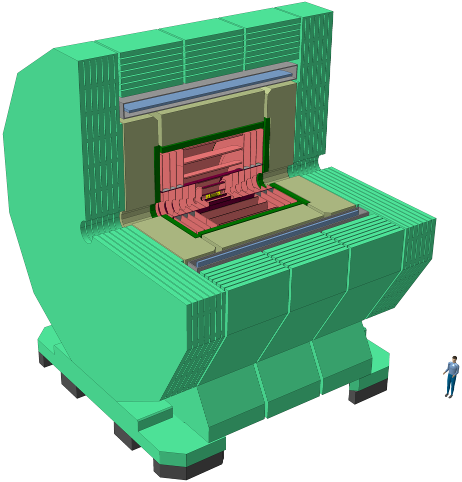
Key4hep: A Software Stack for Future-Collider Studies

André Sailer
for the Key4hep Developers

CERN-EP-SFT

Muon Collider Collaboration Meeting
October 12, 2022

Table of Contents

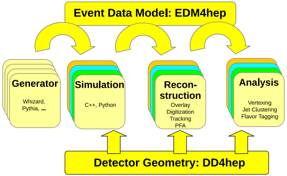


Key4hep Software Stack

Key4hep: Turnkey Software Stack

Create a software stack that connects and extends individual packages towards a complete data processing framework for detector studies with fast or full simulation, reconstruction, and for analysis

- Major ingredients: Event Data Model (EDM), Geometry Information, Processing Framework
- Sharing common components reduces overhead for all users
- Should be easy to use for librarians, developers, users
    - **easy to deploy, extend, set up**
- Full of functionality: plenty of examples for simulation and reconstruction of detectors
- Preserve and adapt existing functionality into the stack, e.g., from iLCSoft, FCCSW, CEPCSW



Event Data Model: EDM4hep

Generator
Whizard,
Pythia, ...

Simulation
C++, Python

Recon-struction
Overlay
Digitization
Tracking
PFA

Analysis
Vertexing
Jet Clustering
Flavor Tagging

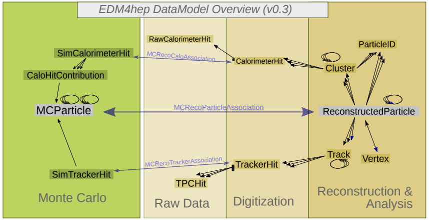
Detector Geometry: DD4hep

Ingredients

The Key4hep EDM: EDM4hep

For a high degree of interoperability, EDM4hep provides a common event data model

- Using podio to manage the EDM (described by `yaml`) and easily change the persistency layer (ROOT, SIO, . . .)
- EDM4hep data model based on LCIO and FCC-edm
- http://github.com/key4hep/edm4hep
- Recent developments for podio or EDM4hep
  - EDM4hep: additional types, associations
  - podio: event, run, collection metadata; UserDataCollection, Subset Collections, Frame
- A number of issues still need to be resolved
  - “Wrapper” for using different hit types transparently
  - multi-threading (the *frame* was recently added to simplify this)
  - schema evolution



Framework: Gaudi

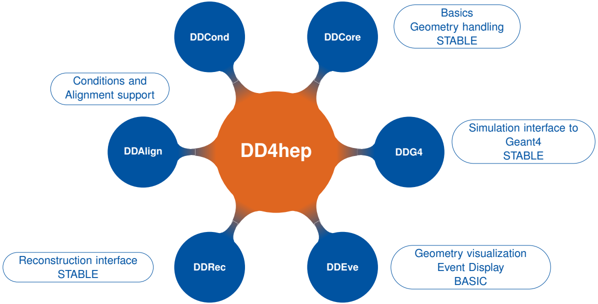
- Data processing frameworks are the skeleton on which HEP applications are built
- Gaudi was chosen as the framework, based on considerations for
  - portability to various computing resources, architectures and accelerators
  - support for task-oriented concurrency
  - adoption and developer community size; is used by LHCb, ATLAS
- Contribute developments were we see a need

k4FWCore

- Basic IO functionality: podio data service
- Reproducible random number seeding

Geometry Information: DD4hep

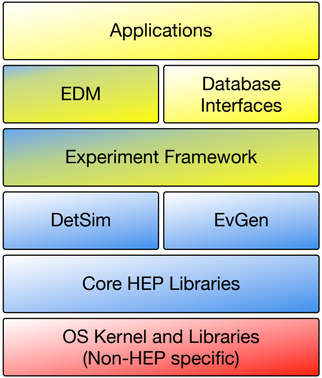
- Complete Detector Description
  - Providing geometry, materials, visualization, readout, alignment, calibration. . .
- Single source of information → consistent description
  - Use in simulation, reconstruction, analysis
- Supports full experiment life cycle
  - Detector concept development, detector optimization, construction, operation
  - Facile transition from one stage to the next



- DD4hep already in use by ILC, CLIC, FCC, and many more

Packaging: Spack

- Need to build a large number of packages to run our applications
- Adopted **spack** as the package manager
- Go beyond sharing of *build results* to sharing of *build recipes*
    - Many packages have build recipes provided by the spack community
    - Separate repository for Key4hep specific recipes
- Can build any and all pieces of the stack with minimum effort
    - `spack install key4hep-stack`
    - `spack dev-build conformaltracking@master`
- Used for nightly builds and releases of the stack
    - `source /cvmfs/sw-nightlies.hsf.org/key4hep/setup.sh`



