for the performance measurements, are either the nominal propagation or, alternatively, two different configurations without material and covariance transport. The latter two configurations thus only run the numeric integration in the stepper using a constant magnetic field and the navigation, once with and once without the grid acceleration data structures in place. It is therefore expected, that the propagation without covariance transport, but using the grids, will be the most performant. The propagation without grids will be less efficient in finding candidate surfaces during navigation and the nominal propagation with grids and covariance transport will include extra computations needed to build the Jacobians and updating the covariances, including material effects.

All benchmark executables are compiled for single precision, which has multiple performance benefits. For one, it reduces the size of the data in memory and thus makes it more cache-friendly. This goes both for the input track data, as well as any temporary data that DETRAY will produce itself, like the navigation state data. For similar reasons, this may also lead to improved vector-register usage in the CPU, which boosts data parallelisation within CPU threads. Furthermore, many GPUs provide less dedicated hardware for double precision floating point operations, which can lead to a significant performance impact for double precision floating point arithmetic.

In addition, DETRAY can be compiled against different linear algebra implementations that reside the ALGEBRA-PLUGINS project, which are expected to have an impact on the track propagation performance. For the benchmarks in this section, the hand-written plugin that uses vectors and matrices based on `std::array` containers is used, since it is at the moment the only linear algebra implementation that is also available in the TRACCC project.

In order to populate the cashes, each benchmark case is preceded by a warm-up phase, where 10 % of the maximal track sample size is propagated without including this work in the timing measurement. In case of the GPU benchmarks, including a cold run makes sure that initial CUDA overheads are avoided, such as potentially the *Just In Time* (JIT) compilation of the intermediate PTX to the respective GPU binary. In general, the warmup is meant to exclude turn-on effects in the first benchmark iterations for both

[3]in this case the surface grids

host and device measurements. In between the call to different benchmark types, for example different propagation configurations or throughput vs. scaling measurements, a cool down period of 5 minutes is also observed to free hardware resources again for the next benchmark. This is, however, not done for the separate benchmark cases that are part of a given benchmark type.

In both the CPU and the GPU benchmarks, the current parallelisation strategy is to assign one thread to each track. When running on the the host, multithreading is implemented using openMP [244], which will assign CPU threads to loop iterations, each of which corresponding to the complete propagation of a single track. The assignment and scheduling of the threads can be steered by specific configurations.

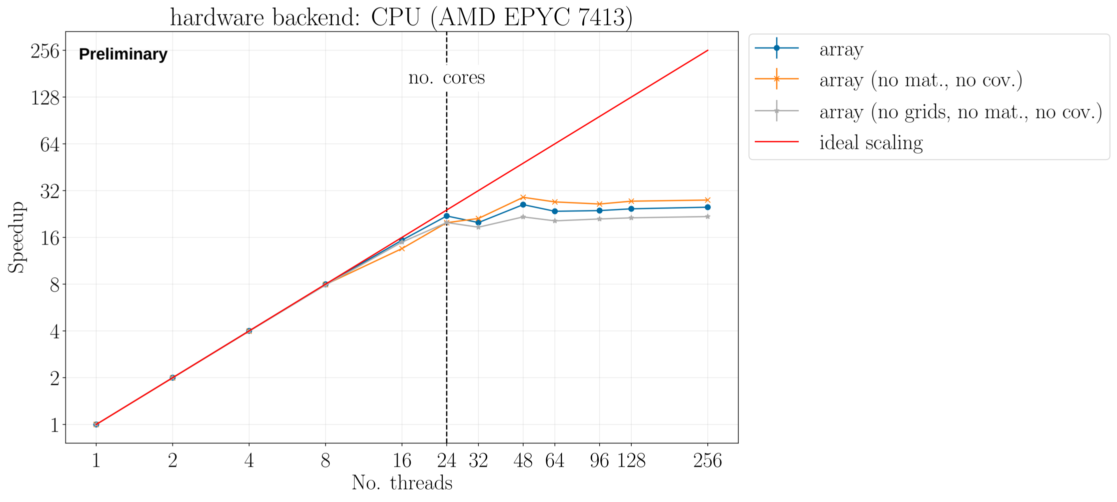
The performance measurements are conducted using the *google benchmark* library [245], which allows to time a particular piece of code and additionally handles common tasks like running multiple repetitions of the benchmark cases and providing the mean and standard deviation of the measured latency. For all of the benchmarks in this section, ten repetitions were run and the mean is reported for each measurement. The repetitions assure that performance fluctuations, for example, due to operating system noise, can be smoothed out. In addition, google benchmark will run the single measurements in multiple iterations, until the measurement is run long enough to be statistically relevant. The number of iterations is in this case determined automatically, based on the time a single measurement iteration takes.

In general, the result of the propagation, i.e. whether it finished successfully or not, is not checked, as this would also influence the measured time of the execution. In order to keep the compiler from removing entire computations during the code optimisation steps in the compilation process, because the results are not saved or used anywhere, google benchmark also provides, for example, the `DoNotOptimize` function. This function ensures that the result of the propagation needs to be written to memory, even if not used in subsequent code. In the GPU code this function is not used at this point.

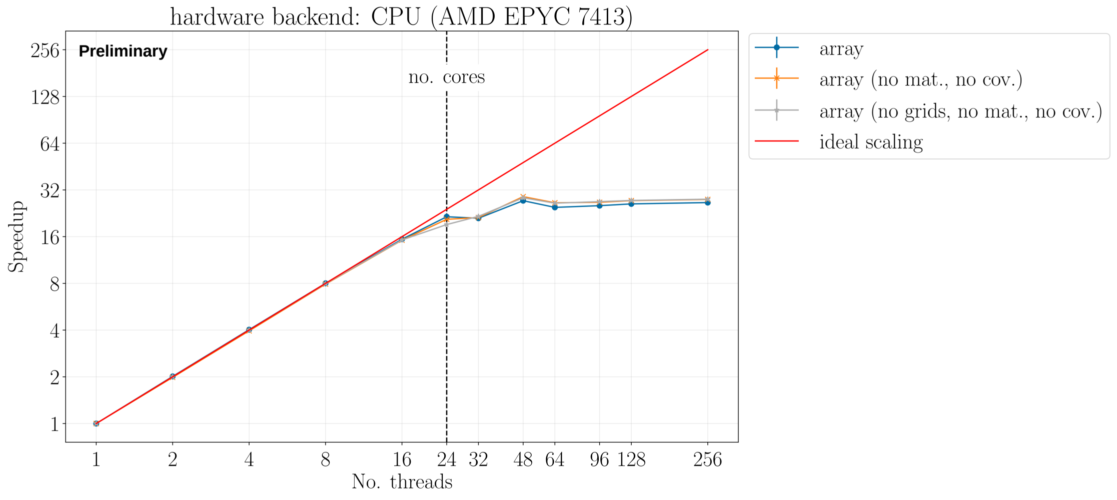
The benchmark code is compiled with `gcc` version 13.2.0 [246] and CUDA version 12.6.20 [247]. Different compilers or compiler versions might result in different timing results, since the optimisations that are performed during compilation may vary. The operating system is *Red Hat Enterprise Linux* version 9.5 [248], with the kernel version `5.14.0-503.40.1.el9_5.x86_64`. All of the performance measurements were run on a node containing an AMD *EPYC 7413* CPU [249], which has 24 physical cores with a clockspeed of 2.65 GHz up to 3.6 GHz and 128 MB L3 cache, with access to 130 GB of RAM. The GPU benchmarks use an NVIDIA *RTX A5000* [250], which provides 8 192 CUDA cores, 27.8 TFLOPS peak single precision performance and 24 GB GDDR6 memory.

6.6.2. Measuring the Scaling Behaviour

In a parallelised system, the scaling behaviour with the degree of parallelisation is an important metric to understand, since it might inform about the type of hardware that the software can be effectively deployed on. If the scaling plateaus with a certain number of compute cores, that is, the use of more cores will not result in a further speed up, then



(a) Strong scaling ODD



(b) Strong scaling ITk

Figure 6.43.: Strong scaling behaviour of different DETRAY propagation configurations on a CPU hardware backend (AMD EPYC 7413). The physical number of cores is indicated by the dashed black line, ideal scaling is represented by the red line.

using compute nodes with a greater number of processors will not lead to a performance improvement. In a *strong scaling* experiment, a fixed problem size is solved using an increasing number of compute cores, often measured against the number of CPU threads. Perfect scaling is achieved, if the latency of the algorithm is halved every time the number of threads or cores is doubled, leading to a linear dependency. In practice, perfect scaling can be broken for many different reasons, for example, information exchange between threads or memory and caching effects.

In the following, the strong scaling behaviour is only measured on a CPU backend, where the number of threads that can be executed in parallel essentially corresponds to the number of cores of the processor. This is difficult to define correctly on a single GPU, due to effects originating from the scheduling of the thread blocks and warps across the processors, which cannot be controlled easily. On the other hand, track propagation on multiple GPUs is not yet supported in the DETRAY benchmarks and scaling with the number of GPUs consequently cannot be investigated at this point.

If more threads are used than there are physical cores on a CPU, it is possible to still see a performance increase due to the use of logical cores. A logical core exists for every physical core, which represents resources on the physical core that are potentially free to be used by another thread executing on the same core (simultaneous multithreading [251]). For even more threads, a speedup could be observed, if, for example, a given thread stalls and another thread takes over and continues to use the processing core.

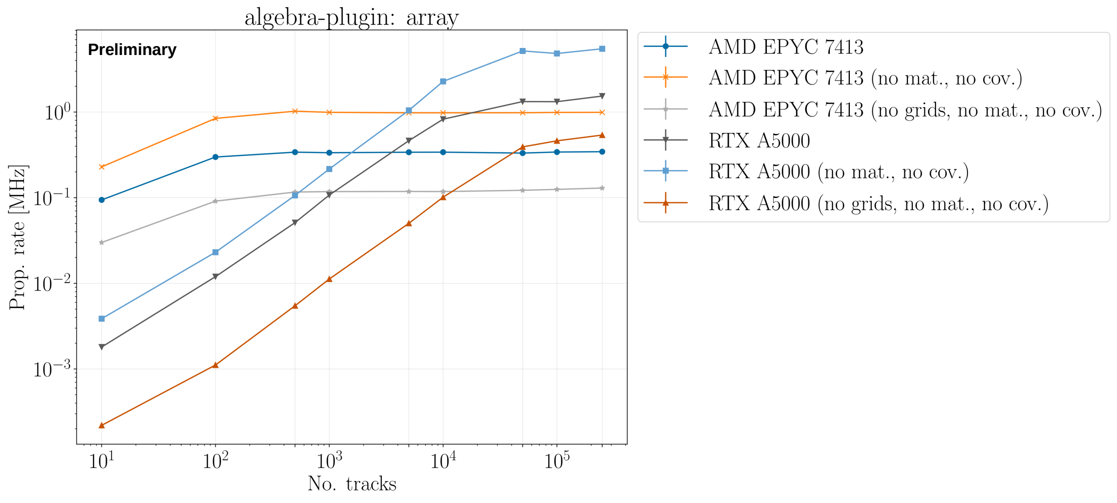
For the scaling measurements, where the track sample of 76 800 tracks is not sorted, each thread is assigned an equally sized *chunk* of tracks. The chunk size for each benchmark case is defined as the size of the track sample divided by the number of threads used, so that each thread only has to work on a single chunk. The configured schedule is *static*, which means that each thread gets the chunks assigned in a *round-robin* fashion, one after the other.

Figure 6.43 shows the strong scaling results for different configurations of the DETRAY propagation on the ODD and ITk against the number of threads. Up to the number of physical cores of the AMD CPU that was used here, the scaling behaviour is very close to the ideal scaling for all benchmarks that were run on both detectors. The next measurement point, for 32 threads, shows a significantly reduced speedup, also for all of the measurements taken. The reason for this is not currently known, it might be a bug in the benchmark implementation or an effect due to the benchmark methodology or the hardware. For 48 threads, which corresponds to the total number of physical and logical cores of this particular CPU, the speedup increases again until it decreases slightly and plateaus when more threads are run than cores are available on the machine. This could be due to increased context switching between multiple threads that are running on the same core, but needs to be investigated further.

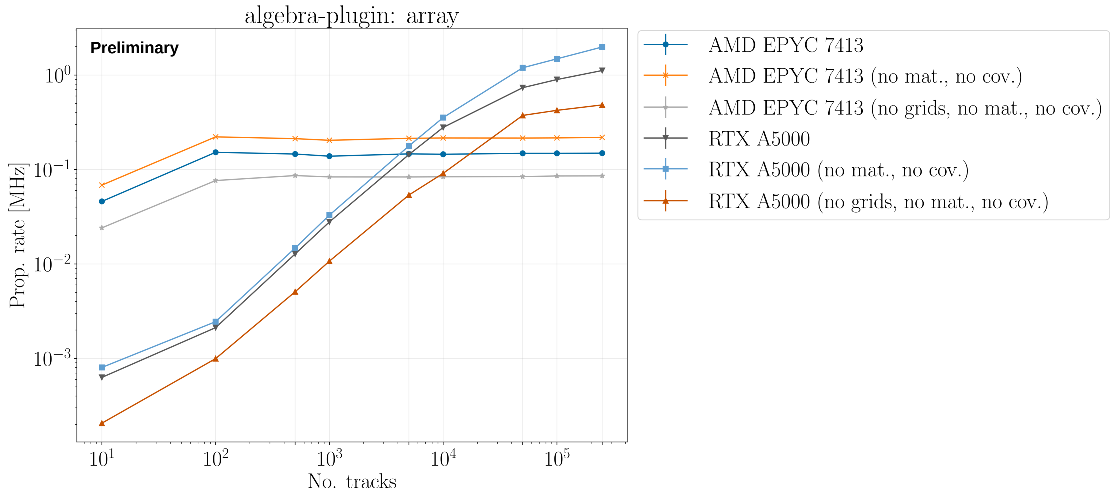
For the ODD, the scaling behaviour between the propagation configurations is slightly different, according to the expectation of the compute performance of the different configurations. This is seemingly not the case for the ITk, possibly because the effect is masked by another inefficiency, however where this behaviour comes from is not yet understood.

6.6.3. Track Propagation Throughput

The throughput measurements have been developed from benchmarks that were added to DETRAY outside the scope of this thesis. In this measurement, the number of processor cores is kept constant, but instead the track sample size is scaled. This allows to investigate how many tracks need to be propagated until the given hardware is saturated and the throughput plateaus, with increasing number of tracks. The throughput is calculated automatically by google benchmark using a custom counter.



(a) Propagation throughput ODD



(b) Propagation throughput ITk

Figure 6.44.: Track throughput measurement of different DETRAY propagation configurations on a CPU hardware backend (AMD EPYC 7413) and GPU backend (RTX A5000) using CUDA.

On the CPU backend, a chunk size per thread of 1 with a *dynamic* scheduling is used. This way, threads can request new tracks to propagate whenever they are ready, instead of having to finish a pre-assigned chunk of threads and then potentially waiting idle for the other threads to finish. However, this can come at the cost of an increased scheduling overhead at runtime. The total number of threads used is determined by the `std::thread::hardware_concurrency` function. On the GPU, the number of thread blocks is scaled with the sample size in order to fill the device efficiently. The number of threads is set to 256 per block, in accordance with *launch bounds* that were given already to the original benchmark kernel [252]. The launch bounds in this case help limit the number of registers used per thread, which could otherwise reduce the number of active warps per processor.

The results of the throughout measurements are shown in Figure 6.44. Some general behaviour can be seen: The CPU propagation saturates much earlier than the GPU-based propagation, staying constant over the different sample sizes. Secondly, the throughput between the propagation configurations differs according to the compute performance expectations that were detailed before. The throughput of just running the stepping and navigation while using the grids is the highest, followed by the nominal propagation. The propagation without grids yields the lowest throughput on both detector geometries on the CPU as well as the GPU backend. Similar to the scaling measurements of the previous section, the throughput differences are less pronounced for the ITk geometry.

Somewhere above a sample size of ~ 1000 tracks, depending on the propagation configuration, it seems that the GPU-based propagation starts to outperform the CPU propagation. However, the exact turning point and speedup are difficult to determine precisely, since it is not clear at this point if the overheads that may be present are fully comparable between the two different executables for the respective hardware backends. Overheads could be introduced due to, for example, the setup of the benchmarks and the benchmarking itself. Furthermore, different compiler optimisations, as discussed below, could result in artificial differences. This will have to be investigated by detailed performance profiling and analysis in the future.

6.6.4. Discussion

The benchmark implementation in DETRAY is still in the process of being established and the results are therefore highly preliminary. An number of issues have been identified so far, chief among them that the execution of the different types of benchmarks needs to be profiled in detail in order to determine unintentional overheads that may even differ between the host and device benchmarks. This makes comparisons between the results for the different hardware backends very difficult at this point.

Furthermore, effects that may have been introduced by the way the google benchmark library was included need to be investigated. For instance, it was discovered in previous DETRAY versions that running a subset of the benchmarks by selecting them at runtime does not always yield the same results as compiling the benchmarks executable without the unselected cases. Potentially even the order in which the benchmark cases are called might have an influence on the result. Such interdependencies between the benchmark calls have to be re-evaluated for the current DETRAY version and understood. This could be done by trying to align the propagation loop instructions or adding empty instructions in order to avoid instruction caching effects, to name just one possibility.

DETRAY itself also contributes to problems in the benchmark results, for example, in a single precision build with the native instruction set for the CPU enabled (using the `-march=native` flag), the navigation seems to hang in an infinite loop in rare cases. This is likely due to slightly different numerical results caused by the use of specific instructions, like *fused multiply-add* (`fma`). On the other hand, turning these instructions off, as was done for the measurements here, could result in a reduced compute performance of the CPU code. In the GPU benchmarks, in a second run in which error printouts were inserted, it was observed that in the ODD throughput benchmark case measuring 100 000 tracks with covariance transport enabled, about 10 tracks do not finish the propagation successfully. This is likely, because *fma* instructions are currently enabled in the CUDA compilation and the same issue appears as on the CPU. The exact reason for this problem still needs to be determined.

Last but not least, there are also performance relevant issues with the DETRAY core implementation itself. The identification and optimisation of these has begun within the community, but is far from finished. As already mentioned briefly, within a warp of usually 32 threads, the compute instructions for all threads are issued simultaneously. This parallelisation approach is called *SIMT* (*Single Data, Multiple Threads*) [173], specialising the *SIMD* classification in Flynn's taxonomy [174]. If the threads then encounter a branching instruction, part of the threads will execute the first branch while the others are waiting, then switching roles to execute the second branch. Hence, this reduces the level of parallelism that can be achieved and can lead to severe performance impacts when running strongly branching code on a GPU.

With regard to DETRAY, a major source of thread divergence is expected to be the detector geometry, since tracks may encounter different shape types at the same step or diverge in the number of integration steps that need to be performed until the next surface is encountered. Furthermore, most of the tracking-related work, such as the Kalman Filter and Combinatorial Kalman Filter updates, is only done on the sensitive elements of the detector. At most other surfaces, the workload will be quite different, for example, only performing parameter transport and material interactions on passive material surfaces or performing volume changes on volume portals.

In particular, it is expected that the workloads can diverge even more in the current one-thread-per-track model, when one track has to resolve two or more sensitive surfaces in a given detector layer, while a closely neighbouring track on a thread in the same warp may only have to resolve one sensitive surface. This can happen, since there is frequently some spatial overlap between the sensors in silicon trackers to guarantee better hermeticity also for low momentum particles. In this case, the lengthy traversal of the subsequent empty gap-volume between two silicon tracker layers towards the next sensitive surface happens earlier for the second track than for the first one.

Task-based parallelisation strategies like *warp specialisation* [253] or *persistent kernels* [254] might help with this in the future. For example, assigning a group of warps to a propagation sub-task, like stepping or navigation, will reduce the thread divergence to the inherent branching of that task. On the other hand, task-based parallelism requires advanced synchronisation strategies between the tasks, which can in turn entail a big performance impact.

Another issue to be addressed is the complexity of the propagation implementation itself, which results in GPU-kernels that request a large amount of the limited overall register space per thread and can therefore not use the available hardware as efficiently as possible.

6.6.5. Conclusions

An initial performance measurement has been done for the DETRAY propagation, using synthetic benchmarks. Even though the implementation of the benchmarks is still preliminary and the results come with many caveats, the results are encouraging, showing a good scaling behaviour in general. In terms of propagation throughput it seems likely that the GPU-implementation can indeed outperform a CPU-based propagation, if enough tracks are available to fill the device.

This will, however, need to be validated across a variety of different hardware platforms and underpinned by profiling before a reliable statement can be made. In particular, the comparison to an already optimised CPU-based track reconstruction chain like ACTS is missing at this point, as it is possible that the current DETRAY CPU-based propagation is itself not yet efficiently implemented, and therefore a poor baseline for comparing the hardware backends. The feasibility to feed the device with enough tracks to make the dispatch to the GPU beneficial, for example by issuing multiple events, will be determined in the full chain demonstrator TRACCC, where the GPU can be provided with event data by multiple CPU threads.

CHAPTER 7

Conclusions and Outlook

Track reconstruction is an essential ingredient to experimental particle physics, as it provides the momentum measurements and vertex positions of particles that are produced in high energy physics experiments. It is run both online and offline, to identify events that include processes which may contain new physics, as well as allowing the precise characterisation of the data that is the basis of particle physics analyses.

In the computing setup of particle physics experiments, the ability to run event reconstruction on heterogeneous hardware may play an important role, if data centres provide more accelerator hardware in the future. Current particle physics reconstruction code bases will then need to be adapted to and optimised for the new hardware in order to utilise resources most efficiently and provide results that are equally precise to state-of-the-art track reconstruction chains.

In this thesis, work has been done on the design and implementation of a tracking geometry description for silicon detectors based on the tracking chain in the ACTS project, such that it can be deployed in heterogeneous code bases for future track reconstruction frameworks that make use of hardware accelerators. The new tracking geometry description was complemented with a custom navigation implementation and validated on several silicon tracker detector geometries, among them the ATLAS Inner Tracker (ITk).

The DETRAY geometry description has been designed to ensure that it can describe the tracking geometry and material of any detector to the same level of detail as it is possible in ACTS. At the same time, it avoids the usage of common code patterns that are problematic for the deployment on GPUs using, for example, CUDA or SYCL. The geometry and navigation in DETRAY have been implemented without the use of virtual function calls, dynamic memory allocations or pointer based data structures. Instead, flat vector-containers are used that are interlinked with indices encoding type and position of the respective data. This way, the detector data can be constructed on the host ahead of time and copied to the GPU for the device-side reconstruction tasks.

The implementation of the main detector components that make up an ACTS tracking geometry have been completed in DETRAY, with the exception of a tracking volume search data structure. A highly modular and flexible IO library for the conversion from ACTS to DETRAY has been added and is currently used via a frontend for `json` data files. The conversion of a detector from ACTS allows the porting of existing tracking geometries to DETRAY and thus opens the possibility to investigate GPU-accelerated tracking for different experiments in a straight-forward way, without the need to apply further time-consuming adaptations to the new hardware backend. Several optimisations still have to be applied to the IO logic in DETRAY, mainly related to the deduplication of data. This is especially true for the material description, which can grow in size rapidly with the detail of the mapping.

