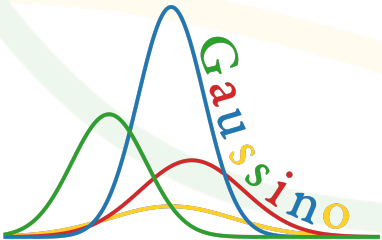
Recent developments in the Gaussino simulation software

CHEP 2024, Kraków

Marco Clemencic, Gloria Corti, Simona Dubs, James Gomes, Antonio Gómez Carrera, Michał Mazurek, **Adam Morris**, Witold Pokorski, Ruben Pozzi, Wenjie Shi

CERN, NCBJ Poland, Durham UK, IUB Bangladesh, UIMP Spain, KTH Sweden, SCNU China

2024-10-24

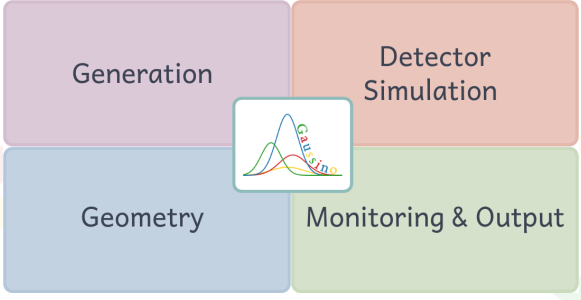


Quick overview of Gaussino

Gaussino

Main idea

- Extract the **experiment-independent functionality** from LHCb's Gauss application
- **Standalone application** with minimal functionality out of the box
  - Testbed for prototyping **new detectors** and **new technologies**
- **Toolkit** for building experiment-specific applications [*e.g.* LHCb, CEPC]

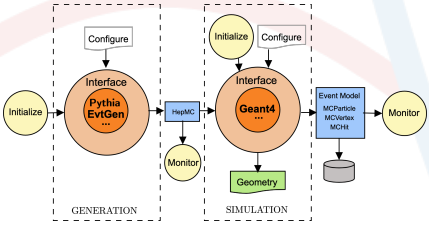


gaussino.docs.cern.ch A recent overview talk GitLab repository

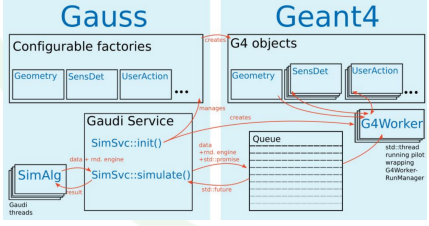
Gaussino

Key features

- Built in the Gaudi framework
- **Modular** event-generation and detector-simulation phases
- **Multi-threaded event loop** in Gaudi
- Steering of **multi-threaded Geant4**
- Support for **custom simulation** [*e.g.* ML, GPU]
- Internal geometry service, support for DD4hep
- High-level configuration in python



Modular gen and sim phases

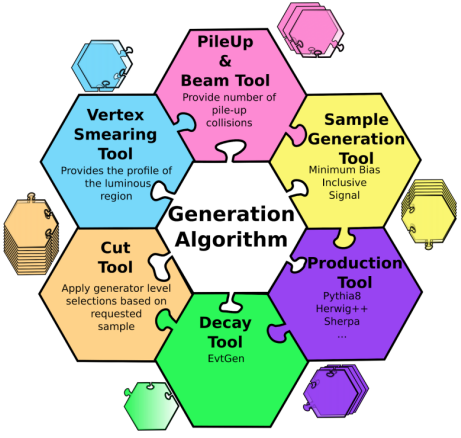


Gaudi tools as factories for Geant4 objects

Generation

Generic generator-level cuts

Generator-level cuts save time by rejecting events before detector simulation



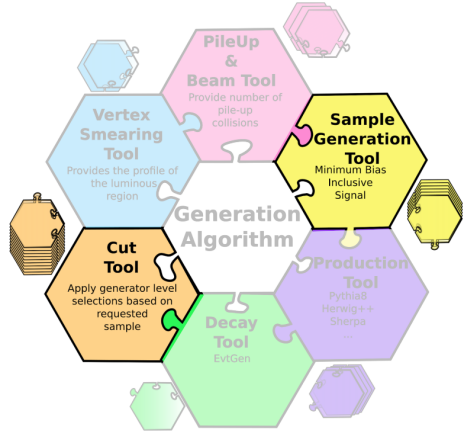
Modular structure of the generation phase