The library provides a rich set of validation tools at this point, ranging from simple consistency checks on the data that make up the detector instance to the detailed validation of its algorithms against truth data. The navigation was thus demonstrated to run successfully on CPU and GPU using CUDA when compared against truth data generated with helical trajectories and a Newton-Raphson/Bisection algorithm. Since this form of truth generation is not guaranteed to find every intersection between the test trajectory and the detector setup, the result constitutes an upper bound on the efficiency with which surfaces are found by the navigator during track propagation.

The correct collection of material during track parameter propagation was investigated and found to be in good agreement with a ray-based geometrical accumulation of the material budget. In rare cases, however, the material found by some rays can differ quite substantially from the material that was encountered by the navigator. The reason for this behaviour is currently not understood and needs further investigation.

Preliminary performance measurements of the CPU-based strong scaling and the track throughput of the CPU- and GPU-based propagation implementations for different track sample sizes were presented. Despite a number of caveats with the measurements which limits the generality of the argument, like missing profiling of overheads, the results are encouraging, showing so far a good strong scaling behaviour up to the number of physical cores of the CPU that was tested and a throughput scaling on the GPU that indicates it might be capable of outperforming the CPU implementation if enough tracks can be sent to the device. A final performance statement for the current DETRAY implementation can, however, only be made once the benchmark implementation is fully validated and has been run on a variety of different hardware setups, as well as including a comparison to existing, better optimised CPU tracking chains like ACTS.

7.1. Future Development

The DETRAY library is under active development within the ACTS R&D project and might in the future be extended to different detector geometries than silicon detectors, such as wire chambers[1] or calorimeters. With the availability of a calorimeter description

[1]A first demonstrator of a wire chamber detector has already been implemented outside the scope of this thesis

in DETRAY, the extension of tracks into surrounding calorimeter systems would become possible and the tracks found in the silicon tracker could be combined with additional information from calorimeter clusters. This might also entail research on the integration of new acceleration data structures besides the currently used surface grids.

The next major milestones for the project will be a full physics validation using more realistic truth data and an in-depth profiling and performance study. Once its new geometry description is fully integrated in ACTS, data can be generated for tracking geometries that are compatible with DETRAY using Geant4. In the long run, also ATLAS simulation data could be used with the ITk geometry. Another improvement that can be achieved within DETRAY as a standalone project will be the use of inhomogeneous magnetic fields, such as the ATLAS field. To this end, the COVFIE library has been integrated to provide a magnetic field description to the stepper and a number of unit tests have been set up so far. A detailed validation of this setup will need to follow.

An important step that will help in understanding the feasibility of running the ACTS tracking chain on GPUs, is to conduct a performance study, both for the complete chain demonstrator, but also for DETRAY in particular. For instance, the DETRAY geometry contains different surface types that incur branching during navigation and track parameter transport, which could have detrimental effects on GPU performance. Furthermore, the complexity of the implementation could impede efficient execution on the device as not enough resources are available to run many threads in parallel. Hints of this were already observed for the DETRAY-related kernels that now run in TRACCC.

With the presented validation, it can be demonstrated that DETRAY is capable of delivering a high quality and navigable tracking geometry description, which allows to find detector surfaces and material with a high accuracy in host and device execution. This is a pivotal corner stone of writing a heterogeneous track reconstruction application as targeted by the TRACCC project.

7.1.1. Integration into ACTS and the traccc Demonstrator

DETRAY is currently being integrated into the TRACCC GPU-tracking demonstrator, as well as the ACTS project. In ACTS, the integration of DETRAY as a plugin will allow an easier exporting of tracking geometries into a GPU-friendly layout without the use of an intermediate data file IO step. This will pave the road for future validation work against the ACTS tracking chain, using the validation workflows present in ACTS. In a first step, it could already be shown that the material accumulated with the DETRAY material scan seems to match the material found by a Geantino scan performed with Geant4 quite well on the ITk. A more rigorous validation of these results including the precise positioning of the mapped material along the test tracks will follow.

In TRACCC, DETRAY will be the main geometry and track state propagation backbone for track finding and fitting. The CKF, for example, calls the DETRAY propagation in order to reach the next surface on a track candidate and perform the covariance transport for it. Another example is the the Kalman Fitter, which implements the fitting step as a DETRAY actor, coupled with a custom aborter. First, highly preliminary tests of the full tracking chain were presented in Ref. [255], using a data set simulated with Geant4 in ACTS on the ODD. These preliminary results already show a promising track finding efficiency of the CKF, as well as a competitive GPU performance for high pile-up events, even though the tested GPUs were outperformed by an NVIDIA Grace CPU [256].

The overall track finding efficiency on the single muon data sample was around 80%, and the track seeding efficiency seems like a probable candidate of where most of the tracks are currently lost. This might simply need some parameter tuning to adapt it to the ODD geometry. All of these results are highly preliminary at this point and not checked to great detail. Optimisations and an in-depth study of both the physics and the compute performance clearly have to follow before any actual statements about the success of the ACTS parallelisation R&D project can be made.

TRACCC is a promising candidate for the deployment of both offline and online track reconstruction on GPUs in the ATLAS experiment, for example, in the *Event Filter* (EF) at the HL-LHC. An investigation is currently underway to determine the performance of TRACCC and other technologies for EF-tracking [190, 191]. If this project has a successful outcome and GPUs can be deployed cost-efficiently, TRACCC may become part of the ATLAS online track reconstruction. TRACCC and DETRAY will also continue to be developed for offline tracking in an experiment independent way. The GPU projects will thus provide a possibility to offload track reconstruction to accelerator chips, especially for experiments that already have an ACTS tracking workflow in place by converting the tracking geometry to the DETRAY layout and then calling a TRACCC-based GPU chain. The details of the final integration of TRACCC into ACTS are currently work in progress.

7.1.2. Vectorization

During the development of the geometry description for this thesis, some effort was also made to prepare performance optimisations on the CPU by using explicit vectorization in the navigation. In particular, the geometry related part of two plugins in the ALGEBRA-PLUGINS project was implemented using the *Vc* library [257, 258] and later integrated into DETRAY for a dedicated version of the ray-surface intersection code.

One of these plugins vectorizes the linear algebra implementation along the dimension of the vectors and matrices used to calculate the intersections (*Array-of-Struct* or AoS). The other is an implementation of an interleaved *Struct-of-Array* layout (SoA), where multiple linear algebra objects are batched and evaluated together. The SoA layout could also prove beneficial for memory accesses outside of vectorization, especially on the GPU. First timing measurements show positive results, both for the linear algebra implementation itself, as well as the intersectors in DETRAY.

In order to bring these plugins fully to DETRAY, some additional work is needed. Currently, a number of methods related to the new algebra plugins are encoded in DETRAY directly, until it is clear what the final interface might look like. Afterwards, it might be possible to find a common implementation for the SoA and the AoS code. Apart from that, the detector memory layout will have to be adapted, as well as the IO modules that construct it.

Bibliography

[1] P. A. M. Dirac, “The quantum theory of the emission and absorption of radiation”, *Proc. R. Soc. London A.*, vol. 114, no. 767, pp. 243–265, 1927. DOI: 10.1098/rspa.1927.0039.

[2] P. A. M. Dirac, “The quantum theory of the electron”, *Proc. Roy. Soc. Lond. A*, vol. 117, pp. 610–624, 1928. DOI: 10.1098/rspa.1928.0023.

[3] P. A. M. Dirac and R. H. Fowler, “A theory of electrons and protons”, *Proc. Roy. Soc. Lond. A*, vol. 126, no. 801, pp. 360–365, 1930. DOI: 10.1098/rspa.1930.0013.

[4] H. A. Bethe, “The Electromagnetic Shift of Energy Levels”, *Phys. Rev.*, vol. 72, pp. 339–341, 4 Aug. 1947. DOI: 10.1103/PhysRev.72.339.

[5] S. Tomonaga, “On a relativistically invariant formulation of the quantum theory of wave fields”, *Prog. Theor. Phys.*, vol. 1, pp. 27–42, 1946. DOI: 10.1143/PTP.1.27.

[6] J. S. Schwinger, “On Quantum electrodynamics and the magnetic moment of the electron”, *Phys. Rev.*, vol. 73, pp. 416–417, 1948. DOI: 10.1103/PhysRev.73.416.

[7] J. Schwinger, “Quantum Electrodynamics. I. A Covariant Formulation”, *Phys. Rev.*, vol. 74, pp. 1439–1461, 10 Nov. 1948. DOI: 10.1103/PhysRev.74.1439.

[8] R. P. Feynman, “Space-Time Approach to Quantum Electrodynamics”, *Phys. Rev.*, vol. 76, pp. 769–789, 6 Sep. 1949. DOI: 10.1103/PhysRev.76.769.

[9] R. P. Feynman, “The Theory of Positrons”, *Phys. Rev.*, vol. 76, pp. 749–759, 6 Sep. 1949. DOI: 10.1103/PhysRev.76.749.

[10] R. P. Feynman, “Mathematical Formulation of the Quantum Theory of Electromagnetic Interaction”, *Phys. Rev.*, vol. 80, pp. 440–457, 3 Nov. 1950. DOI: 10.1103/PhysRev.80.440.

[11] H. Yukawa, “On the Interaction of Elementary Particles I”, *Proc. Phys. Math. Soc. Jap.*, vol. 17, pp. 48–57, 1935. DOI: 10.1143/PTPS.1.1.

[12] C. N. Yang and R. L. Mills, “Conservation of Isotopic Spin and Isotopic Gauge Invariance”, *Phys. Rev.*, vol. 96, pp. 191–195, 1 Oct. 1954. DOI: 10.1103/PhysRev.96.191.

[13] H. Fritzsch and M. Gell-Mann, “Current algebra: Quarks and what else?”, *eConf*, vol. C720906V2, J. D. Jackson and A. Roberts, Eds., pp. 135–165, 1972. DOI: 10.48550/arxiv.hep-ph/0208010. arXiv: hep-ph/0208010.

[14] H. Fritzsch, M. Gell-Mann, and H. Leutwyler, “Advantages of the Color Octet Gluon Picture”, *Phys. Lett. B*, vol. 47, pp. 365–368, 1973. DOI: 10.1016/0370-2693(73)90625-4.

[15] M. Gell-Mann, “The Eightfold Way: A Theory of strong interaction symmetry”, Mar. 1961. DOI: 10.2172/4008239.

[16] Y. Ne’eman, “Derivation of strong interactions from a gauge invariance”, *Nucl. Phys.*, vol. 26, no. 2, pp. 222–229, 1961. DOI: 10.1016/0029-5582(61)90134-1.

[17] J. Chadwick, “Intensitätsverteilung im magnetischen Spectrum der β-Strahlen von radium B + C”, *Verhandl. Dtsc. Phys. Ges.*, vol. 16, p. 383, 1914.

[18] E. Fermi, “An attempt of a theory of beta radiation. 1.”, *Z. Phys.*, vol. 88, pp. 161–177, 1934. DOI: 10.1007/BF01351864.

[19] W. Pauli, “Dear radioactive ladies and gentlemen”, *Phys. Today*, vol. 31N9, p. 27, 1978.

[20] G. ’t Hooft and M. Veltman, “Regularization and renormalization of gauge fields”, *Nucl. Phys. B.*, vol. 44, no. 1, pp. 189–213, 1972, ISSN: 0550-3213. DOI: 10.1016/0550-3213(72)90279-9.

[21] MARK-J Collaboration, “Discovery of Three-Jet Events and a Test of Quantum Chromodynamics at PETRA”, *Phys. Rev. Lett.*, vol. 43, pp. 830–833, 12 Sep. 1979. DOI: 10.1103/PhysRevLett.43.830.

[22] TASSO Collaboration, “Evidence for planar events in e^+e^- annihilation at high energies”, *Phys. Lett. B*, vol. 86, no. 2, pp. 243–249, 1979. DOI: 10.1016/0370-2693(79)90830-X.

[23] JADE Collaboration, “Observation of planar three-jet events in e^+e^- annihilation and evidence for gluon bremsstrahlung”, *Phys. Lett. B*, vol. 91, no. 1, pp. 142–147, 1980. DOI: 10.1016/0370-2693(80)90680-2.

[24] UA1 Collaboration, “Experimental Observation of Lepton Pairs of Invariant Mass Around 95 GeV/c^2 at the CERN SPS Collider”, *Phys. Lett. B*, vol. 126, pp. 398–410, 1983. DOI: 10.1016/0370-2693(83)90188-0.

[25] UA2 Collaboration, “Evidence for $Z^0 \to e^+e^-$ at the CERN pp collider”, *Phys. Lett. B*, vol. 129, no. 1, pp. 130–140, 1983, ISSN: 0370-2693. DOI: 10.1016/0370-2693(83)90744-X.

[26] UA1 Collaboration, “Experimental Observation of Isolated Large Transverse Energy Electrons with Associated Missing Energy at $\sqrt{s} = 540$ GeV”, *Phys. Lett. B*, vol. 122, pp. 103–116, 1983. DOI: 10.1016/0370-2693(83)91177-2.

[27] UA2 Collaboration, “Observation of Single Isolated Electrons of High Transverse Momentum in Events with Missing Transverse Energy at the CERN $p\bar{p}$ Collider”, *Phys. Lett. B*, vol. 122, pp. 476–485, 1983. DOI: 10.1016/0370-2693(83)91605-2.

[28] OPAL Collaboration, "Measurement of the mass of the W boson in e^+e^- collisions at $s = 161\,\text{GeV}$", *Phys. Lett. B*, vol. 389, no. 2, pp. 416–428, 1996. DOI: 10.1016/S0370-2693(96)01452-9.

[29] L3 Collaboration, "Measurement of W-pair cross sections in e^+e^- interactions at $s = 172\,\text{GeV}$ and W-decay branching fractions", *Phys. Lett. B*, vol. 407, no. 3, pp. 419–431, 1997. DOI: 10.1016/S0370-2693(97)00802-2.

[30] ALEPH Collaboration, "Measurement of W-pair production in e^+e^- collisions at 183 GeV", *Phys. Lett. B*, vol. 453, no. 1, pp. 107–120, 1999. DOI: 10.1016/S0370-2693(99)00304-4.

[31] S. L. Glashow, "Partial-symmetries of weak interactions", *Nuclear Physics*, vol. 22, no. 4, pp. 579–588, 1961, ISSN: 0029-5582. DOI: https://doi.org/10.1016/0029-5582(61)90469-2.

[32] S. Weinberg, "A Model of Leptons", *Phys. Rev. Lett.*, vol. 19, pp. 1264–1266, 1967. DOI: 10.1103/PhysRevLett.19.1264.

[33] A. Salam, "Weak and Electromagnetic Interactions", *Conf. Proc. C*, vol. 680519, pp. 367–377, 1968. DOI: 10.1142/9789812795915_0034.

[34] P. W. Higgs, "Broken Symmetries and the Masses of Gauge Bosons", *Phys. Rev. Lett.*, vol. 13, J. C. Taylor, Ed., pp. 508–509, 1964. DOI: 10.1103/PhysRevLett.13.508.

[35] P. W. Higgs, "Broken symmetries, massless particles and gauge fields", *Phys. Lett.*, vol. 12, pp. 132–133, 1964. DOI: 10.1016/0031-9163(64)91136-9.

[36] F. Englert and R. Brout, "Broken Symmetry and the Mass of Gauge Vector Mesons", *Phys. Rev. Lett.*, vol. 13, J. C. Taylor, Ed., pp. 321–323, 1964. DOI: 10.1103/PhysRevLett.13.321.

[37] G. S. Guralnik, C. R. Hagen, and T. W. B. Kibble, "Global Conservation Laws and Massless Particles", *Phys. Rev. Lett.*, vol. 13, J. C. Taylor, Ed., pp. 585–587, 1964. DOI: 10.1103/PhysRevLett.13.585.

[38] P. W. Higgs, "Spontaneous symmetry breakdown without massless bosons", *Phys. Rev.*, vol. 145, pp. 1156–1163, 4 May 1966. DOI: 10.1103/PhysRev.145.1156.

[39] T. W. B. Kibble, "Symmetry Breaking in Non-Abelian Gauge Theories", *Phys. Rev.*, vol. 155, pp. 1554–1561, 5 Mar. 1967. DOI: 10.1103/PhysRev.155.1554.

[40] ATLAS Collaboration, "Observation of a new particle in the search for the Standard Model Higgs boson with the ATLAS detector at the LHC", *Phys. Lett. B*, vol. 716, pp. 1–29, 2012. DOI: 10.1016/j.physletb.2012.08.020.

[41] CMS Collaboration, "Observation of a New Boson at a Mass of 125 GeV with the CMS Experiment at the LHC", *Phys. Lett. B*, vol. 716, pp. 30–61, 2012. DOI: 10.1016/j.physletb.2012.08.021.

[42] Particle Data Group, "Review of Particle Physics", *Prog. Theor. Exp. Phys.*, vol. 2022, no. 8, p. 083C01, Aug. 2022, ISSN: 2050-3911. DOI: 10.1093/ptep/ptac097.

[43] C. L. Cowan *et al.*, “Detection of the free neutrino: A Confirmation”, *Science*, vol. 124, pp. 103–104, 1956. DOI: 10.1126/science.124.3212.103.

[44] J. J. Thomson, “Cathode rays”, *Philos. Mag.*, vol. 44, no. 269, pp. 293–316, 1897.

[45] C. D. Anderson, “The Apparent Existence of Easily Deflectable Positives”, *Science*, vol. 76, no. 1967, pp. 238–239, 1932. DOI: 10.1126/science.76.1967.238.

[46] C. D. Anderson, “The Positive Electron”, *Phys. Rev.*, vol. 43, pp. 491–494, 6 Mar. 1933. DOI: 10.1103/PhysRev.43.491.

[47] C. D. Anderson and S. H. Neddermeyer, “Cloud Chamber Observations of Cosmic Rays at 4300 Meters Elevation and Near Sea-Level”, *Phys. Rev.*, vol. 50, pp. 263–271, 4 Aug. 1936. DOI: 10.1103/PhysRev.50.263.

[48] M. L. Perl *et al.*, “Evidence for Anomalous Lepton Production in $e^+ - e^-$ Annihilation”, *Phys. Rev. Lett.*, vol. 35, pp. 1489–1492, 22 Dec. 1975. DOI: 10.1103/PhysRevLett.35.1489.

[49] DONUT Collaboration, “Observation of tau neutrino interactions”, *Phys. Lett. B*, vol. 504, no. 3, pp. 218–224, 2001. DOI: 10.1016/S0370-2693(01)00307-0.

[50] G. Danby *et al.*, “Observation of High-Energy Neutrino Reactions and the Existence of Two Kinds of Neutrinos”, *Phys. Rev. Lett.*, vol. 9, pp. 36–44, 1 Jul. 1962. DOI: 10.1103/PhysRevLett.9.36.

[51] M. Breidenbach *et al.*, “Observed Behavior of Highly Inelastic Electron-Proton Scattering”, *Phys. Rev. Lett.*, vol. 23, pp. 935–939, 16 Oct. 1969. DOI: 10.1103/PhysRevLett.23.935.

[52] E. D. Bloom *et al.*, “High-Energy Inelastic $e - p$ Scattering at 6° and 10°”, *Phys. Rev. Lett.*, vol. 23, pp. 930–934, 16 Oct. 1969. DOI: 10.1103/PhysRevLett.23.930.

[53] SLAC-SP-017 Collaboration, “Discovery of a Narrow Resonance in e^+e^- Annihilation”, *Phys. Rev. Lett.*, vol. 33, pp. 1406–1408, 23 Dec. 1974. DOI: 10.1103/PhysRevLett.33.1406.

[54] E598 Collaboration, “Experimental Observation of a Heavy Particle J”, *Phys. Rev. Lett.*, vol. 33, pp. 1404–1406, 23 Dec. 1974. DOI: 10.1103/PhysRevLett.33.1404.

[55] S. W. Herb *et al.*, “Observation of a Dimuon Resonance at 9.5 GeV in 400 − GeV Proton-Nucleus Collisions”, *Phys. Rev. Lett.*, vol. 39, pp. 252–255, 5 Aug. 1977. DOI: 10.1103/PhysRevLett.39.252.

[56] M. Gell-Mann, “A Schematic Model of Baryons and Mesons”, *Phys. Lett.*, vol. 8, pp. 214–215, 1964. DOI: 10.1016/S0031-9163(64)92001-3.

[57] G. Zweig, “An SU(3) model for strong interaction symmetry and its breaking. Version 2”, D. B. Lichtenberg and S. P. Rosen, Eds., pp. 22–101, Feb. 1964. DOI: 10.17181/CERN-TH-412.

[58] N. Cabibbo, “Unitary Symmetry and Leptonic Decays”, *Phys. Rev. Lett.*, vol. 10, pp. 531–533, 12 Jun. 1963. DOI: 10.1103/PhysRevLett.10.531.

[59] M. Kobayashi and T. Maskawa, "CP Violation in the Renormalizable Theory of Weak Interaction", *Prog. Theor. Phys.*, vol. 49, pp. 652–657, 1973. DOI: 10.1143/PTP.49.652.

[60] C. S. Wu *et al.*, "Experimental Test of Parity Conservation in Beta Decay", *Phys. Rev.*, vol. 105, pp. 1413–1415, 4 Feb. 1957. DOI: 10.1103/PhysRev.105.1413.

[61] J. H. Christenson *et al.*, "Evidence for the 2π Decay of the K_2^0 Meson", *Phys. Rev. Lett.*, vol. 13, pp. 138–140, 4 Jul. 1964. DOI: 10.1103/PhysRevLett.13.138.

[62] B. Pontecorvo, "Inverse beta processes and nonconservation of lepton charge", *Zh. Eksp. Teor. Fiz.*, vol. 34, p. 247, 1957.

[63] Z. Maki, M. Nakagawa, and S. Sakata, "Remarks on the unified model of elementary particles", *Prog. Theor. Phys.*, vol. 28, pp. 870–880, 1962. DOI: 10.1143/PTP.28.870.

[64] R. Davis, D. S. Harmer, and K. C. Hoffman, "Search for Neutrinos from the Sun", *Phys. Rev. Lett.*, vol. 20, pp. 1205–1209, 21 May 1968. DOI: 10.1103/PhysRevLett.20.1205.

[65] Super-Kamiokande Collaboration, "Evidence for oscillation of atmospheric neutrinos", *Phys. Rev. Lett.*, vol. 81, pp. 1562–1567, 8 Aug. 1998. DOI: 10.1103/PhysRevLett.81.1562.

[66] CDF Collaboration, "Observation of Top Quark Production in $\overline{p}p$ Collisions with the Collider Detector at Fermilab", *Phys. Rev. Lett.*, vol. 74, pp. 2626–2631, 14 Apr. 1995. DOI: 10.1103/PhysRevLett.74.2626.

[67] DØ Collaboration, "Observation of the Top Quark", *Phys. Rev. Lett.*, vol. 74, pp. 2632–2637, 14 Apr. 1995. DOI: 10.1103/PhysRevLett.74.2632.

[68] A. D. Sakharov, "Violation of CP invariance, C asymmetry, and baryon asymmetry of the universe", *Sov. Phys. Uspekhi*, vol. 34, no. 5, p. 392, May 1991. DOI: 10.1070/PU1991v034n05ABEH002497.

[69] H. A. Bethe, "Zur Theorie des Durchgangs schneller Korpuskularstrahlen durch Materie", *Ann. Phys.*, vol. 397, pp. 325–400, 1930.

[70] B. B. Rossi, *High-energy particles* (Prentice-Hall physics series). New York, NY: Prentice-Hall, 1952.

[71] M. S. Livingston and H. A. Bethe, "Nuclear Physics C. Nuclear Dynamics, Experimental", *Rev. Mod. Phys.*, vol. 9, pp. 245–390, 3 Jul. 1937. DOI: 10.1103/RevModPhys.9.245.

[72] M. Berger *et al.*, *Stopping powers for protons and alpha particles*, 1993.

[73] R. K. Bull, "Stopping powers for electrons and positrons: ICRU Report 37", *Int. J. Radiat. Appl. Instrum. Part D*, vol. 11, p. 273, 1986.

[74] F. Bloch, "Zur Bremsung Rasch Bewegter Teilchen beim Durchgang durch Materie", *Ann. Phys.*, vol. 408, pp. 285–320, 1933. DOI: 10.1002/andp.19334080303.

[75] F. Bloch, "Bremsvermögen von Atomen mit mehreren Elektronen", *Z. Phys.*, vol. 81, pp. 363–376, 1933.

[76] H. Bichsel, "Shell corrections in stopping powers", *Phys. Rev. A*, vol. 65, p. 052 709, 5 Apr. 2002. DOI: 10.1103/PhysRevA.65.052709.

[77] S. M. Seltzer and M. J. Berger, "Evaluation of the collision stopping power of elements and compounds for electrons and positrons", *Appl. Radiat. Isot.*, vol. 33, no. 11, pp. 1189–1218, 1982, ISSN: 0020-708X. DOI: https://doi.org/10.1016/0020-708X(82)90244-7.

[78] L. D. Landau, "On the Energy Loss of Fast Particles by Ionisation", *J. Phys. (USSR)*, vol. 8, D. ter Haar, Ed., 1944. DOI: 10.1016/b978-0-08-010586-4.50061-4.

[79] P. V. Vavilov, "Ionization losses of high-energy heavy particles", *Sov. Phys. JETP*, vol. 5, pp. 749–751, 1957.

[80] H. Bichsel, "Straggling in Thin Silicon Detectors", *Rev. Mod. Phys.*, vol. 60, pp. 663–699, 1988. DOI: 10.1103/RevModPhys.60.663.

[81] H. Bethe and W. Heitler, "On the stopping of fast particles and on the creation of positive electrons", *Proc. R. Soc. London A.*, vol. 146, no. 856, pp. 83–112, 1934.

[82] M. J. Berger and S. M. Seltzer, "Tables of energy losses and ranges of electrons and positrons", Tech. Rep., 1964.

[83] Y.-S. Tsai, "Pair Production and Bremsstrahlung of Charged Leptons", *Rev. Mod. Phys.*, vol. 46, p. 815, 1974, [Erratum: Rev.Mod.Phys. 49, 421–423 (1977)]. DOI: 10.1103/RevModPhys.46.815.

[84] G. Molière, "Theorie der Streuung schneller geladener Teilchen I. Einzelstreuung am abgeschirmten Coulomb-Feld", *Z. Naturforsch. A*, vol. 2, no. 3, pp. 133–145, 1947. DOI: 10.1515/zna-1947-0302.

[85] H. A. Bethe, "Molière's Theory of Multiple Scattering", *Phys. Rev.*, vol. 89, pp. 1256–1266, 6 Mar. 1953. DOI: 10.1103/PhysRev.89.1256.

[86] B. Rossi and K. Greisen, "Cosmic-Ray Theory", *Rev. Mod. Phys.*, vol. 13, pp. 240–309, 4 Oct. 1941. DOI: 10.1103/RevModPhys.13.240.

[87] V. L. Highland, "Some practical remarks on multiple scattering", *Nucl. Instrum. Methods*, vol. 129, no. 2, pp. 497–499, 1975. DOI: 10.1016/0029-554X(75)90743-0.

[88] G. R. Lynch and O. I. Dahl, "Approximations to multiple Coulomb scattering", *Nucl. Instrum. Methods Phys. Res., Sect. B*, vol. 58, no. 1, pp. 6–10, 1991. DOI: 10.1016/0168-583X(91)95671-Y.

[89] ATLAS Collaboration, "The ATLAS Experiment at the CERN Large Hadron Collider", *J. Instrum.*, vol. 3, no. 08, S08003, Aug. 2008. DOI: 10.1088/1748-0221/3/08/S08003.

[90] ALICE Collaboration, "The ALICE experiment at the CERN LHC", *JINST*, vol. 3, S08002, 2008. DOI: 10.1088/1748-0221/3/08/S08002.

[91] CMS Collaboration, "The CMS Experiment at the CERN LHC", *JINST*, vol. 3, S08004, 2008. DOI: 10.1088/1748-0221/3/08/S08004.

[92] LHCb Collaboration, "The LHCb Detector at the LHC", *JINST*, vol. 3, S08005, 2008. DOI: 10.1088/1748-0221/3/08/S08005.

[93] ATLAS Collaboration, "Letter of Intent for the Phase-II Upgrade of the ATLAS Experiment", 2012, Draft version for comments.

[94] E. Lopienska, "The CERN accelerator complex, layout in 2022. Complexe des accélérateurs du CERN en janvier 2022", 2022, General Photo. [Online]. Available: https://cds.cern.ch/record/2800984.

[95] L. Evans and P. Bryant, "LHC Machine", *J. Instrum.*, vol. 3, no. 08, S08001, Aug. 2008. DOI: 10.1088/1748-0221/3/08/S08001.

[96] K. Hübner, "Designing and building LEP", *Phys. Rep.*, vol. 403-404, pp. 177–188, 2004, ISSN: 0370-1573. DOI: https://doi.org/10.1016/j.physrep.2004.09.004.

[97] ATLAS Collaboration, "Luminosity determination in *pp* collisions at $\sqrt{s} = 13$ TeV using the ATLAS detector at the LHC", *Eur. Phys. J. C*, vol. 83, no. 10, p. 982, 2023. DOI: 10.1140/epjc/s10052-023-11747-w.

[98] Bianchi, Riccardo Maria and ATLAS Collaboration, "ATLAS experiment schematic or layout illustration", General Photo, 2022. [Online]. Available: https://cds.cern.ch/record/2837191.

[99] ATLAS Collaboration, "The ATLAS experiment at the CERN Large Hadron Collider: a description of the detector configuration for Run-3", *J. Instrum.*, vol. 19, no. 05, P05063, May 2024. DOI: 10.1088/1748-0221/19/05/P05063.

[100] ATLAS Collaboration, "ATLAS central solenoid: Technical design report", no. CERN-LHCC-97-21, Apr. 1997.

[101] ATLAS Collaboration, "ATLAS barrel toroid: Technical Design Report", Technical design report. ATLAS, 1997. DOI: 10.17181/CERN.RF2A.CP5T.

[102] ATLAS Collaboration, *ATLAS end-cap toroids: Technical Design Report* (Technical design report. ATLAS). Geneva: CERN, 1997. DOI: 10.17181/CERN.P03D.WQLV.

[103] ATLAS Collaboration, *ATLAS inner detector: Technical Design Report, 1* (Technical design report. ATLAS). Geneva: CERN, 1997.

[104] ATLAS Collaboration, *ATLAS inner detector: Technical Design Report, 2* (Technical design report. ATLAS). Geneva: CERN, 1997.

[105] ATLAS Collaboration, "ATLAS Insertable B-Layer Technical Design Report", Tech. Rep. CERN-LHCC-2010-013, ATLAS-TDR-19, 2010.

[106] J. Kemmer and G. Lutz, "New detector concepts", *Nucl. Instrum. Methods Phys. Res., Sect. A*, vol. 253, no. 3, pp. 365–377, 1987. DOI: 10.1016/0168-9002(87)90518-3.

[107] G. Lutz, *Semiconductor Radiation Detectors: Device Physics*. New York: Springer, 1999, ISBN: 978-3-540-64859-8.

[108] I. Perić *et al.*, "The FEI3 readout chip for the ATLAS pixel detector", *Nucl. Instrum. Methods Phys. Res., Sect. A*, vol. 565, no. 1, pp. 178–187, 2006, Proceedings of the International Workshop on Semiconductor Pixel Detectors for Particles and Imaging. DOI: 10.1016/j.nima.2006.05.032.

[109] I. Gorelov *et al.*, "A measurement of Lorentz angle and spatial resolution of radiation hard silicon pixel sensors", *Nucl. Instrum. Meth. A*, vol. 481, pp. 204–221, 2002. DOI: 10.1016/S0168-9002(01)01413-9.

[110] ATLAS Collaboration, "IBL Efficiency and Single Point Resolution in Collision Events", CERN, Geneva, Tech. Rep., 2016.

[111] A. Abdesselam *et al.*, "The barrel modules of the ATLAS semiconductor tracker", *Nucl. Instrum. Methods Phys. Res., Sect. A*, vol. 568, no. 2, pp. 642–671, 2006. DOI: 10.1016/j.nima.2006.08.036.

[112] A. Abdesselam *et al.*, "The ATLAS semiconductor tracker end-cap module", *Nucl. Instrum. Methods Phys. Res., Sect. A*, vol. 575, no. 3, pp. 353–389, 2007. DOI: 10.1016/j.nima.2007.02.019.

[113] ATLAS Collaboration, "The ATLAS Inner Detector commissioning and calibration", *Eur. Phys. J. C*, vol. 70, pp. 787–821, 2010.

[114] ATLAS Collaboration, "The ATLAS Transition Radiation Tracker (TRT) proportional drift tube: Design and performance", Tech. Rep., Feb. 2008, P02013. DOI: 10.1088/1748-0221/3/02/P02013.

[115] V. Ginzburg and I. Frank, "Radiation of a uniformly moving electron due to its transition from one medium into another", *J. Phys. (USSR)*, vol. 9, pp. 353–362, 1945.

[116] G. Garibian, "Contribution to the theory of transition radiation", *Sov. Phys. JETP*, vol. 6, no. 6, p. 1079, 1958.

[117] G. Garibian, "Transition radiation effects in particle energy losses", *Sov. Phys. JETP*, vol. 37, no. 10, p. 2, 1960.

[118] ATLAS Collaboration, *ATLAS liquid-argon calorimeter: Technical Design Report* (Technical design report. ATLAS). Geneva: CERN, 1996. DOI: 10.17181/CERN.FWRW.FOOQ.

[119] ATLAS Collaboration, *ATLAS tile calorimeter: Technical Design Report* (Technical design report. ATLAS). Geneva: CERN, 1996. DOI: 10.17181/CERN.JRBJ.7028.

[120] ATLAS Collaboration, *ATLAS muon spectrometer: Technical Design Report* (Technical design report. ATLAS). Geneva: CERN, 1997. [Online]. Available: https://cds.cern.ch/record/331068.

[121] ATLAS Collaboration, "New Small Wheel Technical Design Report", no. CERN-LHCC-2013-006, ATLAS-TDR-020, 2013, ATLAS New Small Wheel Technical Design Report. [Online]. Available: https://cds.cern.ch/record/1552862.

[122] ATLAS Collaboration, "ATLAS level-1 trigger: Technical Design Report", Technical design report. ATLAS, 1998. [Online]. Available: https://cds.cern.ch/record/381429.

[123] ATLAS Collaboration, "ATLAS high-level trigger, data-acquisition and controls: Technical Design Report", Technical design report. ATLAS, 2003. [Online]. Available: https://cds.cern.ch/record/616089.

[124] ATLAS Collaboration, "ATLAS Computing: technical design report", Technical design report. ATLAS, 2005. [Online]. Available: https://cds.cern.ch/record/837738.

[125] HiLumi HL-LHC Project, *HL-LHC Project Schedule*. Accessed: Jul. 7, 2025. [Online]. Available: https://project-hl-lhc-industry.web.cern.ch/content/project-schedule.

[126] ATLAS Collaboration, "ATLAS HL-LHC Computing Conceptual Design Report", CERN, Geneva, Tech. Rep. CERN-LHCC-2020-015, LHCC-G-178, 2020.

[127] ATLAS Collaboration, "Expected tracking and related performance with the updated ATLAS Inner Tracker layout at the High-Luminosity LHC", 2021, All figures including auxiliary figures are available at https://atlas.web.cern.ch/Atlas/GROUPS/PHYSICS/PUBNOTES/ATL-PHYS-PUB-2021-024.

[128] ATLAS Collaboration, "Technical Design Report for the ATLAS Inner Tracker Pixel Detector", CERN, Geneva, Tech. Rep. CERN-LHCC-2017-021, ATLAS-TDR-030, 2017. DOI: 10.17181/CERN.FOZZ.ZP3Q.

[129] ATLAS Collaboration, "Technical Design Report for the ATLAS Inner Tracker Strip Detector", CERN, Geneva, Tech. Rep. CERN-LHCC-2017-005, ATLAS-TDR-025, 2017.

[130] N. Hessey, "Building a Stereo-angle into strip-sensors for the ATLAS-Upgrade Inner-Tracker Endcaps", 2013. [Online]. Available: https://cds.cern.ch/record/1514636.

[131] T. G. Cornelissen *et al.*, "The global χ^2 track fitter in ATLAS", *Journal of Physics: Conference Series*, vol. 119, no. 3, p. 032 013, Jul. 2008. DOI: 10.1088/1742-6596/119/3/032013.

[132] R. O. Duda and P. E. Hart, "Use of the Hough transformation to detect lines and curves in pictures", *Commun. ACM*, vol. 15, pp. 11–15, 1972.

[133] ATLAS Collaboration, “Software Performance of the ATLAS Track Reconstruction for LHC Run-3”, *Computing and Software for Big Science*, vol. 8, Apr. 2024. DOI: 10.1007/s41781-023-00111-y.

[134] R. Frühwirth and A. Strandlie, “Pattern Recognition, Tracking and Vertex Reconstruction in Particle Detectors”, p. 203, Jan. 2021.

[135] T. Cornelissen *et al.*, “The new ATLAS track reconstruction (NEWT)”, *J. Phys. Conf. Ser.*, vol. 119, no. 3, p. 032 014, Jul. 2008. DOI: 10.1088/1742-6596/119/3/032014.

[136] ATLAS Collaboration, “Performance of the ATLAS Track Reconstruction Algorithms in Dense Environments in LHC Run-2”, *Eur. Phys. J. C*, vol. 77, no. 10, p. 673, 2017. DOI: 10.1140/epjc/s10052-017-5225-7.

[137] A. Rosenfeld and J. Pfaltz, “Sequential operations in digital picture processing”, *J. ACM*, vol. 13, pp. 471–494, Oct. 1966. DOI: 10.1145/321356.321357.

[138] K. Selbach, “Neural network based cluster reconstruction in the ATLAS pixel detector”, *Nucl. Instrum. Methods Phys. Res., Sect. A*, vol. 718, pp. 363–365, 2013, Proceedings of the 12th Pisa Meeting on Advanced Detectors, ISSN: 0168-9002. DOI: 10.1016/j.nima.2012.10.033.

[139] ATLAS collaboration, “A neural network clustering algorithm for the ATLAS silicon pixel detector”, *J. Instrum.*, vol. 9, no. 09, P09009, Sep. 2014. DOI: 10.1088/1748-0221/9/09/P09009.

[140] ATLAS Collaboration, “Performance of the ATLAS Silicon Pattern Recognition Algorithm in Data and Simulation at $\sqrt{s} = 7$ TeV”, CERN, Geneva, Tech. Rep. ATLAS-CONF-2010-072, 2010. [Online]. Available: https://cds.cern.ch/record/1281363.

[141] W. Wittek, “Transformation of error matrices for different sets of variables which describe a particle trajectory in a magnetic field”, CERN, Geneva, Tech. Rep. EMCSW-80-39, 1980. [Online]. Available: https://cds.cern.ch/record/931167.

[142] A. Strandlie and W. Wittek, “Derivation of Jacobians for the propagation of covariance matrices of track parameters in homogeneous magnetic fields”, *Nucl. Instrum. Methods Phys. Res., Sect. A*, vol. 566, pp. 687–698, Oct. 2006. DOI: 10.1016/j.nima.2006.07.032.

[143] P. F. Åkesson *et al.*, “ATLAS Tracking Event Data Model”, CERN, Geneva, Tech. Rep. ATL-SOFT-PUB-2006-004, ATL-COM-SOFT-2006-005, CERN-ATL-COM-SOFT-2006-005, 2006. [Online]. Available: https://cds.cern.ch/record/973401.

[144] T. G. Cornelissen *et al.*, “Updates of the ATLAS Tracking Event Data Model (Release 13)”, CERN, Geneva, Tech. Rep. ATL-SOFT-PUB-2007-003, ATL-COM-SOFT-2007-008, 2007. [Online]. Available: https://cds.cern.ch/record/1038095.

[145] J. Myrheim and L. Bugge, “A fast Runge-Kutta method for fitting tracks in a magnetic field”, *Nucl. Instrum. Methods*, vol. 160, no. 1, pp. 43–48, 1979, ISSN: 0029-554X. DOI: 10.1016/0029-554X(79)90163-0.

[146] E. Lund *et al.*, “Track parameter propagation through the application of a new adaptive Runge-Kutta-Nyström method in the ATLAS experiment”, *J. Instrum.*, vol. 4, no. 04, P04001, Apr. 2009. DOI: 10.1088/1748-0221/4/04/P04001.

[147] L. Bugge and J. Myrheim, “Tracking and track fitting”, *Nucl. Instrum. Methods*, vol. 179, no. 2, pp. 365–381, 1981, ISSN: 0029-554X. DOI: 10.1016/0029-554X(81)90063-X.

[148] E. Lund *et al.*, “Transport of covariance matrices in the inhomogeneous magnetic field of the ATLAS experiment by the application of a semi-analytical method”, *J. Instrum.*, vol. 4, no. 04, P04016, Apr. 2009. DOI: 10.1088/1748-0221/4/04/P04016.

[149] W. Wittek, “Propagation of errors along a particle trajectory in a magnetic field”, CERN, Geneva, Tech. Rep. EMC-80-15, 1980. [Online]. Available: https://cds.cern.ch/record/931170.

[150] W. Wittek, “Error propagation along a helix”, CERN, Geneva, Tech. Rep. EMCSW-81-18, 1981. [Online]. Available: https://cds.cern.ch/record/931174.

[151] R. E. Kalman, “A New Approach to Linear Filtering and Prediction Problems”, *Journal of Basic Engineering*, vol. 82, no. 1, pp. 35–45, Mar. 1960, ISSN: 0021-9223. DOI: 10.1115/1.3662552.

[152] R. Frühwirth, “Application of Kalman filtering to track and vertex fitting”, *Nucl. Instrum. Methods Phys. Res., Sect. A*, vol. 262, no. 2, pp. 444–450, 1987, ISSN: 0168-9002. DOI: https://doi.org/10.1016/0168-9002(87)90887-4.

[153] R. Frühwirth, “Track fitting with non-Gaussian noise”, *Comput. Phys. Commun.*, vol. 100, no. 1, pp. 1–16, 1997, ISSN: 0010-4655. DOI: https://doi.org/10.1016/S0010-4655(96)00155-5.

[154] R. Frühwirth and S. Frühwirth-Schnatter, “On the treatment of energy loss in track fitting”, *Comput. Phys. Commun.*, vol. 110, no. 1, pp. 80–86, 1998, ISSN: 0010-4655. DOI: https://doi.org/10.1016/S0010-4655(97)00157-4.

[155] R. Frühwirth and A. Strandlie, “Track fitting with ambiguities and noise: A study of elastic tracking and nonlinear filters”, *Computer Physics Communications*, vol. 120, no. 2, pp. 197–214, 1999, ISSN: 0010-4655. DOI: https://doi.org/10.1016/S0010-4655(99)00231-3.

[156] R. Frühwirth and A. Strandlie, “Track fitting with ambiguities and noise: A study of elastic tracking and nonlinear filters”, *Computer Physics Communications*, vol. 120, no. 2, pp. 197–214, 1999, ISSN: 0010-4655. DOI: https://doi.org/10.1016/S0010-4655(99)00231-3.

[157] R. Mankel, "A concurrent track evolution algorithm for pattern recognition in the HERA-B main tracking system", *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 395, no. 2, pp. 169–184, 1997, ISSN: 0168-9002. DOI: https://doi.org/10.1016/S0168-9002(97)00705-5.

[158] R. Mankel and A. Spiridonov, "The Concurrent Track Evolution algorithm: extension for track finding in the inhomogeneous magnetic field of the HERA-B spectrometer", *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 426, no. 2, pp. 268–282, 1999, ISSN: 0168-9002. DOI: https://doi.org/10.1016/S0168-9002(99)00013-3.

[159] R. Mankel, "Pattern recognition and event reconstruction in particle physics experiments", *Reports on Progress in Physics*, vol. 67, no. 4, p. 553, Mar. 2004. DOI: 10.1088/0034-4885/67/4/R03.

[160] X. Ai *et al.*, "A Common Tracking Software Project", *Comput. Softw. Big Sci.*, vol. 6, no. 1, p. 8, 2022. DOI: 10.1007/s41781-021-00078-8. arXiv: 2106.13593 [physics.ins-det].

[161] A. Salzburger *et al.*, *Acts-project/acts*, Jul. 2025. DOI: 10.5281/zenodo.5141418. [Online]. Available: https://zenodo.org/doi/10.5281/zenodo.5141418.

[162] J. Osborn *et al.*, "Implementation of ACTS into sPHENIX Track Reconstruction", *Computing and Software for Big Science*, vol. 5, Dec. 2021. DOI: 10.1007/s41781-021-00068-w.

[163] A. Salzburger, S. Todorova, and M. Wolter, "The ATLAS Tracking Geometry Description", CERN, Geneva, Tech. Rep. ATL-SOFT-PUB-2007-004, ATL-COM-SOFT-2007-009, 2007. [Online]. Available: https://cds.cern.ch/record/1038098.

[164] S. Agostinelli *et al.*, "Geant4 - a simulation toolkit", *Nucl. Instrum. Methods Phys. Res., Sect. A*, vol. 506, no. 3, pp. 250–303, 2003. DOI: 10.1016/S0168-9002(03)01368-8.

[165] ATLAS Collaboration, "Technical Design Report: A High-Granularity Timing Detector for the ATLAS Phase-II Upgrade", CERN, Geneva, Tech. Rep. CERN-LHCC-2020-007, ATLAS-TDR-031, 2020.

[166] P. Gessinger-Befurt, A. Salzburger, and J. Niermann, "The Open Data Detector Tracking System", *J. Phys. Conf. Ser.*, vol. 2438, no. 1, p. 012 110, Feb. 2023. DOI: 10.1088/1742-6596/2438/1/012110.

[167] A. Salzburger *et al.*, *OpenDataDetector*, gitlab, 2021. Accessed: Jul. 17, 2025. [Online]. Available: https://gitlab.cern.ch/acts/OpenDataDetector.

[168] M. Frank *et al.*, "DD4hep: A Detector Description Toolkit for High Energy Physics Experiments", *J. Phys. Conf. Ser.*, vol. 513, no. 2, p. 022 010, Jun. 2014. DOI: 10.1088/1742-6596/513/2/022010.

[169] The ACTS project, *Material assignment.* Accessed: Jul. 17, 2025. [Online]. Available: https://acts.readthedocs.io/en/latest/core/material.html#projective-approximation-of-passive-material.

[170] ATLAS Collaboration, "Fast Track Reconstruction for HL-LHC", CERN, Geneva, Tech. Rep., 2019. [Online]. Available: https://cds.cern.ch/record/2693670.