Generic generator-level cuts

Generator-level cuts save time by rejecting events before detector simulation

Signal vs Full Event

- A **signal** sample has a particle of a desired type with a desired decay
- The `GenCutTool` class applies cuts to the signal particle before the rest of the event is decayed
- Cuts on the **full event** are applied with `FullGenEventCutTool`



Tools involved in applying cuts

Generic generator-level cuts

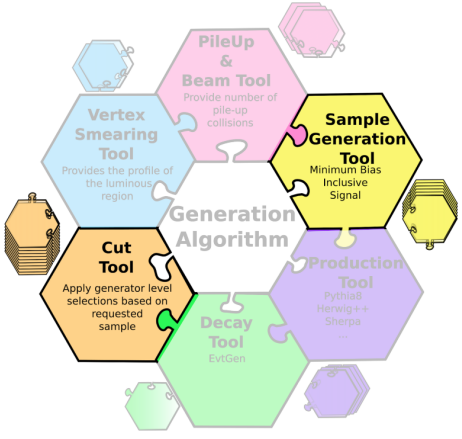
Generator-level cuts save time by rejecting events before detector simulation

Signal vs Full Event

- A **signal** sample has a particle of a desired type with a desired decay
- The `GenCutTool` class applies cuts to the signal particle before the rest of the event is decayed
- Cuts on the **full event** are applied with `FullGenEventCutTool`

Generic cut tool

- Configure arbitrary cuts at run-time
- In Gauss: LoKiGen for many years [Docs]
- New in Gaussino: **HepMC3 Filters**



Tools involved in applying cuts

Generic generator-level cuts: key ingredients

HepMC3 Features & Filters

- Gaussino uses **HepMC3 records as exchange format** between gen and sim phases
- HepMC3's 📖 "Search" library provides **native functionality for cuts**:
  - `Feature` returns a numerical value from a `HepMC3::GenParticle`
    - **comparison operators** (`!=`, `==`, `>`, `>=`, `<`, `<=`) that return a `Filter`
  - `Filter` is an alias for `std::function<bool(ConstGenParticlePtr)>`
    - **logical operators** (`!`, `||`, `&&`) that return another `Filter`
- Define a standard list of named `Features` in Gaussino's `HepMCUser` package
  - Easily extensible: 2 lines of C++ to add a new one!

Intuitive syntax in C++

```
Feature<double> PT([](ConstGenParticlePtr p)->double{return p->momentum().pt();});
Feature<double> ETA([](ConstGenParticlePtr p)->double{return p->momentum().eta();});

bool pass_cut = (abs(ETA) < 2.5 && PT > 500)(particle);
```

... but we don't want to configure our tool with C++

Generic generator-level cuts: key ingredients

HepMC3 Features & Filters

- Gaussino uses **HepMC3 records as exchange format** between gen and sim phases
- HepMC3's 📖 "Search" library provides **native functionality for cuts**:
  - `Feature` returns a numerical value from a `HepMC3::GenParticle`
    - **comparison operators** (`!=`, `==`, `>`, `>=`, `<`, `<=`) that return a `Filter`
  - `Filter` is an alias for `std::function<bool(ConstGenParticlePtr)>`
    - **logical operators** (`!`, `||`, `&&`) that return another `Filter`
- Define a standard list of named `Features` in Gaussino's `HepMCUser` package
  - Easily extensible: 2 lines of C++ to add a new one!

Cutstring parsing

- Use the `boost::spirit::qi` library to parse strings into `Filter` objects
- Inspired by the calculator example → allows nested composite expressions
- Understands units and supports the `abs` function
- *e.g.* `"abs(ETA) < 2.5 && PT > 0.5 GeV"`

Generic generator-level cuts

Current status

- Implemented a set of features covering a large fraction of existing LHCb use-cases
  - Kinematic quantities
  - Angles
  - Vertex positions

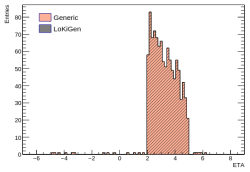


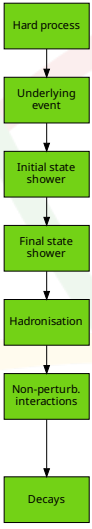
Figure 12: Pseudorapidity distribution of D^0 in a $D^0 \to K^- \pi^+$ signal.