[171] ATLAS Collaboration, "ATLAS HL-LHC Computing Conceptual Design Report", CERN, Geneva, Tech. Rep. CERN-LHCC-2020-015, LHCC-G-178, 2020. [Online]. Available: https://cds.cern.ch/record/2729668.

[172] NVIDIA, *CUDA C++ programming guide*, Jun. 2025. Accessed: Jul. 17, 2025. [Online]. Available: https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[173] E. Lindholm *et al.*, "NVIDIA Tesla: A Unified Graphics and Computing Architecture", *IEEE Micro*, vol. 28, no. 2, pp. 39–55, 2008. DOI: 10.1109/MM.2008.31.

[174] M. Flynn, "Very high-speed computing systems", *Proceedings of the IEEE*, vol. 54, no. 12, pp. 1901–1909, 1966. DOI: 10.1109/PROC.1966.5273.

[175] J. Nickolls *et al.*, "Scalable Parallel Programming with CUDA: Is CUDA the Parallel Programming Model That Application Developers Have Been Waiting For?", *Queue*, vol. 6, no. 2, pp. 40–53, Mar. 2008, ISSN: 1542-7730. DOI: 10.1145/1365490.1365500.

[176] R. Reyes and V. Lomüller, "SYCL: Single-source C++ accelerator programming", in *Parallel Computing: On the Road to Exascale*, IOS Press, 2016, pp. 673–682. DOI: 10.3233/978-1-61499-621-7-673.

[177] X. Ai *et al.*, "A GPU-based Kalman filter for track fitting", *Computing and Software for Big Science*, vol. 5, no. 1, p. 20, Oct. 2021, ISSN: 2510-2044. DOI: 10.1007/s41781-021-00065-z.

[178] B. Yeo, *Implentation of CUDA for seed finding*, GitHub, 2020. Accessed: Jul. 17, 2025. [Online]. Available: https://github.com/acts-project/acts/pull/104.

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

[180] Y. He *et al.*, "Preparing NERSC users for Cori, a Cray XC40 system with Intel many integrated cores", *Concurrency and Computation: Practice and Experience*, vol. 30, Aug. 2017. DOI: 10.1002/cpe.4291.

[181] S. N. Swatman *et al.*, "Systematically Exploring High-Performance Representations of Vector Fields Through Compile-Time Composition", in *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '23, Coimbra, Portugal: Association for Computing Machinery, 2023, pp. 55–66, ISBN: 9798400700682. DOI: 10.1145/3578244.3583723. [Online]. Available: https://doi.org/10.1145/3578244.3583723.

[182] S. N. Swatman, A. Krasznahorkay, and P. Gessinger, *ACTS project - covfie: A library for storing interpolatable vector fields on co-processors*, GitHub, 2022. Accessed: Jul. 17, 2025. [Online]. Available: https://github.com/acts-project/covfie.

[183] J. Niermann *et al.*, *ACTS project - algebra-plugins*, GitHub, 2021. Accessed: Jul. 17, 2025. [Online]. Available: https://github.com/acts-project/algebra-plugins.

[184] T. Mazumder, "Acts - Vectorized Linear Algebra Implementation", Google Summer of Code, 2021.

[185] S. N. Swatman, A. Krasznahorkay, and P. Gessinger, "Managing heterogeneous device memory using c++17 memory resources", *J. Phys. Conf. Ser.*, vol. 2438, no. 1, p. 012 050, Feb. 2023. DOI: 10.1088/1742-6596/2438/1/012050. [Online]. Available: https://dx.doi.org/10.1088/1742-6596/2438/1/012050.

[186] A. Krasznahorkay *et al.*, *ACTS project - vecmem: Vectorised data model base and helper classes*, GitHub, 2020. Accessed: Jul. 17, 2025. [Online]. Available: https://github.com/acts-project/vecmem.

[187] M. Kiehn *et al.*, "The TrackML high-energy physics tracking challenge on Kaggle", *EPJ Web Conf.*, vol. 214, p. 06 037, 2019. DOI: 10.1051/epjconf/201921406037.

[188] N. Lohmann, *JSON for Modern C++*, https://json.nlohmann.me, 2015.

[189] A. Krasznahorkay *et al.*, online, WLCG/HSF Workshop 2024, 2022. Accessed: Jun. 12, 2025. [Online]. Available: https://indico.cern.ch/event/1369601/contributions/5898656/.

[190] ATLAS Collaboration, "Technical Design Report for the Phase-II Upgrade of the ATLAS TDAQ System", CERN, Geneva, Tech. Rep. CERN-LHCC-2017-020, ATLAS-TDR-029, 2017. DOI: 10.17181/CERN.2LBB.4IAL. [Online]. Available: https://cds.cern.ch/record/2285584.

[191] ATLAS Collaboration, "Technical Design Report for the Phase-II Upgrade of the ATLAS Trigger and Data Acquisition System - Event Filter Tracking Amendment", CERN, Geneva, Tech. Rep. CERN-LHCC-2022-004, ATLAS-TDR-029-ADD-1, 2022. DOI: 10.17181/CERN.ZK85.5TDL.

[192] R. Brun, A. Gheata, and M. Gheata, "The ROOT geometry package", *Nucl. Instrum. Methods Phys. Res., Sect. A*, vol. 502, no. 2, pp. 676–680, 2003. DOI: 10.1016/S0168-9002(03)00541-2.

[193] A. Salzburger *et al.*, *ACTS project - detray: Test library for detector surface intersection*, GitHub, v0.100.1, 2020. Accessed: Jul. 9, 2025. [Online]. Available: https://github.com/acts-project/detray.

[194] A. Salzburger *et al.*, "Detray: a compile time polymorphic tracking geometry description", *J. Phys. Conf. Ser.*, vol. 2438, p. 012 026, Feb. 2023. DOI: 10.1088/1742-6596/2438/1/012026.

[195] A. Salzburger, *detray - Test library for detector surface intersection*, GitHub, 2021. Accessed: Jul. 17, 2025. [Online]. Available: https://github.com/acts-project/detray/tree/b1dcab7ea6d195735390aa368797f800c783af0b.

[196] F. V. Barba, "A Plugin for Visualizing and Analyzing Detector Geometry and Track Propagation in Detray", CERN Summer Student Project, 2023.

[197] A. Salszburger *et al.*, *ACTS project - actsvg: An svg based c++17 plotting library for acts detectors and surfaces*, GitHub, 2022. Accessed: Jul. 17, 2025. [Online]. Available: https://github.com/acts-project/actsvg.

[198] GitHub, *What is CI/CD?*, 2024. Accessed: Jul. 17, 2025. [Online]. Available: https://github.com/resources/articles/devops/ci-cd.

[199] ISO, *ISO/IEC 14882:2017 Information technology — Programming languages — C++*, Fifth. pub-ISO, 2017. Accessed: Jul. 17, 2025. [Online]. Available: https://www.iso.org/standard/68564.html.

[200] Intel Corporation, *oneAPI DPC++ Compiler documentation*. Accessed: Jul. 17, 2025. [Online]. Available: https://intel.github.io/llvm-docs/index.html#.

[201] N. Bell and J. Hoberock, "Thrust: A Productivity-Oriented Library for CUDA", in *GPU Computing Gems Jade Edition*, ser. Applications of GPU Computing Series, W.-m. W. Hwu, Ed., Boston: Morgan Kaufmann, 2012, pp. 359–371, ISBN: 978-0-12-385963-1. DOI: 10.1016/B978-0-12-385963-1.00026-5.

[202] S. N. Swatman, *Implement detray::tuple*, GitHub, 2023. Accessed: Jun. 28, 2024. [Online]. Available: https://github.com/acts-project/detray/pull/415.

[203] V. Tsulaia, *New function for creating misaligned detector views*, GitHub, 2025. Accessed: Jul. 9, 2025. [Online]. Available: https://github.com/acts-project/detray/pull/894.

[204] B. Yeo, *Generic container*, GitHub, 2022. Accessed: Jul. 17, 2025. [Online]. Available: https://github.com/acts-project/detray/pull/255.

[205] B. Yeo, *Add line mask and its intersector*, GitHub, 2022. Accessed: Jul. 17, 2025. [Online]. Available: https://github.com/acts-project/detray/pull/270.

[206] J. H. Clark, "Hierarchical geometric models for visible surface algorithms", *Commun. ACM*, vol. 19, no. 10, pp. 547–554, Oct. 1976. DOI: 10.1145/360349.360354.

[207] S. M. Rubin and T. Whitted, "A 3-dimensional representation for fast rendering of complex scenes", *SIGGRAPH Comput. Graph.*, vol. 14, no. 3, pp. 110–116, Jul. 1980. DOI: 10.1145/965105.807479.

[208] J. L. Bentley, "Multidimensional binary search trees used for associative searching", *Commun. ACM*, vol. 18, no. 9, pp. 509–517, Sep. 1975. DOI: 10.1145/361002.361007.

[209] J. G. Cleary *et al.*, "Multiprocessor Ray Tracing", *Comput. Graph. Forum*, vol. 5, no. 1, pp. 3–12, 1986. DOI: 10.1111/J.1467-8659.1986.TB00263.X.

[210] A. Fujimoto, T. Tanaka, and K. Iwata, “Arts: Accelerated ray-tracing system”, *IEEE Comput. Graphics Appl.*, vol. 6, no. 4, pp. 16–26, 1986.

[211] J. Cleary and G. Wyvill, “Analysis of an Algorithm for Fast Ray Tracing using Uniform Space Subdivision”, *The Visual Computer*, vol. 4, pp. 65–83, Mar. 1988. DOI: 10.1007/BF01905559.

[212] ISO, *ISO/IEC 14882:2020 Information technology — Programming languages — C++*, Sixth. pub-ISO, 2020. Accessed: Jul. 17, 2025. [Online]. Available: https://www.iso.org/standard/79358.html.

[213] G. M. Morton, *A computer oriented geodetic data base and a new technique in file sequencing*. International Business Machines Company New York, 1966.

[214] T. Bially, “Space-filling curves: Their generation and their application to bandwidth reduction”, *IEEE Trans. Inf. Theory*, vol. 15, pp. 658–664, 1969. [Online]. Available: https://api.semanticscholar.org/CorpusID:30122901.

[215] E. Niebler, *Range-v3*. Accessed: Jul. 17, 2025. [Online]. Available: https://github.com/ericniebler/range-v3.

[216] E. Niebler, S. Parent, and A. Sutton, *N4128: Ranges for the standard library, revision 1*, 2014. Accessed: Jul. 17, 2025. [Online]. Available: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4128.html.

[217] *The GNU C++ Standard Library*. Accessed: Jul. 17, 2024. [Online]. Available: https://github.com/gcc-mirror/gcc/tree/master/libstdc%2B%2B-v3.

[218] S. Brand, *P2374R1 views::cartesian_product*, Apr. 23, 2021. Accessed: Jul. 17, 2025. [Online]. Available: http://wg21.link/P2374.

[219] A. Salzburger, “The ATLAS Track Extrapolation Package”, Jun. 2007.

[220] Graphviz, *DOT Language*. Accessed: Jul. 17, 2025. [Online]. Available: https://graphviz.org/doc/info/lang.html.

[221] B. Yeo, Private Communication, 2022.

[222] J. Berman *et al.*, *python-jsonschema/jsonschema*, version v4.21.1, Jan. 2024. DOI: 10.5281/zenodo.10535924.

[223] E. Xochelli, *feat: Detray plugin geometry*, GitHub, 2024. Accessed: Jul. 10, 2025. [Online]. Available: https://github.com/acts-project/acts/pull/3299.

[224] A. Salzburger and P. Gessinger, “Next generation geometry model for Tracking in ACTS”, CHEP 2024 - Conference on Computing in High Energy and Nuclear Physics, 2024. Accessed: Jul. 10, 2025. [Online]. Available: https://indico.cern.ch/event/1410793/contributions/5930054/attachments/2872109/5028693/traccc%20Development%20Update%202024.06.06..pdf.

[225] E. Gamma *et al.*, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed. Addison-Wesley Professional, 1994, ISBN: 0201633612.

[226] B. Yeo, *Add RK stepper*, GitHub, 2022. Accessed: Jul. 17, 2025. [Online]. Available: https://github.com/acts-project/detray/pull/204.

[227] The ACTS Project, *CylinderSurface*, GitHub. Accessed: Jul. 17, 2025. [Online]. Available: https://github.com/acts-project/acts/blob/main/Core/include/Acts/Surfaces/CylinderSurface.hpp.

[228] W. H. Press *et al.*, *Numerical Recipes 3rd Edition: The Art of Scientific Computing*, version 3.04. USA: Cambridge University Press, 2011, ISBN: 978-0-521-88068-8.

[229] C. R. Harris *et al.*, "Array programming with NumPy", *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. DOI: 10.1038/s41586-020-2649-2.

[230] The pandas development team, *Pandas-dev/pandas: Pandas*, version 2.2.2, Feb. 2024. DOI: 10.5281/zenodo.3509134. Accessed: Jul. 17, 2025. [Online]. Available: https://doi.org/10.5281/zenodo.3509134.

[231] R. Gommers *et al.*, *Scipy/scipy: Scipy 1.13.1*, version v1.13.1, May 2024. DOI: 10.5281/zenodo.11255513. Accessed: Jul. 17, 2025. [Online]. Available: https://doi.org/10.5281/zenodo.11255513.

[232] J. D. Hunter, "Matplotlib: A 2D graphics environment", *Comput Sci Eng*, vol. 9, no. 3, pp. 90–95, 2007. DOI: 10.1109/MCSE.2007.55.

[233] The Matplotlib Development Team, *Matplotlib: Visualization with Python*, version v3.9.0, May 2024. DOI: 10.5281/zenodo.11201097. Accessed: Jul. 17, 2025.

[234] G. Civil *et al.*, *Googletest 1.14.0*, version v1.14.0, 2023. Accessed: Jul. 17, 2025. [Online]. Available: https://github.com/google/googletest.

[235] M. Bandieramonte *et al.*, "The GeoModel tool suite for detector description", *EPJ Web of Conferences*, vol. 251, p. 03 007, Jan. 2021. DOI: 10.1051/epjconf/202125103007.

[236] ATLAS Collaboration, *Athena*, version 22.0.1, Apr. 2019. DOI: 10.5281/zenodo.2641997. [Online]. Available: https://doi.org/10.5281/zenodo.2641997.

[237] A. Salzburger, *adding json writing, reading infrastructure*, GitHub, 2023. Accessed: Jul. 17, 2025. [Online]. Available: https://github.com/acts-project/acts/pull/2283.

[238] The ACTS Project, *ACTS material averaging*, GitHub, ACTS v42.0.0. Accessed: Jul. 8, 2025. [Online]. Available: https://github.com/acts-project/acts/blob/main/Core/src/Material/AverageMaterials.cpp.

[239] M. Paterno, "Calculating efficiencies and their uncertainties", Tech. Rep. FERMILAB-TM-2286-CD, Dec. 2004. DOI: 10.2172/15017262.

[240] T. Ullrich and Z. Xu, *Treatment of Errors in Efficiency Calculations*, 2007. arXiv: physics/0701199v1.

[241] D. Casadei, "Estimating the selection efficiency", *Journal of Instrumentation*, vol. 7, no. 08, P08021–P08021, Aug. 2012, ISSN: 1748-0221. DOI: 10.1088/1748-0221/7/08/p08021. [Online]. Available: http://dx.doi.org/10.1088/1748-0221/7/08/P08021.

[242] D. Goldberg, "What every computer scientist should know about floating-point arithmetic", *ACM Comput. Surv.*, vol. 23, no. 1, pp. 5–48, Mar. 1991, ISSN: 0360-0300. DOI: 10.1145/103162.103163.

[243] B. Yeo, *ref: More realistic toy detector benchmarks*, Private Communications, 2024. Accessed: Jul. 6, 2025. [Online]. Available: https://github.com/acts-project/detray/pull/885.

[244] R. Menon and L. Dagum, "OpenMP: An Industry-Standard API for Shared-Memory Programming", *Comput Sci Eng*, vol. v, no. 01, pp. 46–55, Jan. 1998. DOI: 10.1109/99.660313.

[245] *google benchmark*, 2025. Accessed: Jul. 6, 2025. [Online]. Available: https://github.com/google/benchmark.

[246] Free Software Foundation, Inc., *gcc 13.2*, 2025. Accessed: Jul. 5, 2025. [Online]. Available: https://gcc.gnu.org/onlinedocs/13.2.0/.

[247] NVIDIA, *CUDA Toolkit 12.6.2*, 2025. Accessed: Jul. 5, 2025. [Online]. Available: https://docs.nvidia.com/cuda/archive/12.6.2/.

[248] Red Hat, *Red Hat Enterprise Linux 9.5*, 2025. Accessed: Jul. 5, 2025. [Online]. Available: https://docs.redhat.com/de/documentation/red_hat_enterprise_linux/9/html/9.5_release_notes/index.

[249] AMD, *AMD EPYC 7413*, 2025. Accessed: Jul. 5, 2025. [Online]. Available: https://www.amd.com/en/products/processors/server/epyc/7003-series/amd-epyc-7413.html#product-specs.

[250] NVIDIA, *RTX A5000*, 2025. Accessed: Jul. 5, 2025. [Online]. Available: https://resources.nvidia.com/en-us-briefcase-for-datasheets/nvidia-rtx-a5000-dat-1?ncid=no-ncid.

[251] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: maximizing on-chip parallelism", *SIGARCH Comput. Archit. News*, vol. 23, no. 2, pp. 392–403, May 1995. DOI: 10.1145/225830.224449.

[252] S. Swatman, *perf: Optimize propagation benchmark kernel launch*, Private Communications, 2023. Accessed: Jul. 6, 2025. [Online]. Available: https://github.com/acts-project/detray/pull/414.

[253] M. Bauer, H. Cook, and B. Khailany, "CudaDMA: optimizing GPU memory bandwidth via warp specialization", in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11, Seattle, Washington: Association for Computing Machinery, 2011, ISBN: 9781450307710. DOI: 10.1145/2063384.2063400.

[254] K. Gupta, J. A. Stuart, and J. D. Owens, "A study of Persistent Threads style GPU programming for GPGPU workloads", in *2012 Innovative Parallel Computing (InPar)*, 2012, pp. 1–14. DOI: 10.1109/InPar.2012.6339596.

[255] A. Krasznahorkay, “traccc Development Update”, ATLAS Software & Computing Week #78, 2024. Accessed: Jul. 17, 2025. [Online]. Available: https://indico.cern.ch/event/1410793/contributions/5930054/attachments/2872109/5028693/traccc%20Development%20Update%202024.06.06..pdf.

[256] NVIDIA Corporation, *NVIDIA Grace CPU*, 2024. Accessed: Jul. 17, 2025. [Online]. Available: https://www.nvidia.com/en-us/data-center/grace-cpu/.

[257] M. Kretz and V. Lindenstruth, “Vc: A C++ library for explicit vectorization”, *Software: Practice and Experience*, vol. 42, 2012. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1002/spe.1149.

[258] M. Kretz *et al.*, *Vcdevel/vc: Vc 1.4.5*, version 1.4.5, Jun. 2024. DOI: 10.5281/zenodo.11501874.

Appendices

APPENDIX A

Track State Generators

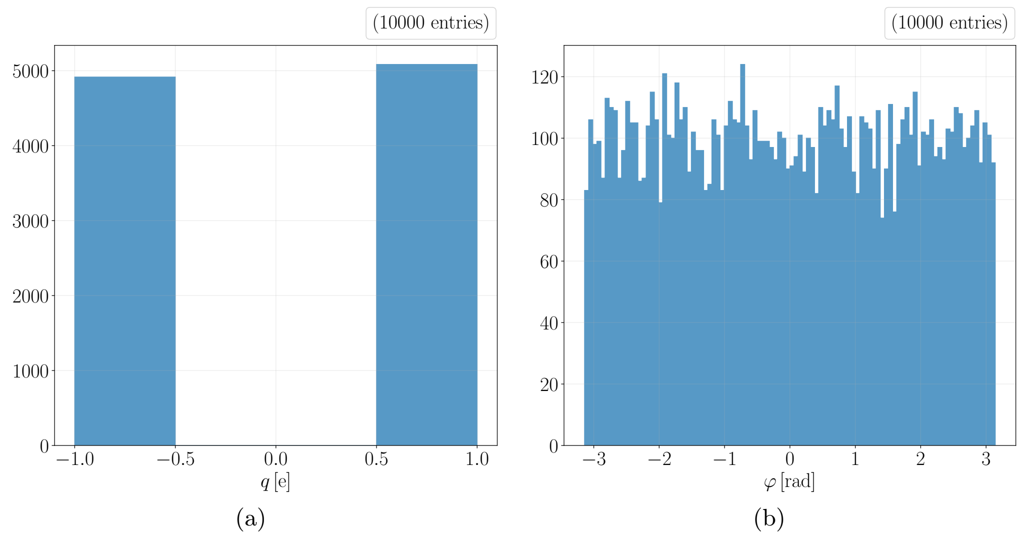


Figure A.1.: Distribution of q and ϕ of helices with $p_T = 0.5\,\mathrm{GeV}$, $|\eta| \leq 4$ and randomized charge, generated by the `detray::random_track_generator`.

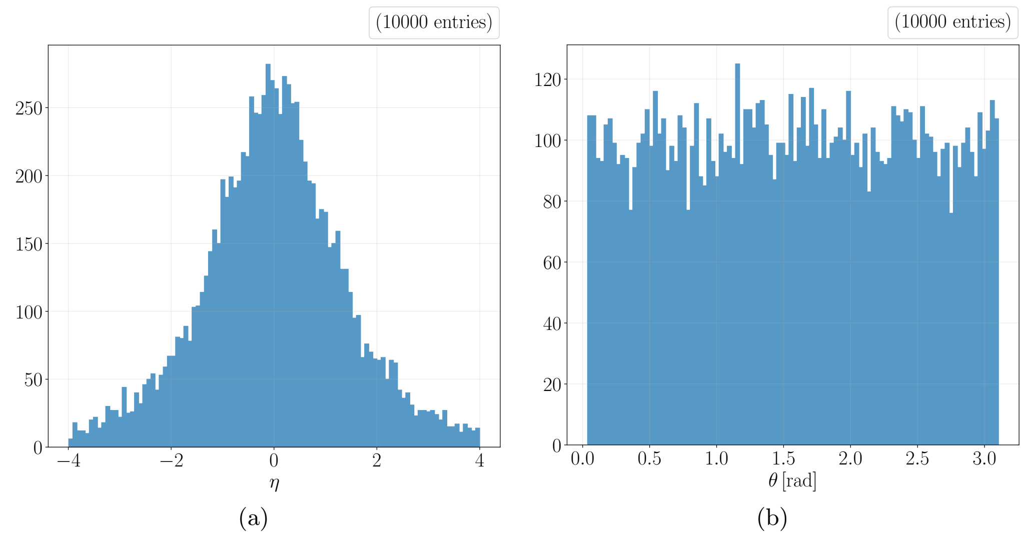


Figure A.2.: Distribution of η and θ of helices with $p_T = 0.5\,\text{GeV}$, $|\eta| \leq 4$ and randomized charge, generated by the `detray::random_track_generator`.

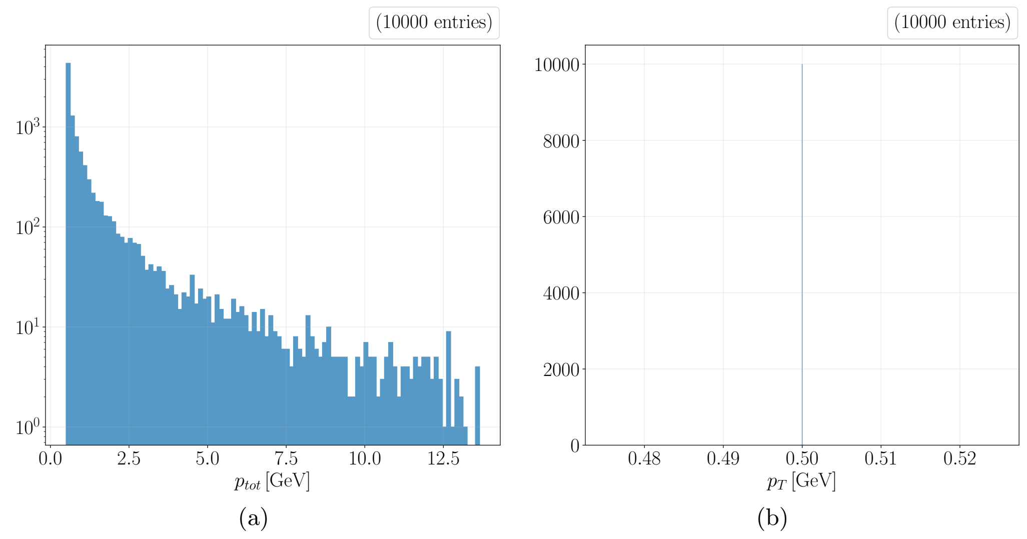


Figure A.3.: Distribution of $|\mathbf{p}|$ and p_T of helices with $p_T = 0.5\,\text{GeV}$, $|\eta| \leq 4$ and randomized charge, generated by the `detray::random_track_generator`.

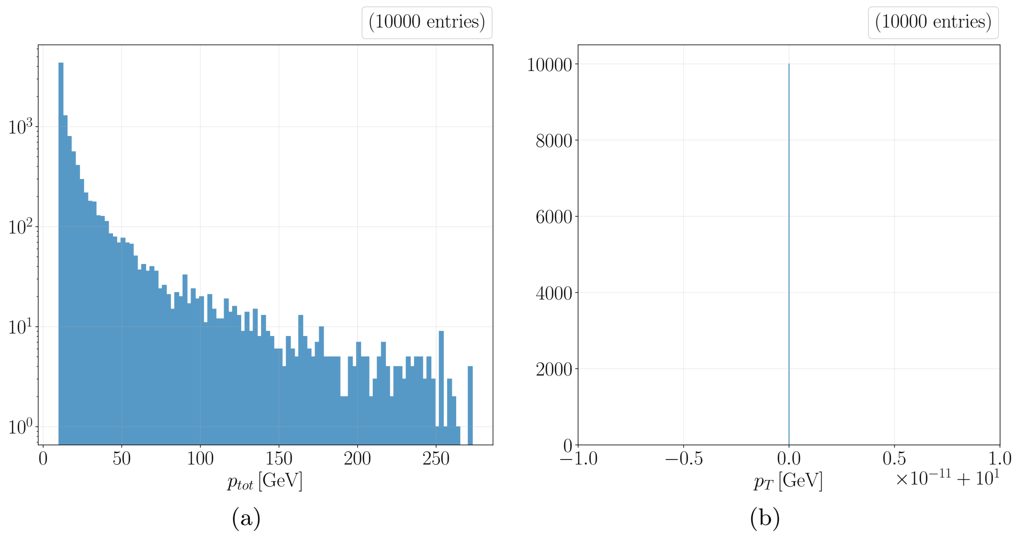


Figure A.4.: Distribution of $|\mathbf{p}|$ and p_T of helices with $p_T = 10\,\text{GeV}$, $|\eta| \leq 4$ and randomized charge, generated by the `detray::random_track_generator`.

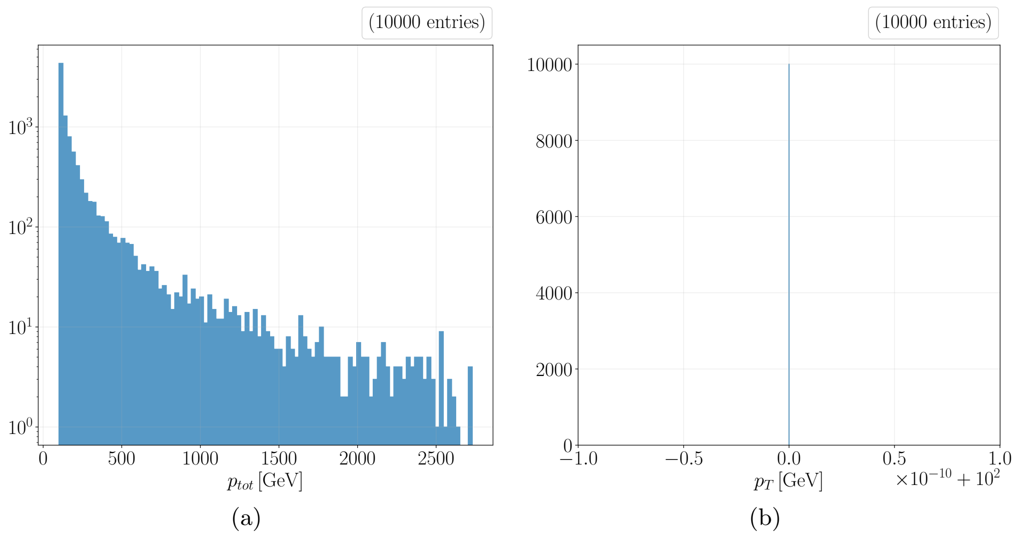


Figure A.5.: Distribution of $|\mathbf{p}|$ and p_T of helices with $p_T = 100\,\text{GeV}$, $|\eta| \leq 4$ and randomized charge, generated by the `detray::random_track_generator`.

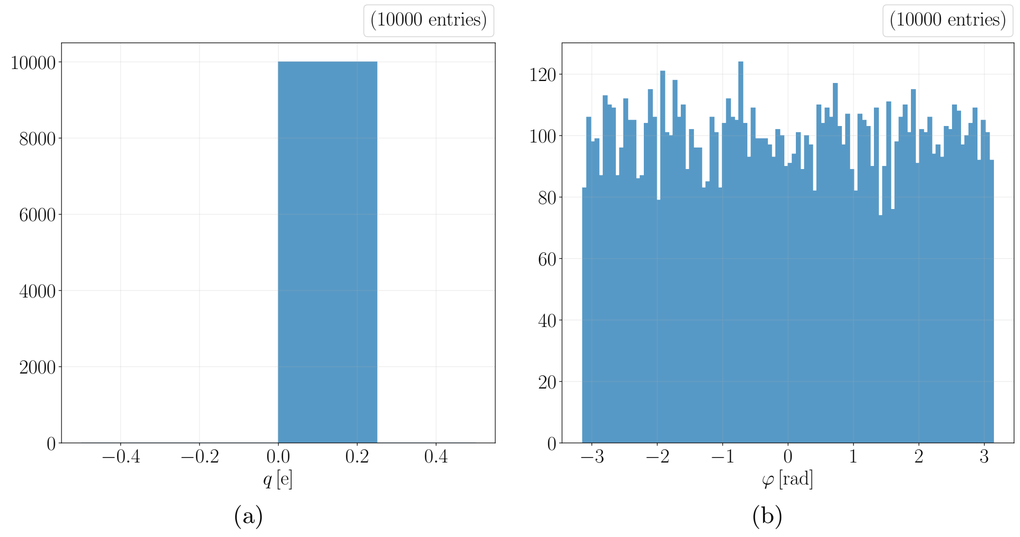


Figure A.6.: Distribution of q and ϕ of rays with $|\eta| \leq 4$, generated by the `detray::random_track_generator`.

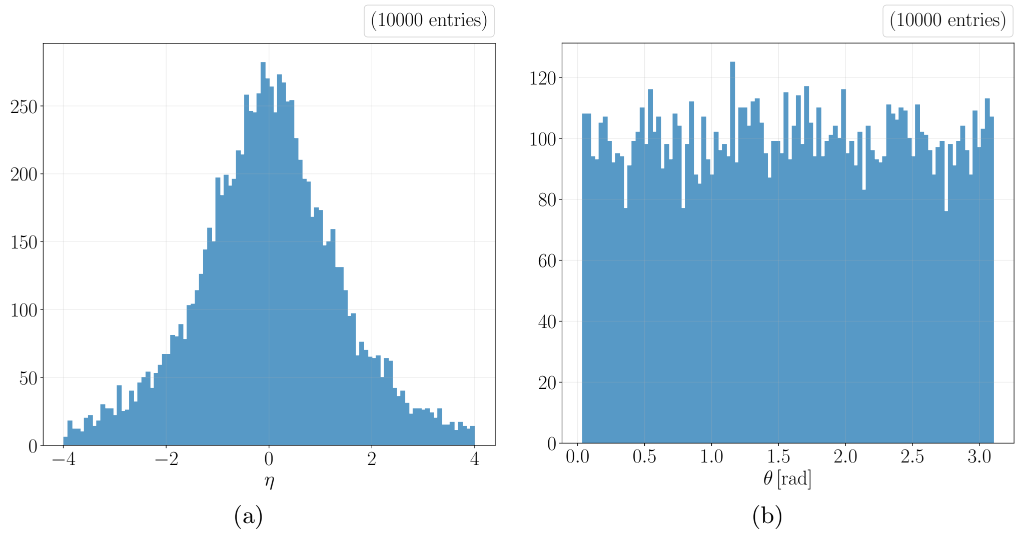


Figure A.7.: Distribution of η and θ of rays with $|\eta| \leq 4$, generated by the `detray::random_track_generator`.

APPENDIX B

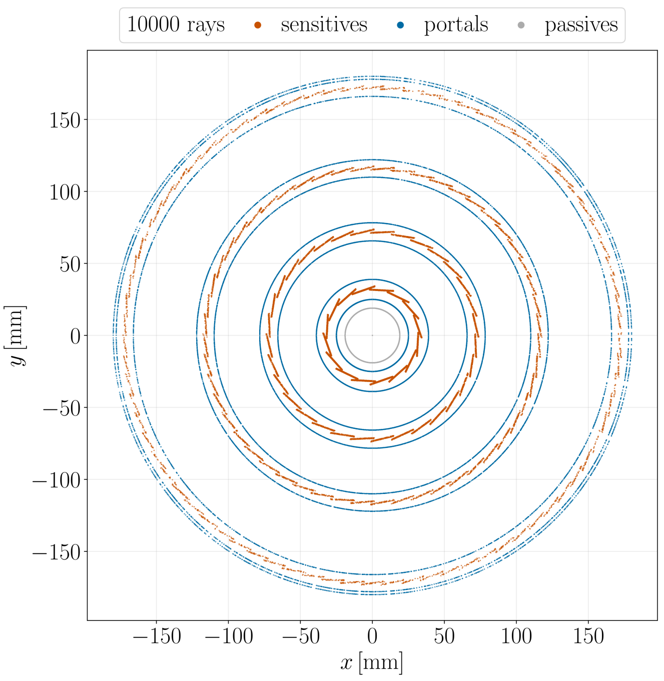
Ray and Helix Scans

B.1. Motivation for the slow Convergence Criterion

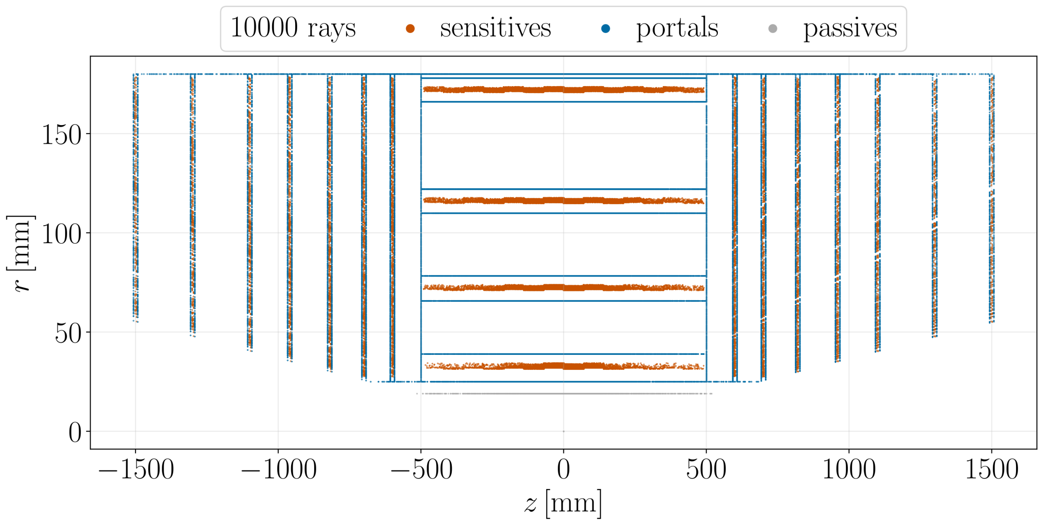
Take a Newton-Raphson step, whenever that seems to get towards the root slower than a bisection step. This assumes, that the iteration is already sufficiently close to the root and the step sizes in the iteration are small:

$$\begin{aligned}
\left|f(s_{n+1}^{newton})\right| &> \left|f(s_{n+1}^{bisection})\right| \\
\left|f'(s_n)\cdot \mathrm{d}s_n^{newton} + f(s_n)\right| &> \left|f'(s_n)\cdot \mathrm{d}s_n^{bisection} + f(s_n)\right| \\
\left|f'(s_n)\cdot \mathrm{d}s_n^{newton}\right| + \left|f(s_n)\right| &> \left|f'(s_n)\cdot \mathrm{d}s_n^{bisection} + f(s_n)\right| \\
\left|f'(s_n)\cdot\left(-\frac{f(s_n)}{f'(s_n)}\right)\right| + \left|f(s_n)\right| &> \left|f'(s_n)\cdot\left(\frac{1}{2}\left(a_n+b_n\right)-s_n\right)+f(s_n)\right| \\
\left|2\,f(s_n)\right| &> \left|f'(s_n)\cdot\left(\frac{1}{2}\left(a_n+b_n\right)-s_n\right)+f(s_n)\right| ,
\end{aligned} \tag{B.1}$$

with a_n and b_n being the bracket boundaries at step n. This criterion is only invalidated in the implementation of the algorithm, if the last step size ($|s_n - s_{n-1}|$) is below 10 mm.

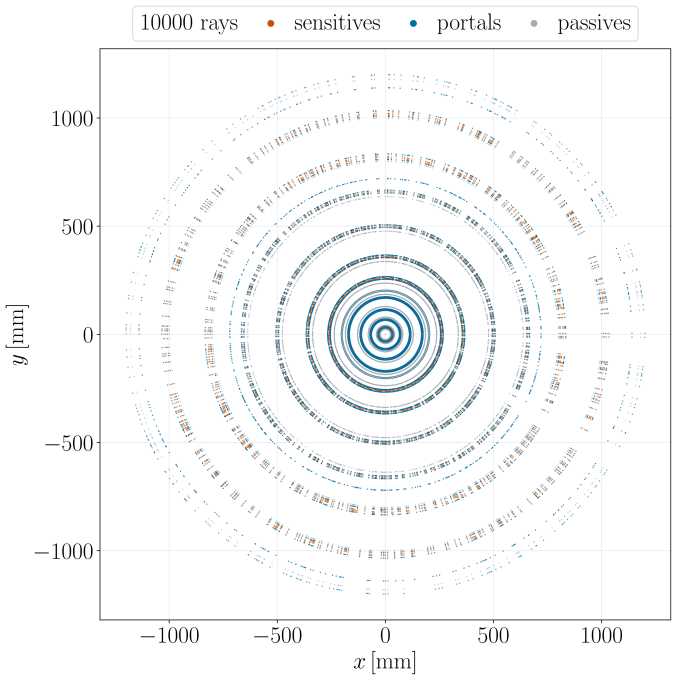


(a) xy-view (z-range of ± 50 mm)

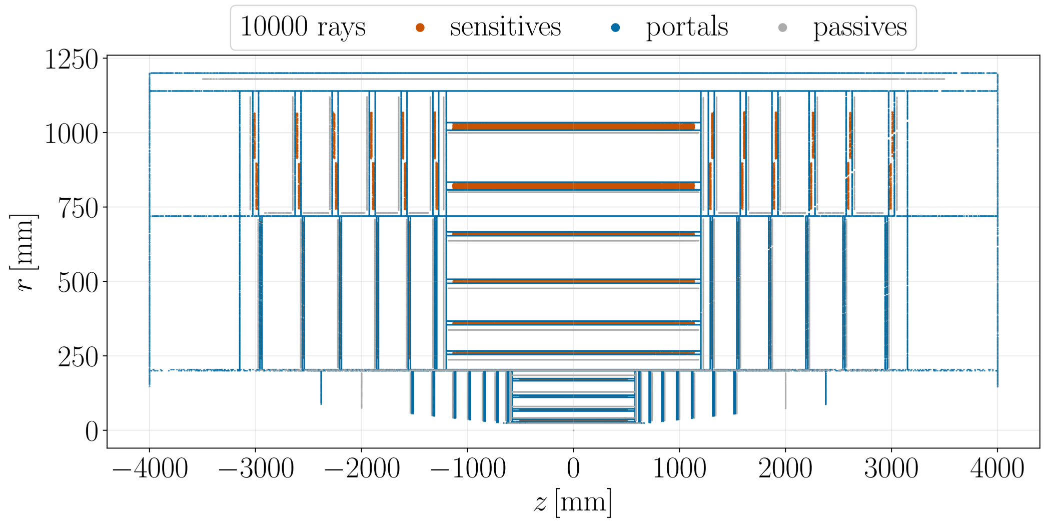


(b) rz-view

Figure B.1.: Ray scan of a toy detector geometry using rays with $|\eta| \leq 4$, generated by the `detray::random_track_generator`. It shows the intersection points of the rays with the detector surfaces, colored by the surface type.

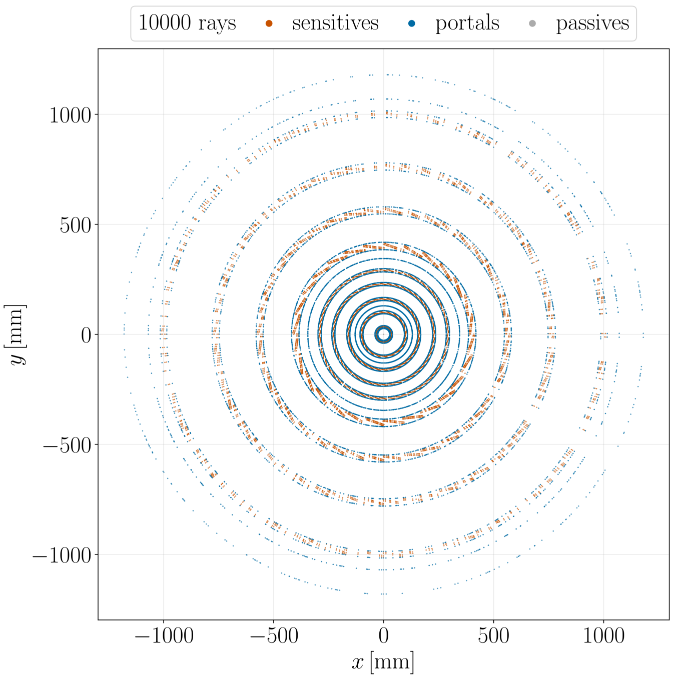


(a) xy-view (z-range of $\pm 50\,\mathrm{mm}$)

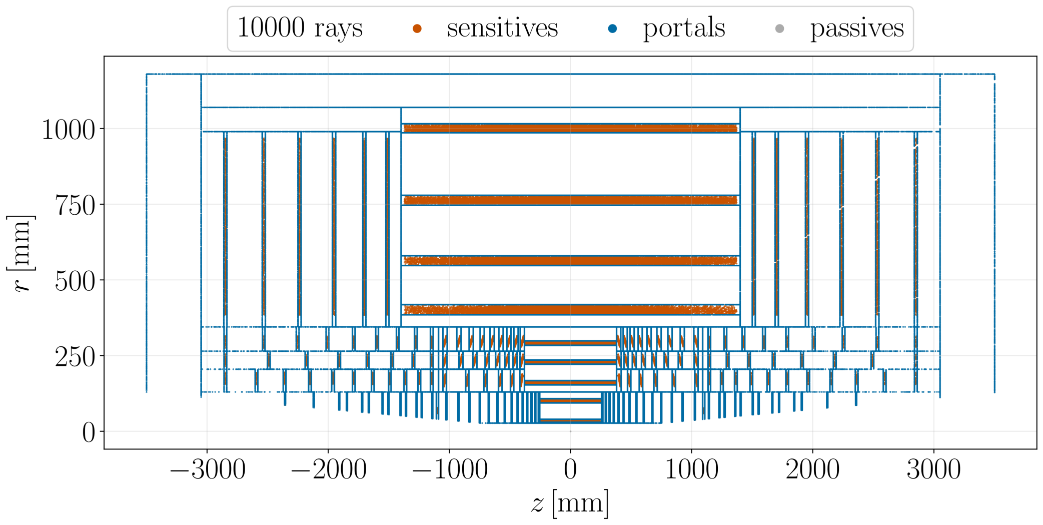


(b) rz-view

Figure B.2.: Ray scan of the ODD geometry using rays with $|\eta| \leq 4$, generated by the `detray::random_track_generator`. It shows the intersection points of the rays with the detector surfaces, colored by the surface type.



(a) xy-view (z-range of $\pm 50\,\mathrm{mm}$)



(b) rz-view

Figure B.3.: Ray scan of the ITk geometry using rays with $|\eta| \leq 4$, generated by the `detray::random_track_generator`. It shows the intersection points of the rays with the detector surfaces, colored by the surface type.