- Tested exact reproduction of cuts from LoKiGen tool by generating samples
- Currently can only apply cuts to the signal particle…
  -  LoKiGen decay descriptor parsing
  -  Support for `HepMC3::Relatives`
  -  Cuts on the full event (conditional accumulators, min/max values)

If any of this is useful generically, we are happy to push it upstream to HepMC3

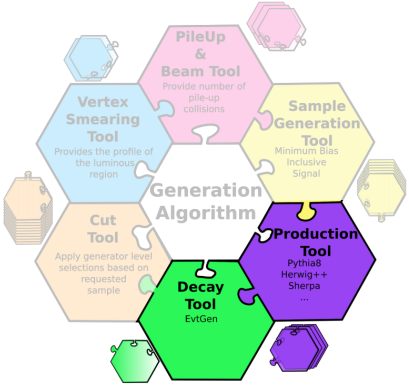
Redesigned generator interface

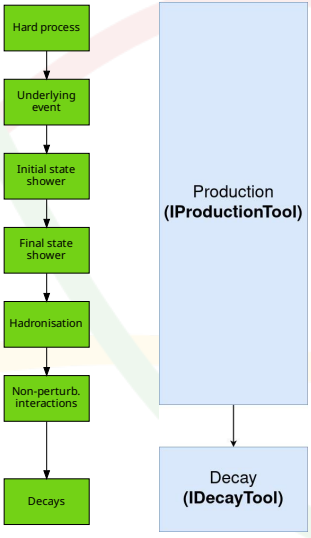
- Want **fine-grained control over different stages of generation**



Redesigned generator interface

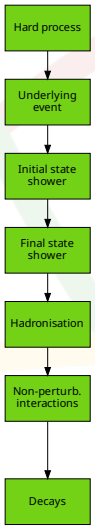
- Want **fine-grained control over different stages of generation**
- Old interface nearly monolithic

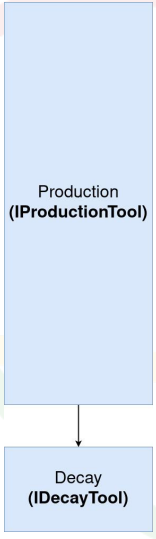


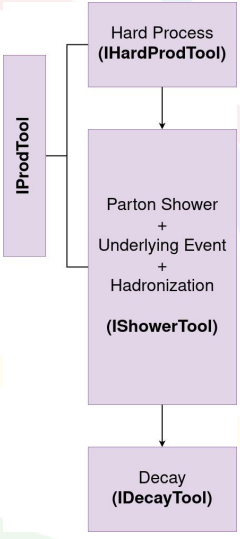


Redesigned generator interface

- Want **fine-grained control over different stages of generation**
- Old interface nearly monolithic
- **Idea:** separate **hard-process** (*e.g.* MadGraph, Powheg) from **showering & hadronisation** (*e.g.* Pythia8)
- More control in the configuration
- Design choice influenced by what generators provide as user hooks
- LHE as exchange format







[Design: Phil Ilten]

Detector Simulation

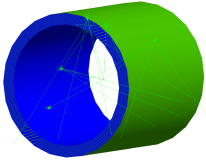
Integration of AdePT

AdePT

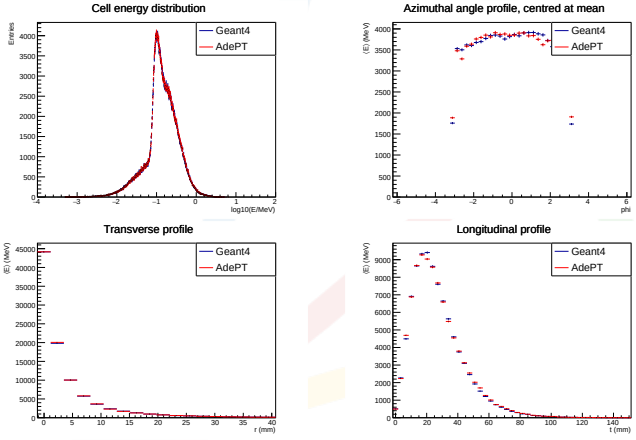
- EM particle transport on GPUs [Talk on Monday]
  - Whole detector or specific region
- Physics with **G4HepEM**
- Geometry with **VecGeom**
- Interface with Geant4 via **special `G4VTrackingManager`**
  - Configured via physics list & macro commands or API

Integration with Gaussino

- Test of AdePT inside a simulation application
- Compare output of AdePT with Geant4
- Make use of CaloChallenge geometry [Talk on Tuesday]
  - designed for evaluating ML models for fast simulation



Integration of AdePT



NB: Agreement is statistical due to different implementations

Generate 10 events, each with 1000× photons at 1 GeV

Overall **good agreement**

Speedup depends on particle multiplicity → may want to selectively offload

- Our test: 5× speedup

⚠ Missing full MCTruth & ‘user’ track information on GPU

Monitoring

Material scan tool

Idea

Tool that scans the detector to map interaction- and radiation-length

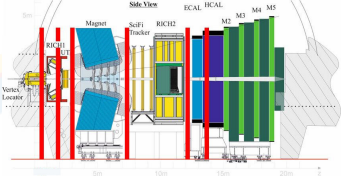
- Shoot a non-interacting particle and accumulate interaction/radiation-length in each volume it passes through
- Save values at scoring planes $\rightarrow$ 2D maps
- Crucial for studies of new detector designs and checking data–simulation agreement

Material scan tool

Implementation

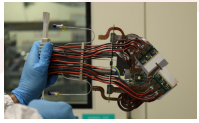
Re-write of existing tool from Gauss

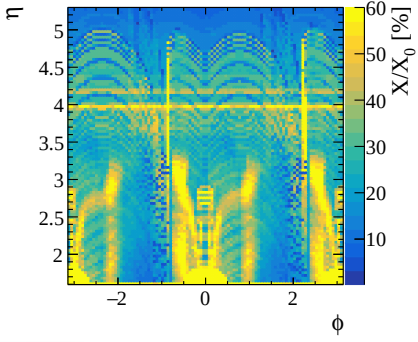
- Extend the particle-gun scan to **different geometries**
  - forward $\rightarrow x - y$ or $\eta - \phi$ maps in steps of z
  - cylindrical $\rightarrow z - \phi$ maps in steps of ρ
- Accumulation and scoring made **agnostic to coordinate system**
- Port to Gaussino's way of steering Geant4
  - Gaudi tool as factory for `G4UserSteppingAction`
- Use thread-safe Gaudi histograms
- Create the scoring planes in **Parallel Geometry** to handle overlaps



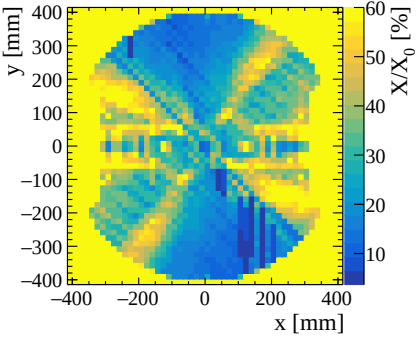
Example: LHCb Vertex Locator

Scoring plane placed just after the VELO





$\eta-\phi$ scan within acceptance



$x-y$ scan showing the electronics and structures outside acceptance

Configuration with GaudiConfig2

Configuration in Gaudi

Gaudi **objects** (algorithms, tools, services) are configured by setting values of **properties** through **python bindings**.

Users interact with **high-level configurables** which translate options to low-level properties.

Old Configurables

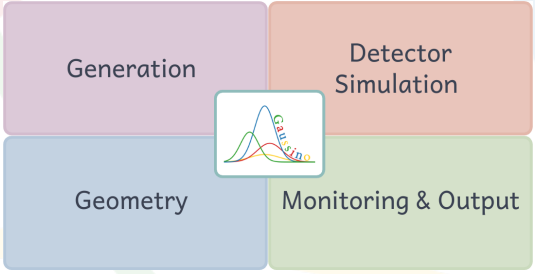
- Global singletons (name-based)
- Implicit and silent overrides
- Patchy type-checking
- Unset properties do not exist

GaudiConfig2

- Local objects + functions that return configurables
- Overrides must be explicit by wrapping functions
- Precise and easily-extensible type-checking
- Unset properties return default values

Implementation in Gaussino

- Keep modular structure of Gaussino configuration
- Some extra bookkeeping to collect configurable objects
- Pass around a pydantic object containing user-set high-level options



High-level options with Pydantic

- Pydantic provides an easy way to do type-checking and set default options
  - Custom types
  - Validator functions
  - Dynamic default values *e.g.* based on other options
- Defines a YAML schema

Command-line execution

- Native `gaudirun.py` → limited flexibility

```
gaudirun.py SomeModule:my_function
```