APPENDIX C

Comparison with the Navigator

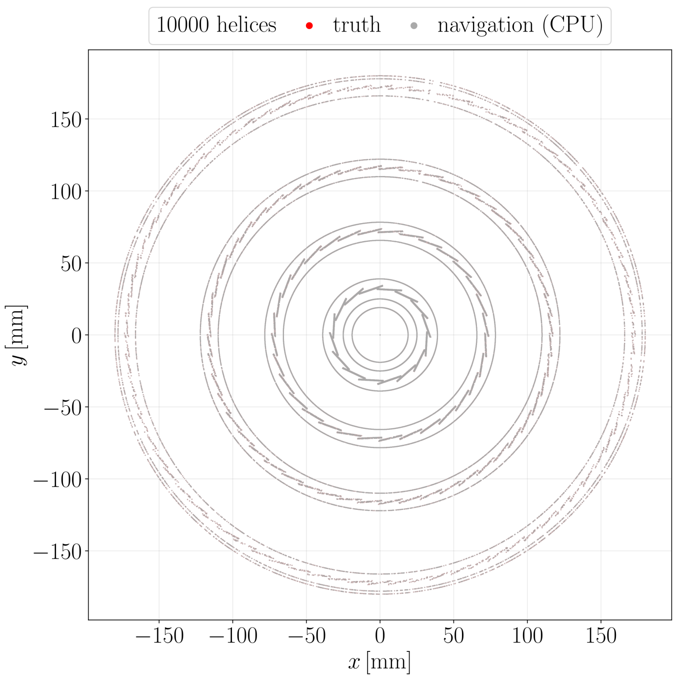
Host					
Detector	$\mathbf{p_T}$ [GeV]	**miss.**	**add.**	**total**	$\epsilon_{\mathbf{surfaces}}$
Toy Detector	0.5	0	0	279644	100.000 ± 0.0004
Toy Detector	10	0	0	279644	100.000 ± 0.0004
Toy Detector	100	0	0	279644	100.000 ± 0.0004
ODD	0.5	0	0	820295	100.000 ± 0.0002
ODD	10	0	0	820295	100.000 ± 0.0002
ODD	100	0	0	820295	100.000 ± 0.0002
ITk	0.5	0	0	771905	100.000 ± 0.0002
ITk	10	0	0	771905	100.000 ± 0.0002
ITk	100	0	0	771905	100.000 ± 0.0002
Device (CUDA)					
Toy Detector	0.5	0	0	279644	100.000 ± 0.0004
Toy Detector	10	0	0	279644	100.000 ± 0.0004
Toy Detector	100	0	0	279644	100.000 ± 0.0004
ODD	0.5	0	0	820295	100.000 ± 0.0002
ODD	10	0	0	820295	100.000 ± 0.0002
ODD	100	0	0	820295	100.000 ± 0.0002
ITk	0.5	0	0	771905	100.000 ± 0.0002
ITk	10	0	0	771905	100.000 ± 0.0002
ITk	100	0	0	771905	100.000 ± 0.0002

Table C.1.: Surface finding efficiency of the DETRAY navigator on host and device (CUDA), compared to a ray scan with 10 000 rays. The errors on the efficiency is a statistical error obtained by a Bayesian approach [239–241].

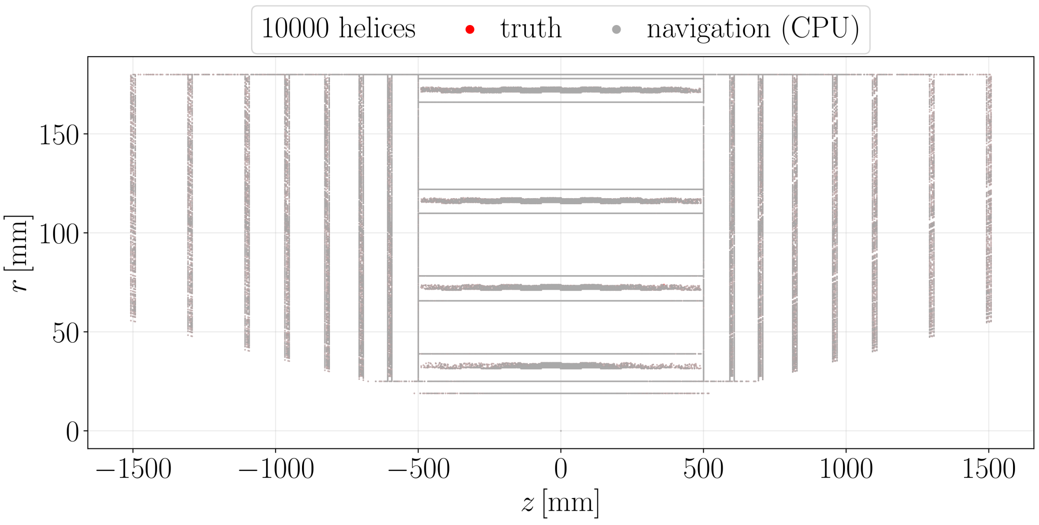
Option	**value**
Track generator	
Number of tracks	10 000
Abs. charge	1 e
Rand. charge	`true`
η range	-4, 4
Origin	(0, 0, 0)
Vertex smearing	`false`
Random seed	5489
Navigation	
Minimum mask tolerance	$1 \cdot 10^{-5}$ mm
Maximum mask tolerance	3 mm
Mask tolerance scale factor	0.05
Path tolerance	1 μm
Overstep tolerance	$-300\,\mu$m
Parameter Transport	
Minimum Step size	$1 \cdot 10^{-4}$ mm
Runge-Kutta tolerance	$1 \cdot 10^{-4}$ mm
Maximum step updates	10 000
Step size constraint	$3.40282 \cdot 10^{38}$ mm
Path limit	5 m

Table C.2.: Configuration for the navigation validation. For the toy detector the grid search window size was 3×3, for the other two detectors it was 0×0, as the grids exported from ACTS apply neighbourhood packing.

C.1. Toy Detector Track Position Comparison

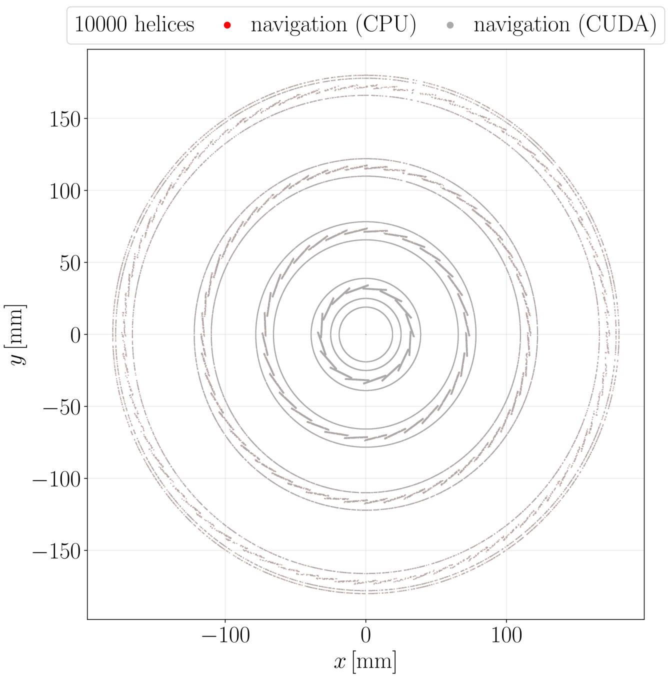


(a) xy-view (z-range of $\pm 50\,\mathrm{mm}$)

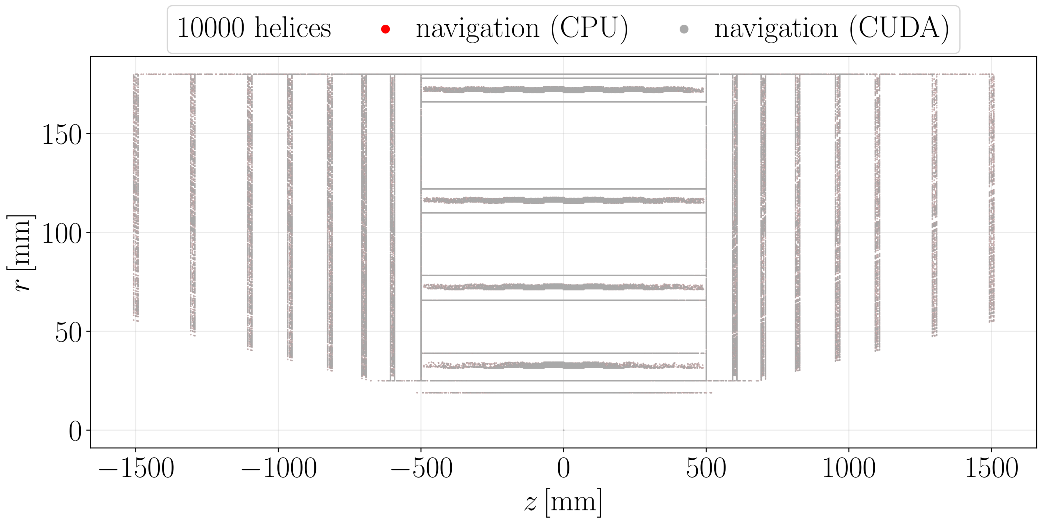


(b) rz-view (entire ϕ-range overlayed)

Figure C.1.: Overlay of track positions found during a helix scan on the host (red) and surface intersection trace recorded during navigation on the host (grey) for the toy detector.



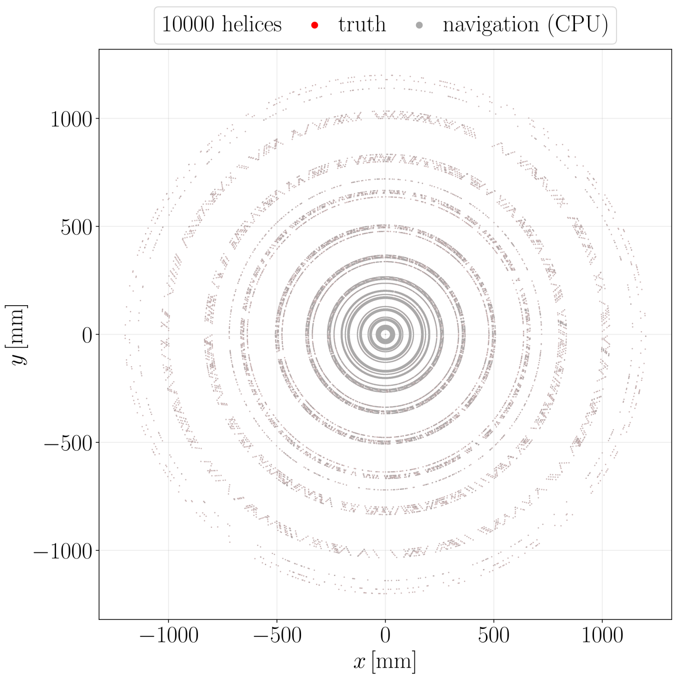
(a) xy-view (z-range of $\pm 50\,\text{mm}$)



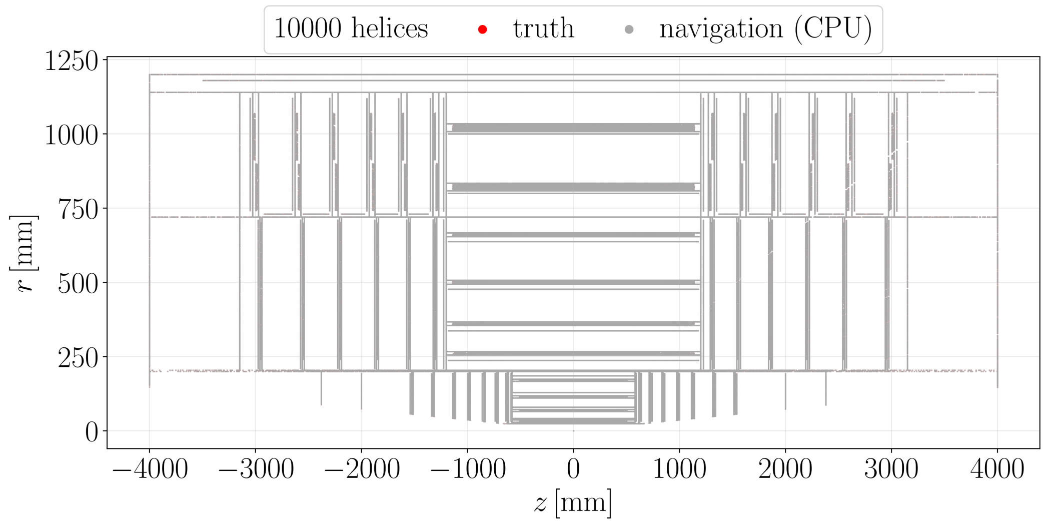
(b) rz-view (entire ϕ-range overlayed)

Figure C.2.: Overlay of track positions found during navigation on the host (red) and track positions recorded during navigation on the device (grey) for the toy detector.

C.2. ODD Track Position Comparison

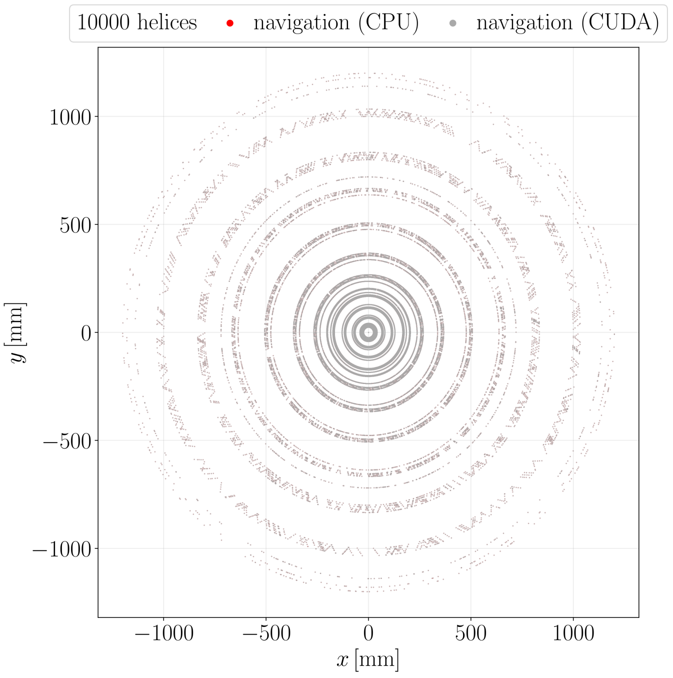


(a) xy-view (z-range of $\pm 50\,\mathrm{mm}$)

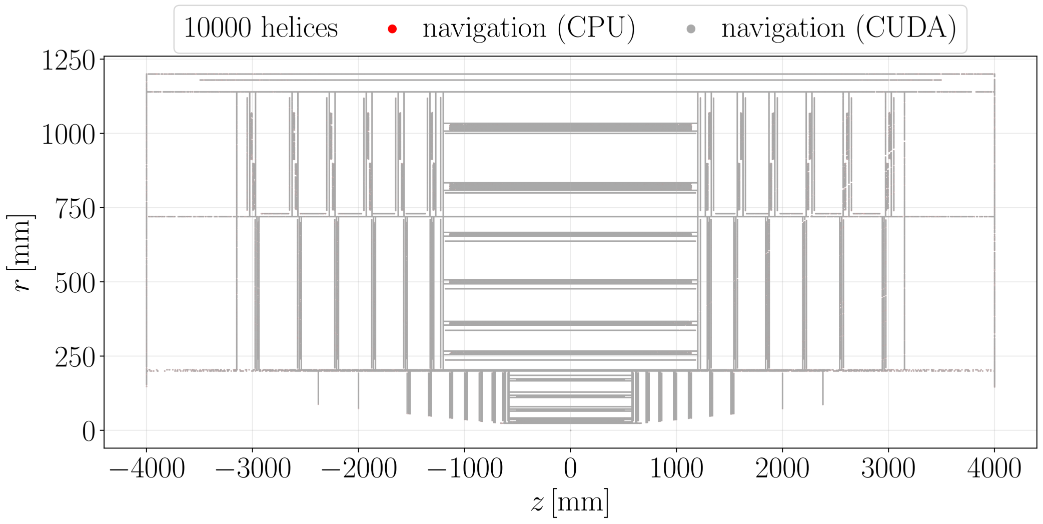


(b) rz-view (entire ϕ-range overlayed)

Figure C.3.: Overlay of track positions found during a helix scan on the host (red) and surface intersection trace recorded during navigation on the host (grey) for the ODD.



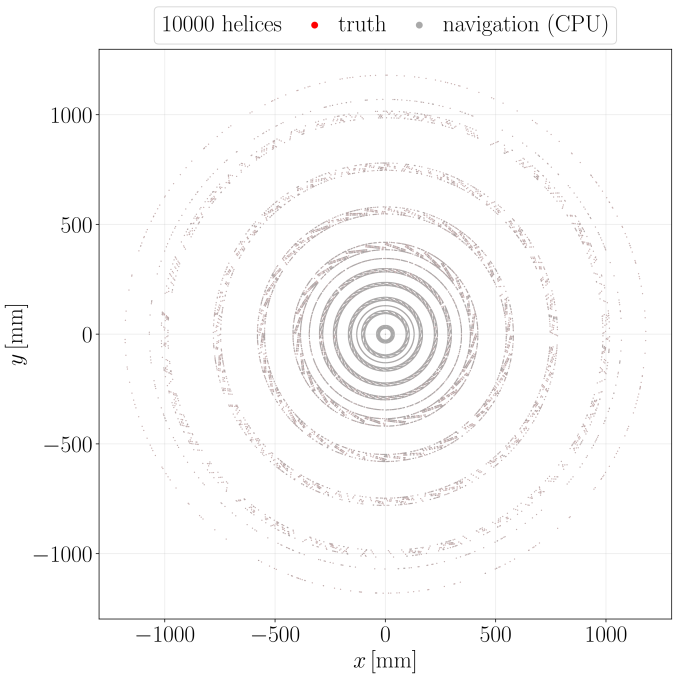
(a) xy-view (z-range of $\pm 50\,\mathrm{mm}$)



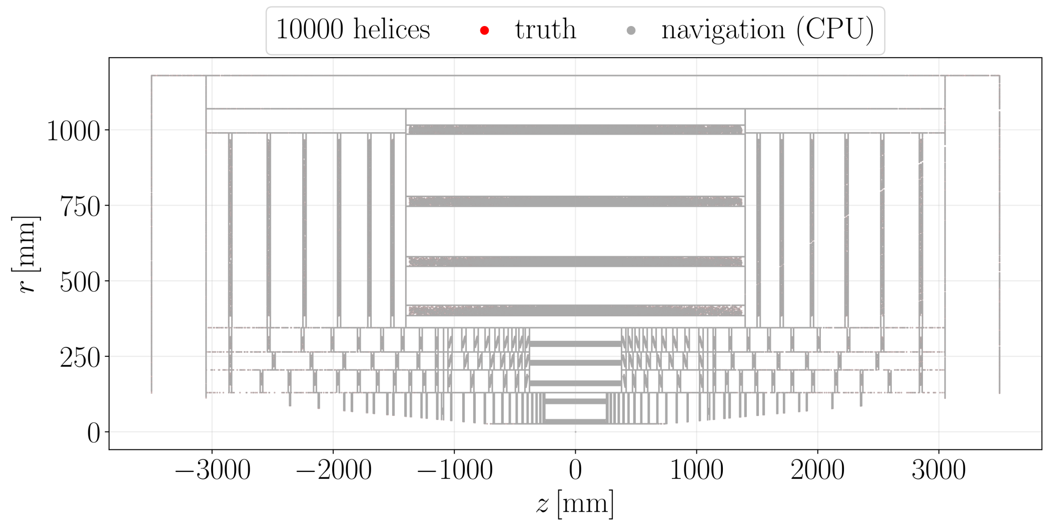
(b) rz-view (entire ϕ-range overlayed)

Figure C.4.: Overlay of track positions found during navigation on the host (red) and track positions recorded during navigation on the device (grey) for the ODD.

C.3. ITk Track Position Comparison

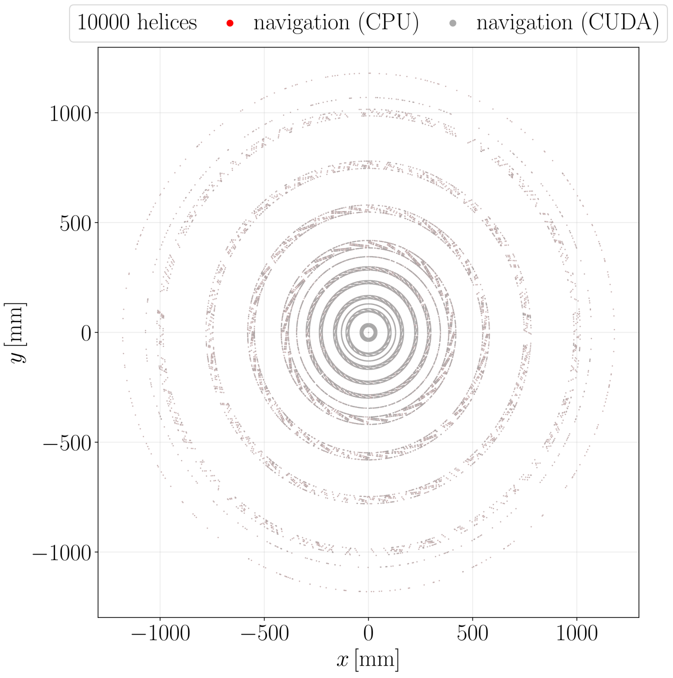


(a) xy-view (z-range of $\pm 50\,\text{mm}$)

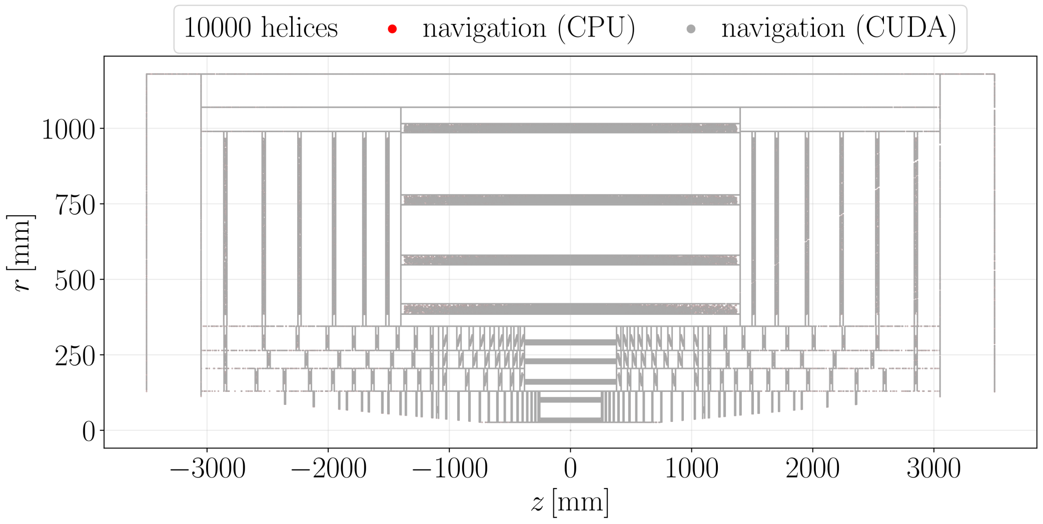


(b) rz-view (entire ϕ-range overlayed)

Figure C.5.: Overlay of track positions found during a helix scan on the host (red) and surface intersection trace recorded during navigation on the host (grey) for the ITk.



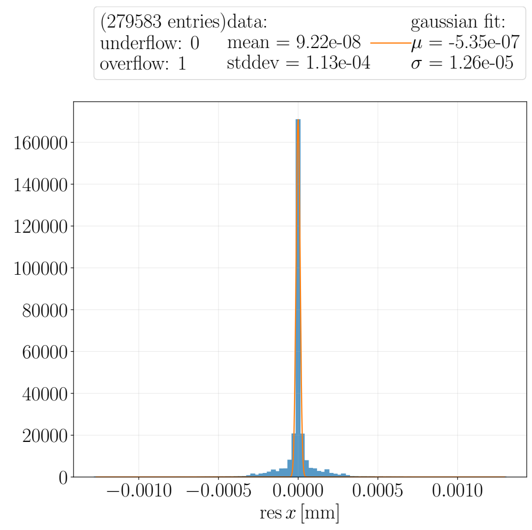
(a) xy-view (z-range of $\pm 50\,\mathrm{mm}$)



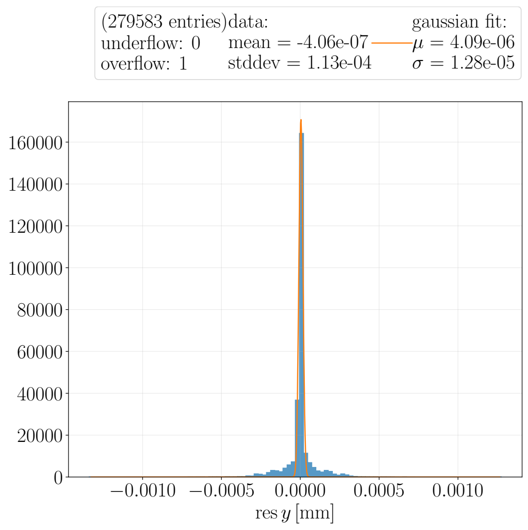
(b) rz-view (entire ϕ-range overlayed)

Figure C.6.: Overlay of track positions found during navigation on the host (red) and track positions recorded during navigation on the device (grey) for the ITk.

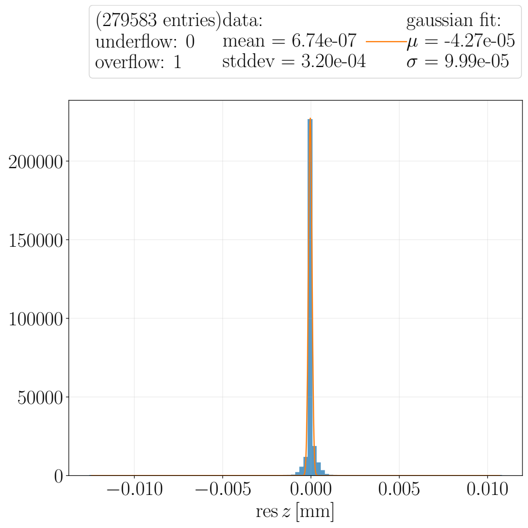
C.4. Toy Detector Track Position Residuals



(a) Residual x

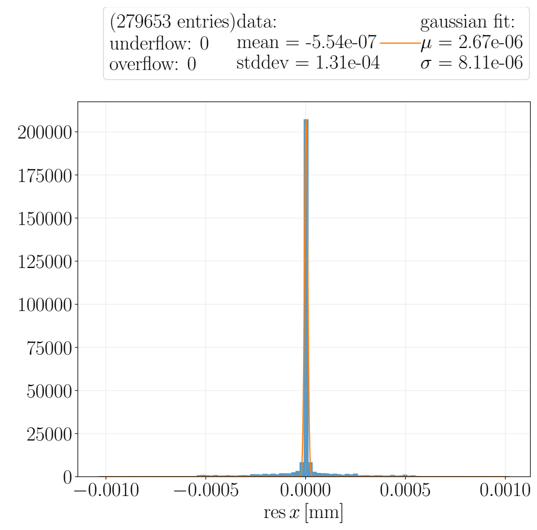


(b) Residual y

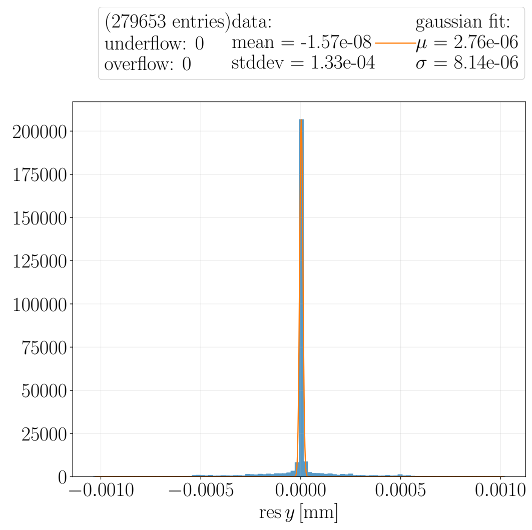


(c) Residual z

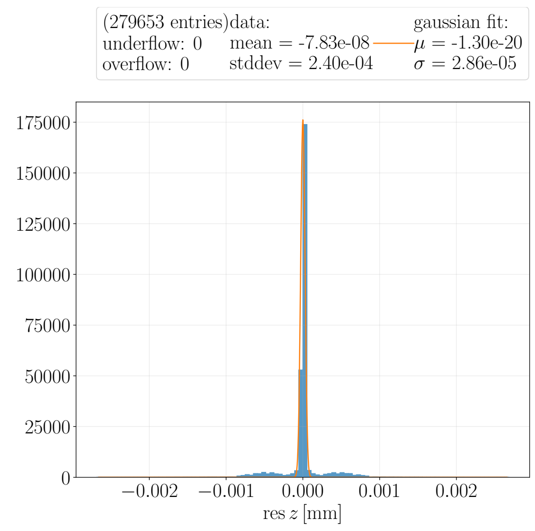
Figure C.7.: Track position residuals between the helix scan and device navigation traces for the toy detector with $p_T = 10\text{GeV}$.



(a) Residual x

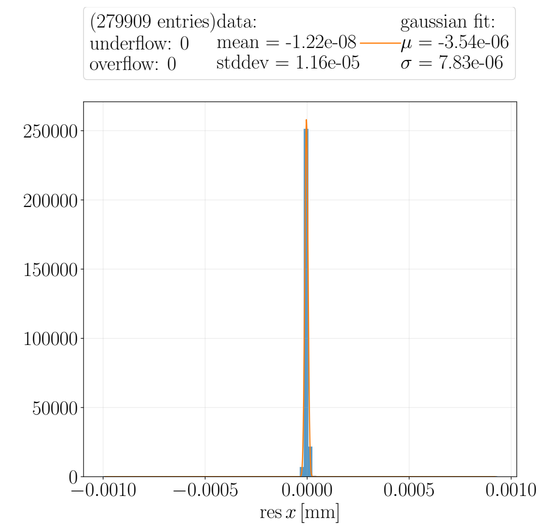


(b) Residual y

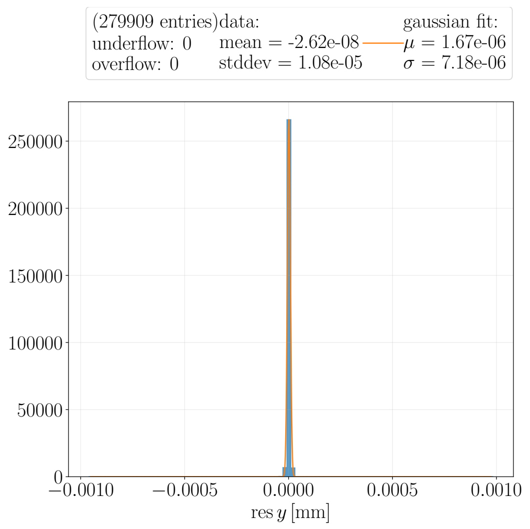


(c) Residual z

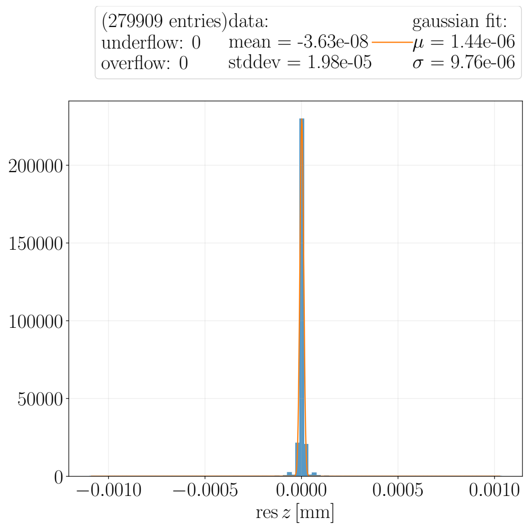
Figure C.8.: Track position residuals between the helix scan and device navigation traces for the toy detector with $p_T = 100\text{GeV}$.



(a) Residual x

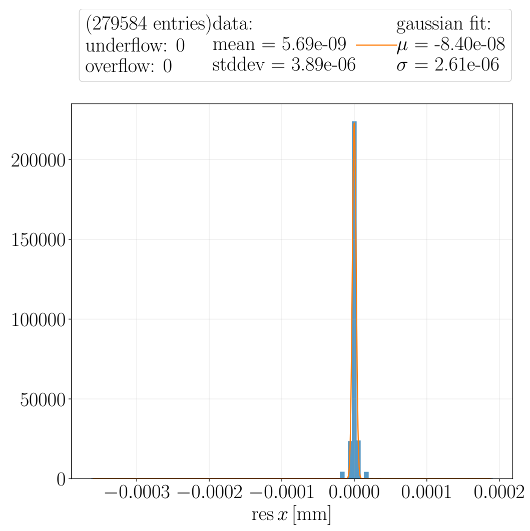


(b) Residual y

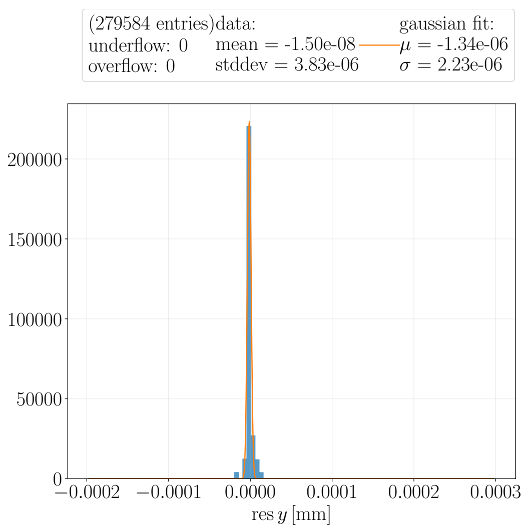


(c) Residual z

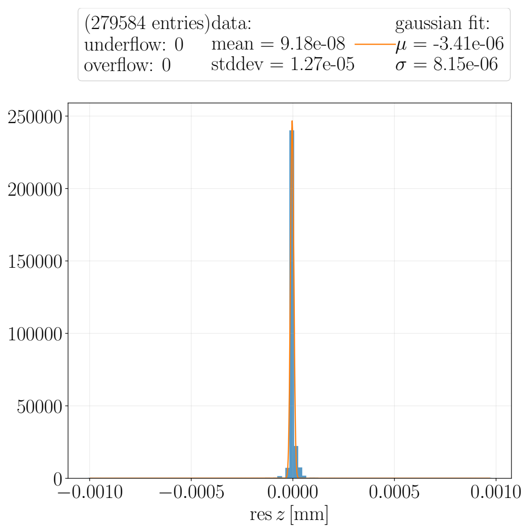
Figure C.9.: Track position residuals between the host and device navigation traces for the toy detector with $p_T = 0.5$GeV.



(a) Residual x

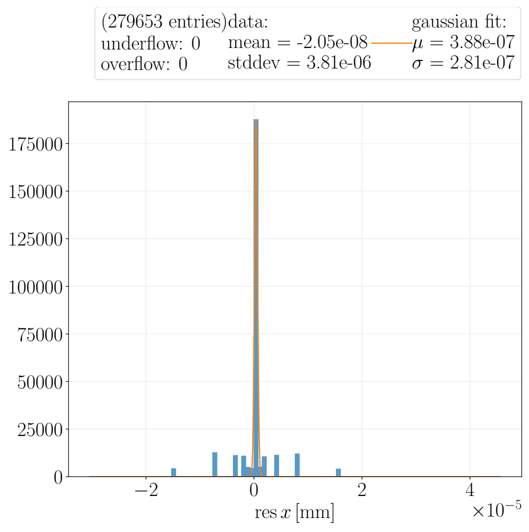


(b) Residual y

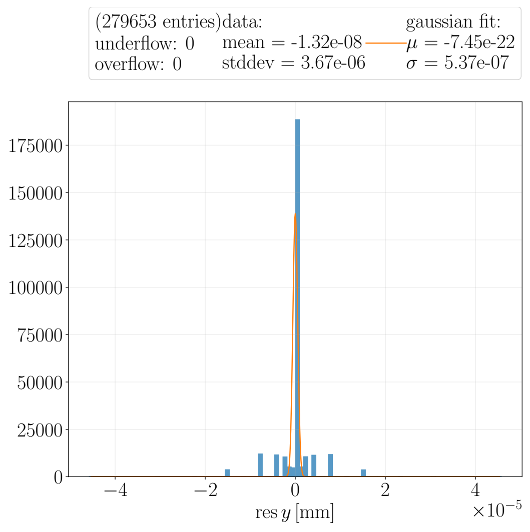


(c) Residual z

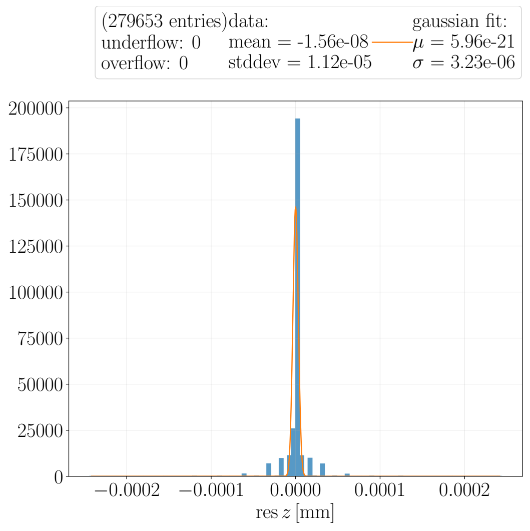
Figure C.10.: Track position residuals between the host and device navigation traces for the toy detector with $p_T = 10\text{GeV}$.



(a) Residual x



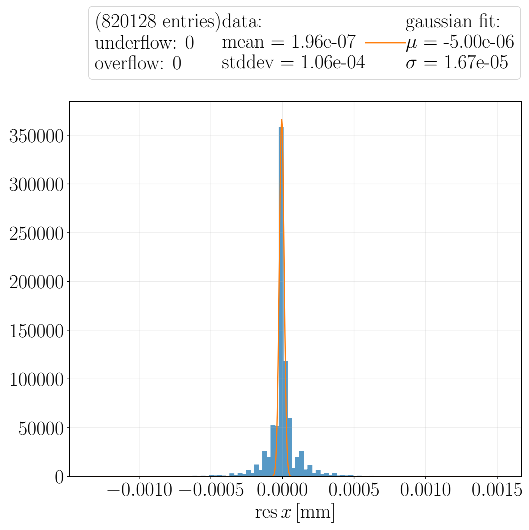
(b) Residual y



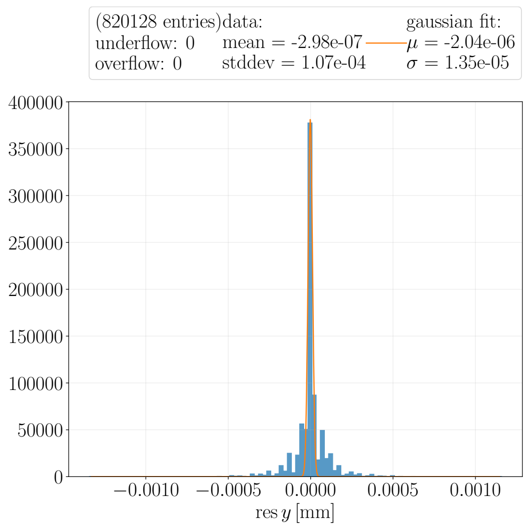
(c) Residual z

Figure C.11.: Track position residuals between the host and device navigation traces for the toy detector with $p_T = 100\text{GeV}$.

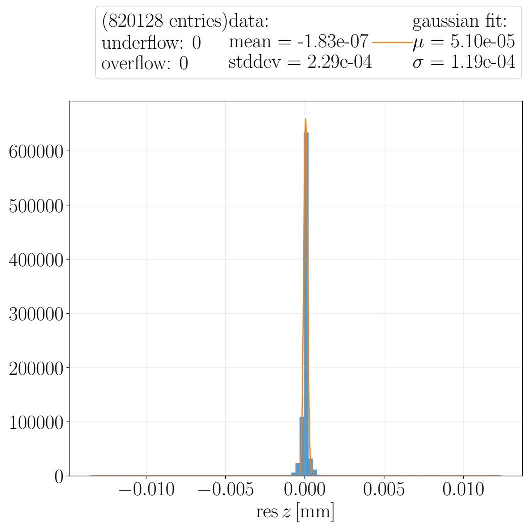
C.5. ODD Track Position Residuals



(a) Residual x

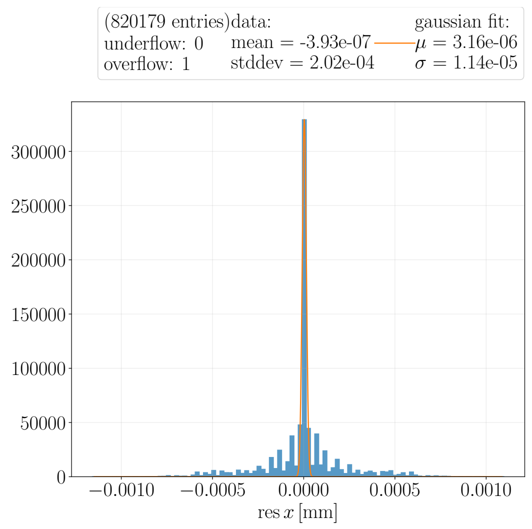


(b) Residual y

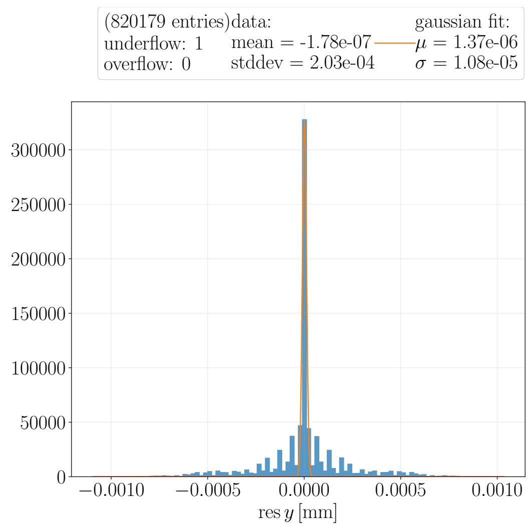


(c) Residual z

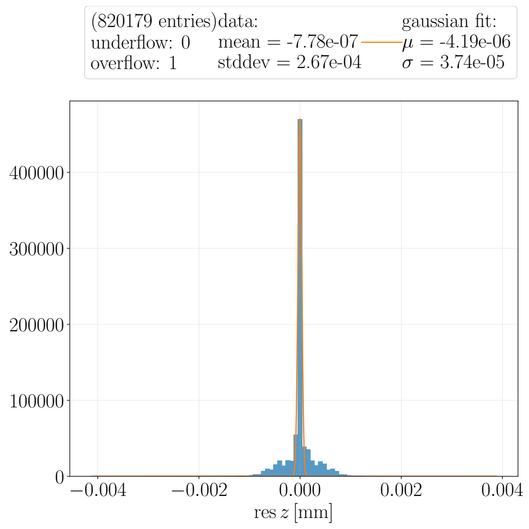
Figure C.12.: Track position residuals between the helix scan and device navigation traces for the ODD with $p_T = 10\text{GeV}$.



(a) Residual x

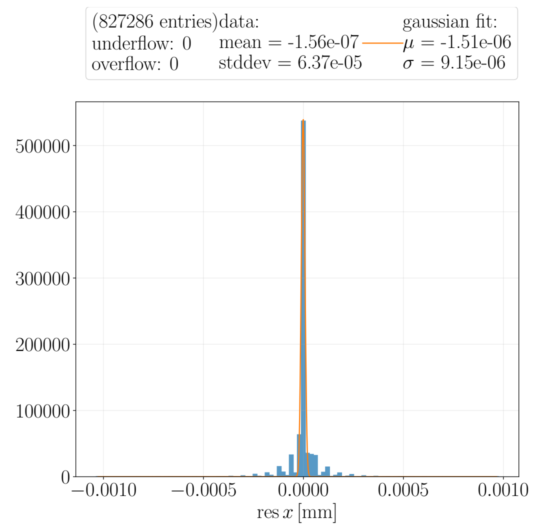


(b) Residual y

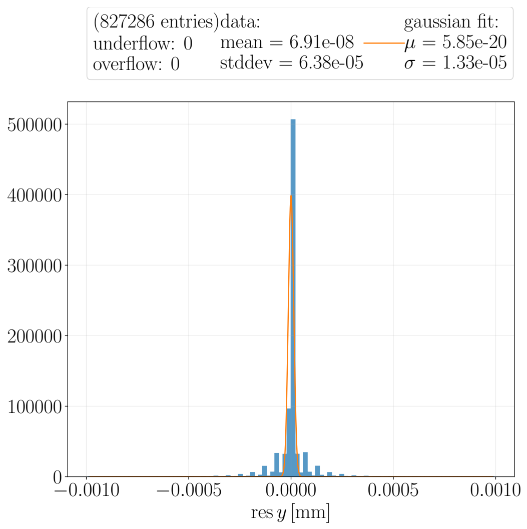


(c) Residual z

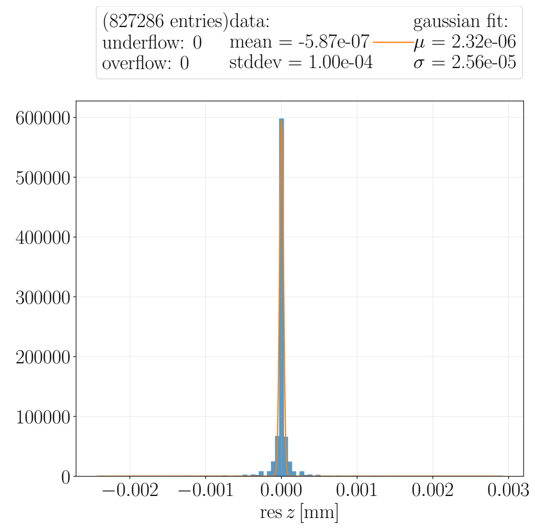
Figure C.13.: Track position residuals between the helix scan and device navigation traces for the ODD with $p_T = 100\text{GeV}$.



(a) Residual x

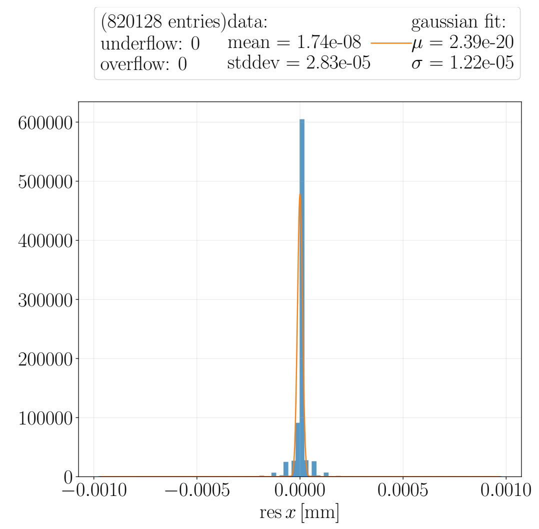


(b) Residual y

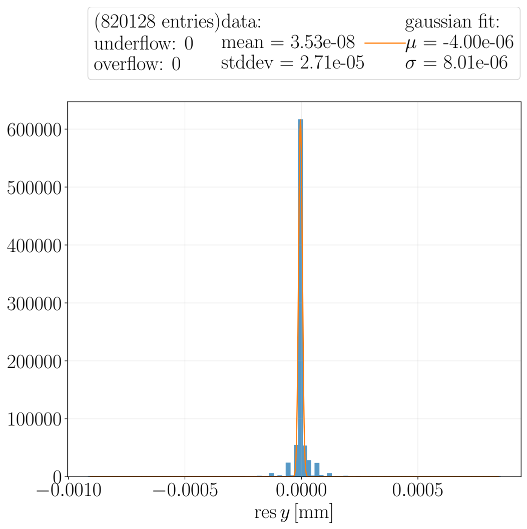


(c) Residual z

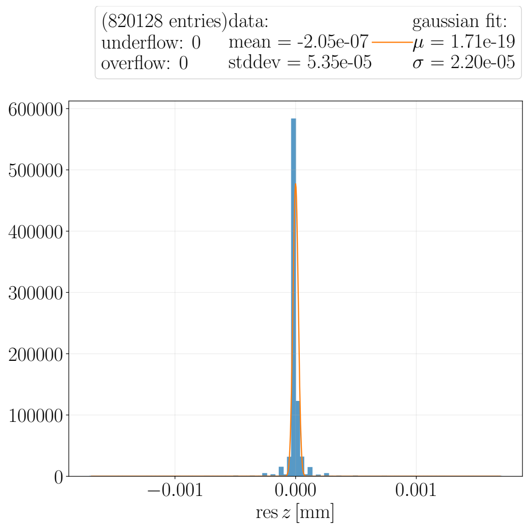
Figure C.14.: Track position residuals between the host and device navigation traces for the ODD with $p_T = 0.5\text{GeV}$.