- Implementation with `LbExec` → LHCb-specific → port core functionality to Gaudi?
  - Handles reading & merging YAML files and conversion to pydantic objects
  - Custom command-line flags

```
lbexec module:function options.yaml[+more_options.yaml][+...] [--flag] [arg] [...]
```

- Exploring alternatives (*e.g.* take inspiration from Athena)

Conclusion

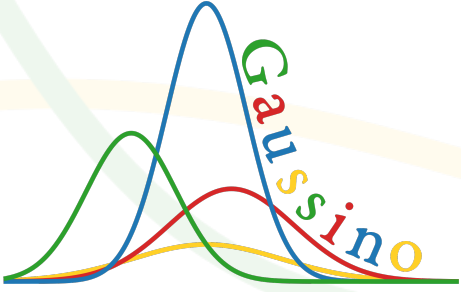
Conclusion

Summary

- Improved support for hard-process-only generators
- Simplified and more experiment-independent generic generator cuts
- Reference implementation for accelerated particle transport
- Updated material scan tool, now geometry-agnostic
- More robust configuration

Want to get involved?

- Mattermost Team
- Alternating Wednesdays 14:30
- CERN e-group: `gsino-user`
- gaussino.docs.cern.ch
- GitLab repository



Thank you for listening

Appendix

Generic generator-level cuts: extend functors

Define a new HepMC3::Feature in the HepMCUser package…

```
namespace HepMC3 {
  namespace StandardFeature {
    using Particle = ConstGenParticlePtr;
    const Feature<double> P( []( Particle p ) -> double { return p->momentum().p3mod(); } );
    const Feature<double> PT( []( Particle p ) -> double { return p->momentum().pt(); } );
    const Feature<double> PX( []( Particle p ) -> double { return p->momentum().px(); } );
    const Feature<double> PY( []( Particle p ) -> double { return p->momentum().py(); } );
    const Feature<double> PZ( []( Particle p ) -> double { return p->momentum().pz(); } );
    const Feature<double> ENERGY( []( Particle p ) -> double { return p->momentum().e(); } );
  }
}
```

… and add a new entry in the map of names to features

```
struct functor_table : qi::symbols<char, const Feature*> {
  functor_table() {
    using namespace HepMC3::StandardFeature;
    add
        // Momentum:
        ( "P", &P )( "PT", &PT )( "PX", &PX )( "PY", &PY )( "PZ", &PZ )
        // Energy:
        ( "ENERGY", &ENERGY )
  }
} functor;
```

High-level Options: Pydantic classes

```python
class GaussinoOptions(Options): # LbExec.options.Options
    """Properties of the main application."""

    # Main
    evt_max: PositiveInt = 1
    phases: list[Phases] = [  # Phases is an Enum
        Phases.Generator,
        Phases.Simulation,
    ]
    first_evt: PositiveInt = 1  # pydantic.PositiveInt
    run_number: PositiveInt = 1
    ...
    # Beam options
    beam: BeamOptions = BeamOptions()
    # Phase options
    generation: GenerationOptions = GenerationOptions()
    simulation: Optional[SimulationOptions]
    data_type: Optional[str]  # Expected by base-class
                              # but we don't use it
```

High-level Options: custom types & checkers

Can define custom types with validator functions to check that the value conforms to some user-defined rules

```
class EventType(str):
    """A string composed of 8 decimal digits"""

    @classmethod
    def __get_validators__(cls):
        yield cls.validate

    @classmethod
    def validate(cls, value: str) -> str:
        assert len(value) == 8 and value.isdigit()
        return value
```

```
class ConfigurableName(str):
    """The name of a configurable"""

    @classmethod
    def __get_validators__(cls):
        yield cls.validate

    @classmethod
    def validate(cls, value: str) -> str:
        assert hasattr(GaudiConfig2.Configurables, value)
        return value
```

High-level Options: dynamic default values

Can set dynamic default values using validators e.g. based on values of other options

```
@validator("output_name", always=True)
def set_output_name(cls, output_name, values) -> str:
    """Build a name for the output file, based on input options.
    Combines DatasetName, EventType, Number of events and Date
    """
    if output_name:
        return output_name

    segments = [values["dataset_name"]]

    if evt_type := values.get("generation", {}).get("event_type"):
        segments += [evt_type]

    if (evtmax := values["evt_max"]) > 0:
        # Somewhat redundant since this is forced to be >= 1
        segments += [f"{evtmax}ev"]

    segments += [time.strftime("%Y%m%d")]

    return "-".join(segments)
```