(a) Residual x

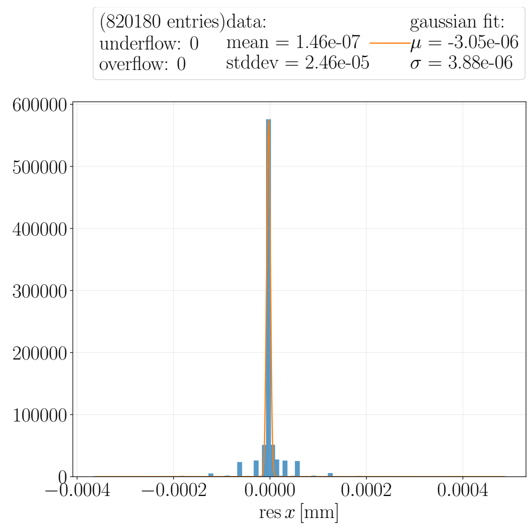


(b) Residual y

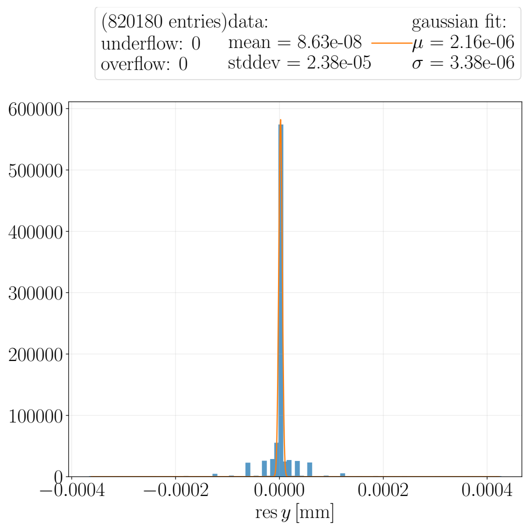


(c) Residual z

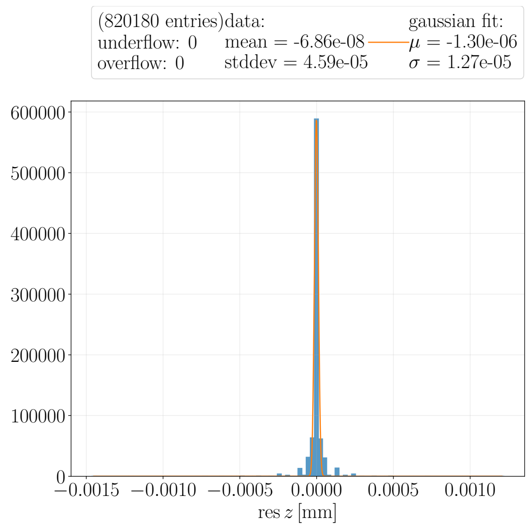
Figure C.15.: Track position residuals between the host and device navigation traces for the ODD with $p_T = 10\text{GeV}$.



(a) Residual x



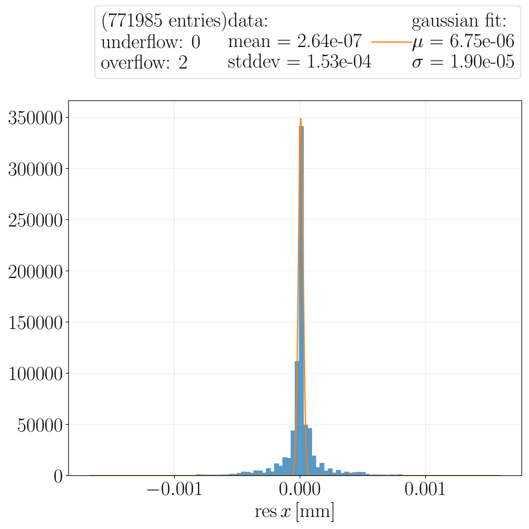
(b) Residual y



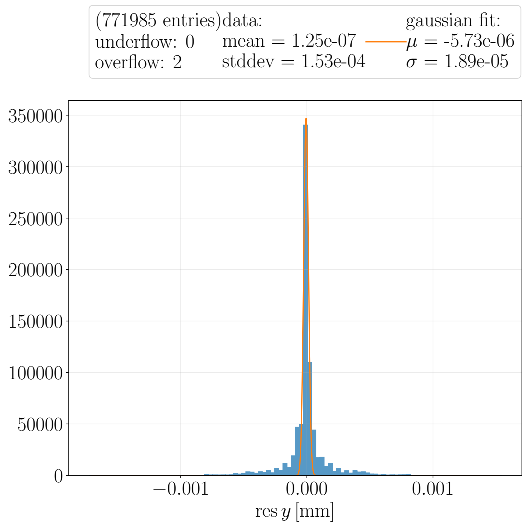
(c) Residual z

Figure C.16.: Track position residuals between the host and device navigation traces for the ODD with $p_T = 100\text{GeV}$.

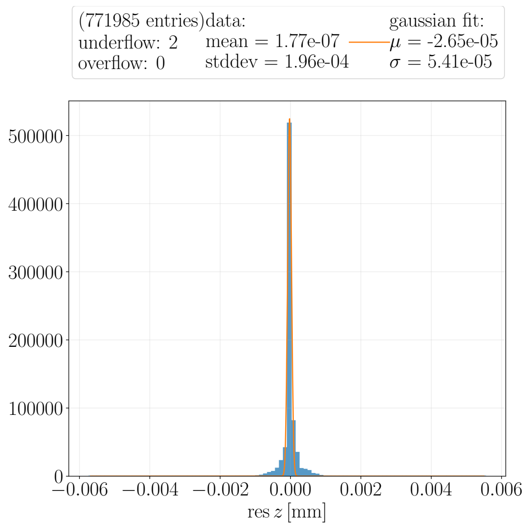
C.6. ITk Track Position Residuals



(a) Residual x

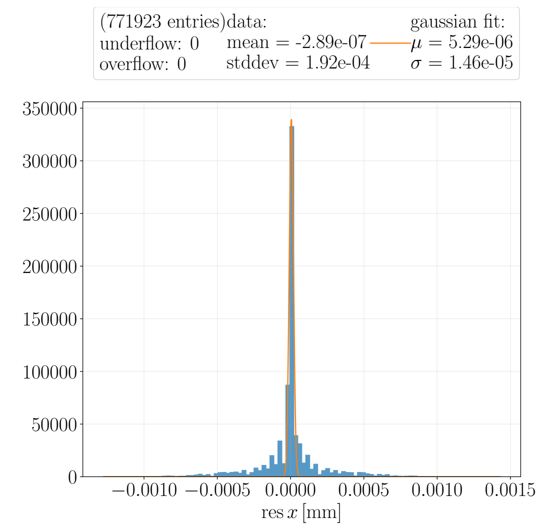


(b) Residual y

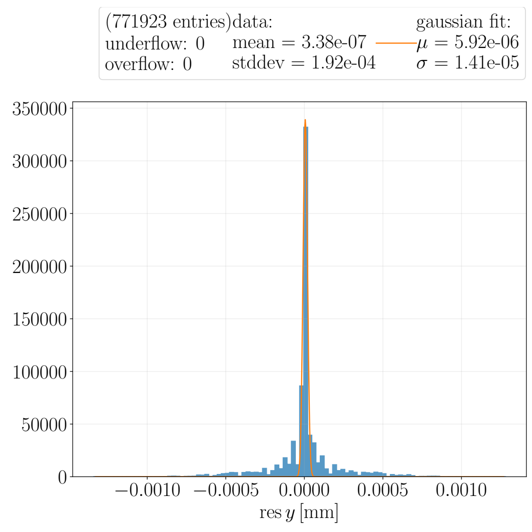


(c) Residual z

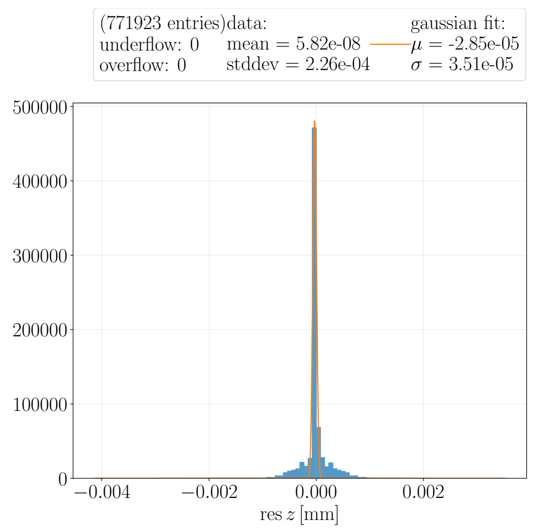
Figure C.17.: Track position residuals between the helix scan and device navigation traces for the ITk with $p_T = 10\text{GeV}$.



(a) Residual x

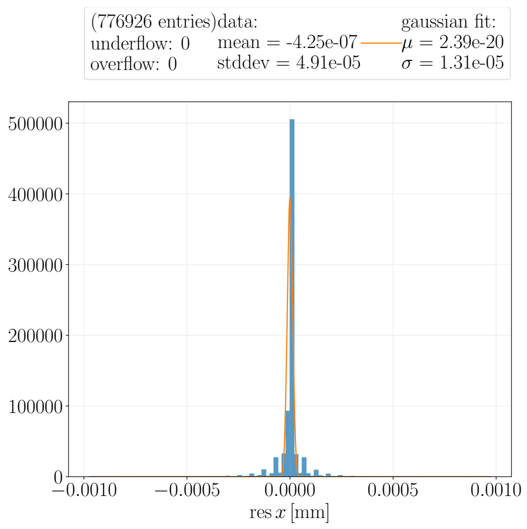


(b) Residual y

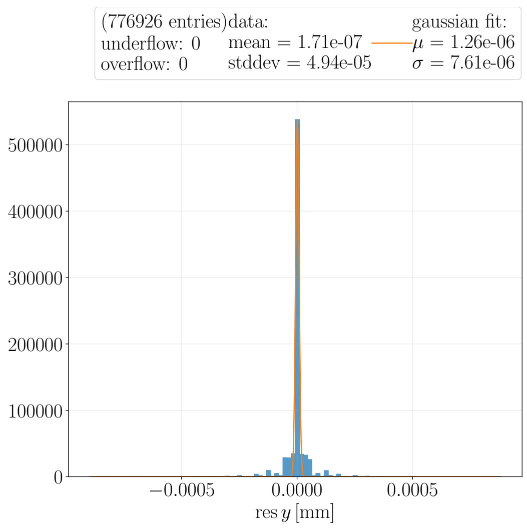


(c) Residual z

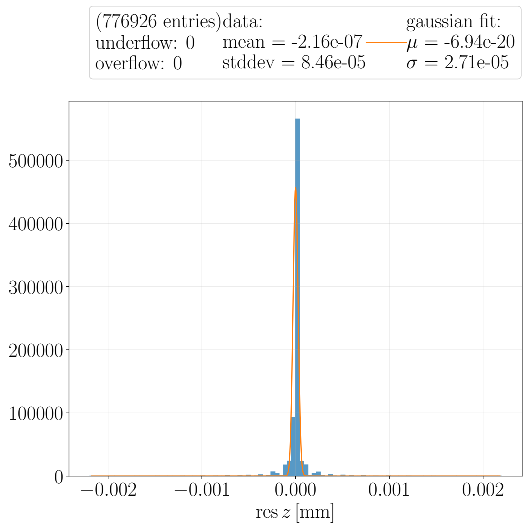
Figure C.18.: Track position residuals between the helix scan and device navigation traces for the ITk with $p_T = 100\text{GeV}$.



(a) Residual x

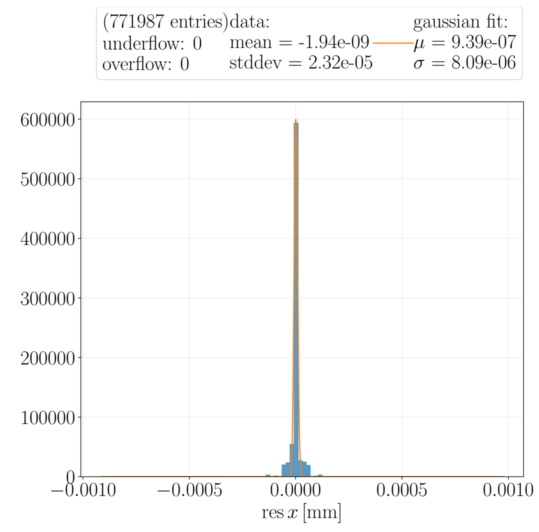


(b) Residual y

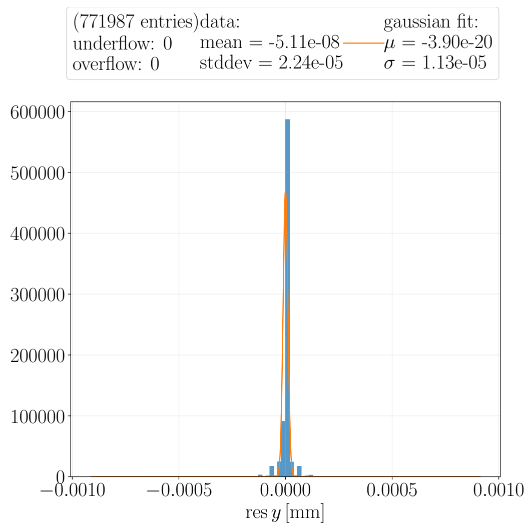


(c) Residual z

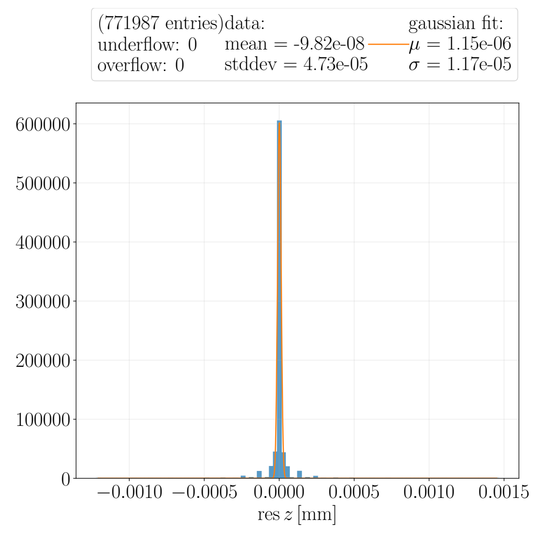
Figure C.19.: Track position residuals between the host and device navigation traces for the ITk with $p_T = 0.5\text{GeV}$.



(a) Residual x

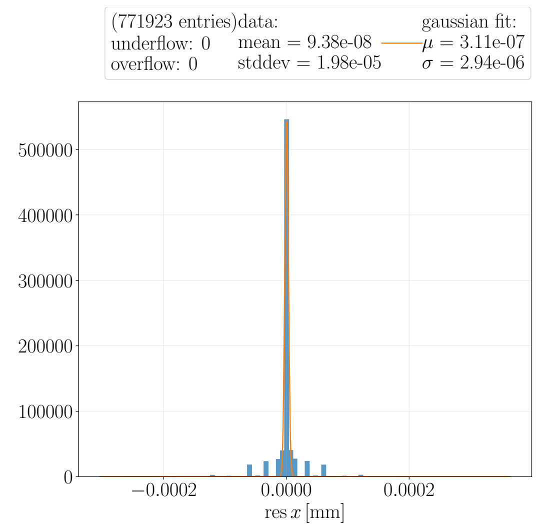


(b) Residual y

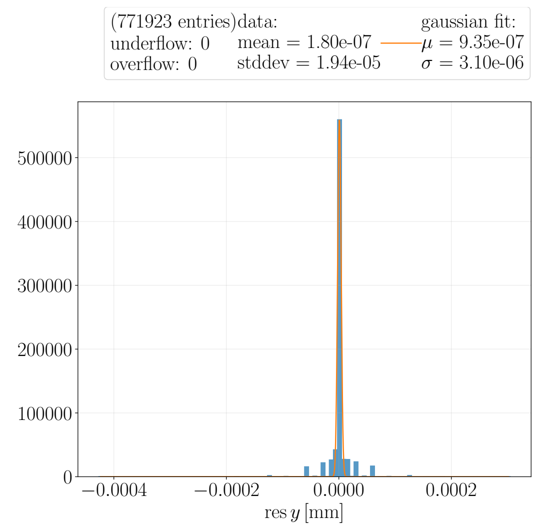


(c) Residual z

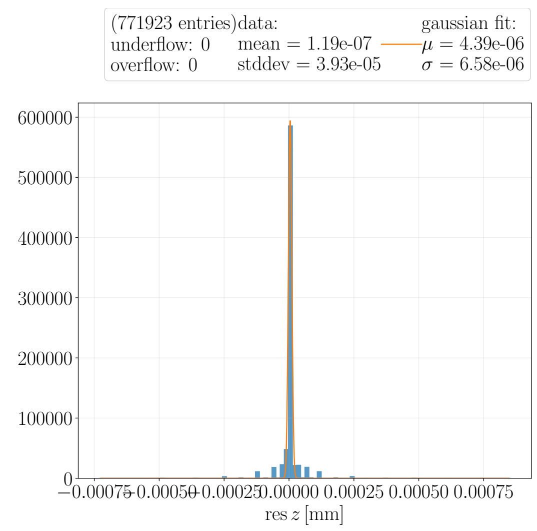
Figure C.20.: Track position residuals between the host and device navigation traces for the ITk with $p_T = 10\text{GeV}$.



(a) Residual x



(b) Residual y

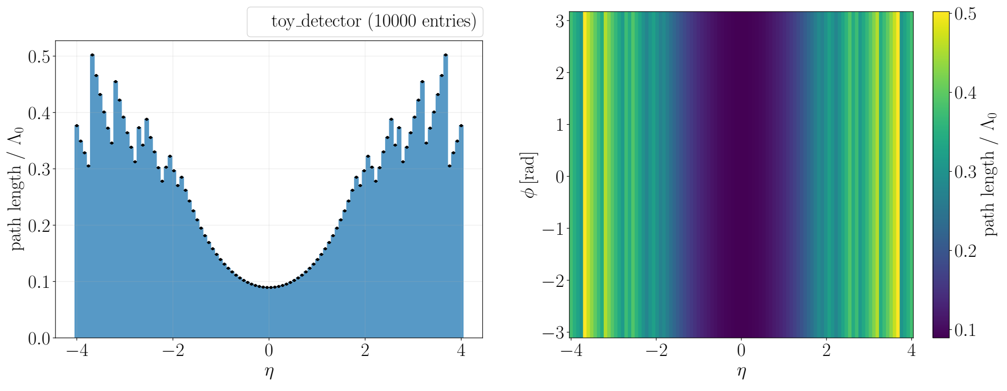


(c) Residual z

Figure C.21.: Track position residuals between the host and device navigation traces for the ITk with $p_T = 100\text{GeV}$.

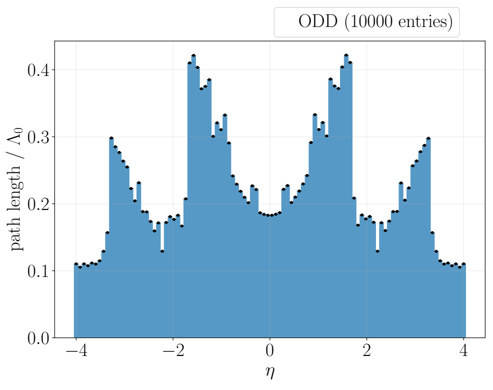
APPENDIX D

Material Validation

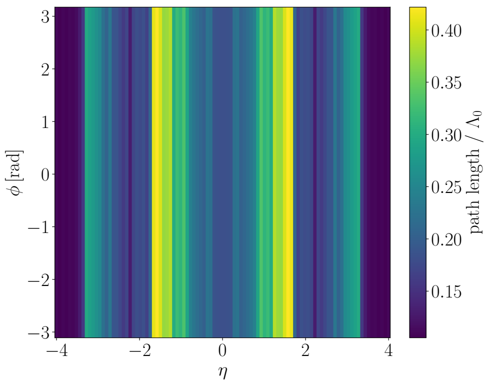


(a) Scan profile (b) Scan map

Figure D.1.: Material scan of the toy detector geometry.

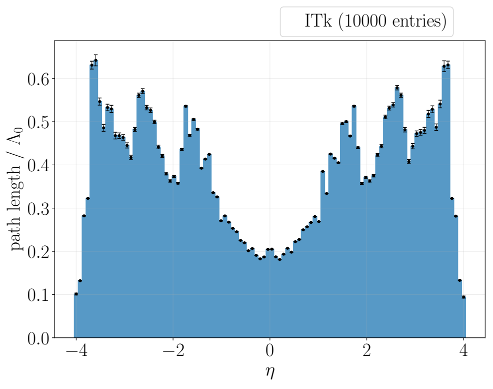


(a) Scan profile

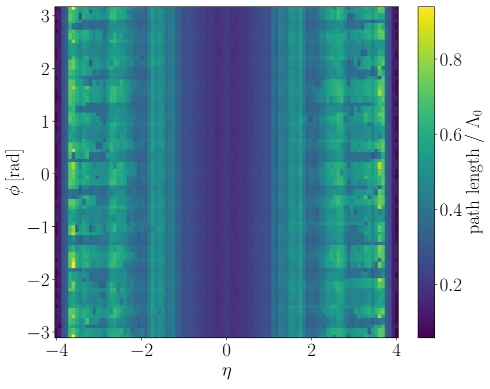


(b) Scan map

Figure D.2.: Material scan of the ODD geometry.



(a) Scan profile



(b) Scan map

Figure D.3.: Material scan of the ITk geometry.

Danksagung

I would like to take this opportunity to thank everyone who supported me throughout the realization of this thesis, in particular my supervisors Prof. Dr. Stan Lai and Dr. Andreas Salzburger. I had a great experience working on this project!

A big thank you goes to my parents, friends and colleagues for a wonderful time and for your encouragements and support, especially during those last weeks and days! Thank you all!