Applications

EDM

Database Interfaces

Experiment Framework

DetSim

EvGen

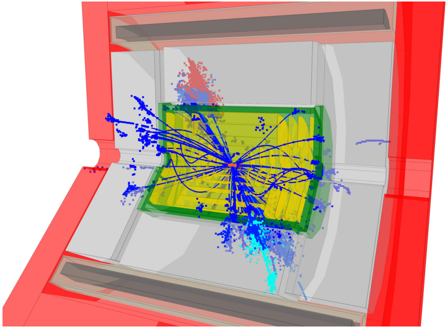
Core HEP Libraries

OS Kernel and Libraries (Non-HEP specific)

From Marlin to Gaudi

CLIC Reco Evolution: Adiabatic Changes

- Full CLIC reconstruction implemented in iLCSoft
- While transitioning to Key4hep, need to be able to keep running the CLIC reconstruction
- Switch components one by one, validate changes
  - Geometry provided by DD4hep, no changes needed
  - Move framework from Marlin to Gaudi: wrap existing processors
  - Move from `LCIO` to EDM4hep
  - Replace wrapped processors with native Gaudi algorithms, where necessary
- Incidentally will make iLCSoft functionality available to other users of the stack



Marlin & Gaudi

Apart from some naming conventions, very similar ideas in the two frameworks*

	Marlin	Gaudi
language	`c++`	`c++`
working unit	Processor	Algorithm
configuration language	XML	Python
set up function	`init`	`initialize`
working function	`processEvent`	`execute`
wrap up function	`end`	`finalize`
Transient data format	`LCIO`	anything
Executable	Marlin	k4run

- To start using Gaudi: use a generic wrapper around the processors.
- Implementation: **https://github.com/key4hep/k4MarlinWrapper**
- Read LCIO files and pass the `LCIO::Event` to our processors

---

*Of course subtle differences emerge

Wrapper Configuration

- Translate the `XML` to `python`, using a stand alone python script: `convertMarlinSteeringToGaudi.py`
- Pass arbitrary number, types, and names of parameters to the processor

Marlin/XML

```
<processor name="VXDBarrelDigitiser" type="DDPlanarDigiProcessor">
  <parameter name="SubDetectorName" type="string">Vertex </parameter>
  <parameter name="IsStrip" type="bool">false </parameter>
  <parameter name="ResolutionU" type="float"> 0.003 0.003 0.003 0.003 0.003 0.003 </parameter>
  <parameter name="ResolutionV" type="float"> 0.003 0.003 0.003 0.003 0.003 0.003  </parameter>
  <parameter name="SimTrackHitCollectionName" type="string" lcioInType="SimTrackerHit">
          VertexBarrelCollection </parameter>
  <parameter name="SimTrkHitRelCollection" type="string" lcioOutType="LCRelation">
          VXDTrackerHitRelations </parameter>
  <parameter name="TrackerHitCollectionName" type="string" lcioOutType="TrackerHitPlane">
          VXDTrackerHits </parameter>
  <parameter name="Verbosity" type="string">WARNING </parameter>
</processor>
```

Wrapper Configuration

- Translate the `XML` to `python`, using a stand alone python script: `convertMarlinSteeringToGaudi.py`
- Pass arbitrary number, types, and names of parameters to the processor

Gaudi/Python

```
VXDBarrelDigitiser = MarlinProcessorWrapper("VXDBarrelDigitiser")
VXDBarrelDigitiser.OutputLevel = WARNING
VXDBarrelDigitiser.ProcessorType = "DDPlanarDigiProcessor"
VXDBarrelDigitiser.Parameters = {
                                 "IsStrip": ["false"],
                                 "ResolutionU": ["0.003"] * 6,
                                 "ResolutionV": ["0.003"] * 6,
                                 "SimTrackHitCollectionName": ["VertexBarrelCollection"],
                                 "SimTrkHitRelCollection": ["VXDTrackerHitRelations"],
                                 "SubDetectorName": ["Vertex"],
                                 "TrackerHitCollectionName": ["VXDTrackerHits"],
                                 }
```

Configuration: Control flow

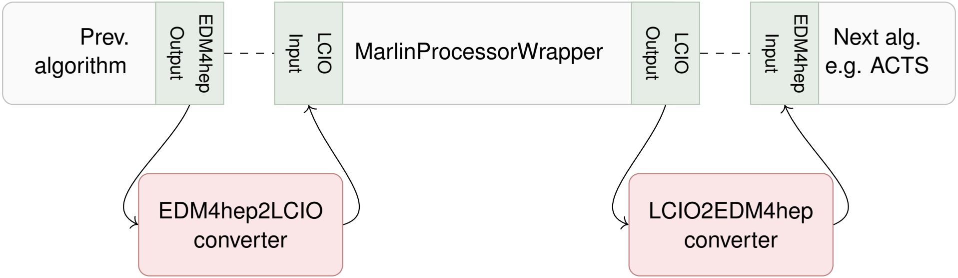
- XML `execute` section translated to a python list

```xml
<execute>
  <processor name="MyAIDAProcessor"/>
  <processor name="EventNumber" />
  <processor name="InitDD4hep"/>
  <processor name="Config" />
  <!-- ... -->
</execute>
```

```python
algList = []
algList.append(lcioReader)
algList.append(MyAIDAProcessor)
algList.append(EventNumber)
algList.append(InitDD4hep)
algList.append(OverlayFalse)
algList.append(VXDBarrelDigitiser)
#...
```

Event Data Model Conversion in Memory

- To use EDM4hep files as primary input, have to convert EDM4hep to LCIO and back at run time
- Integrate iLCSoft processors with Gaudi-based processors
- Configurable which collections to convert for which processor



Event Data Model Conversion in Memory

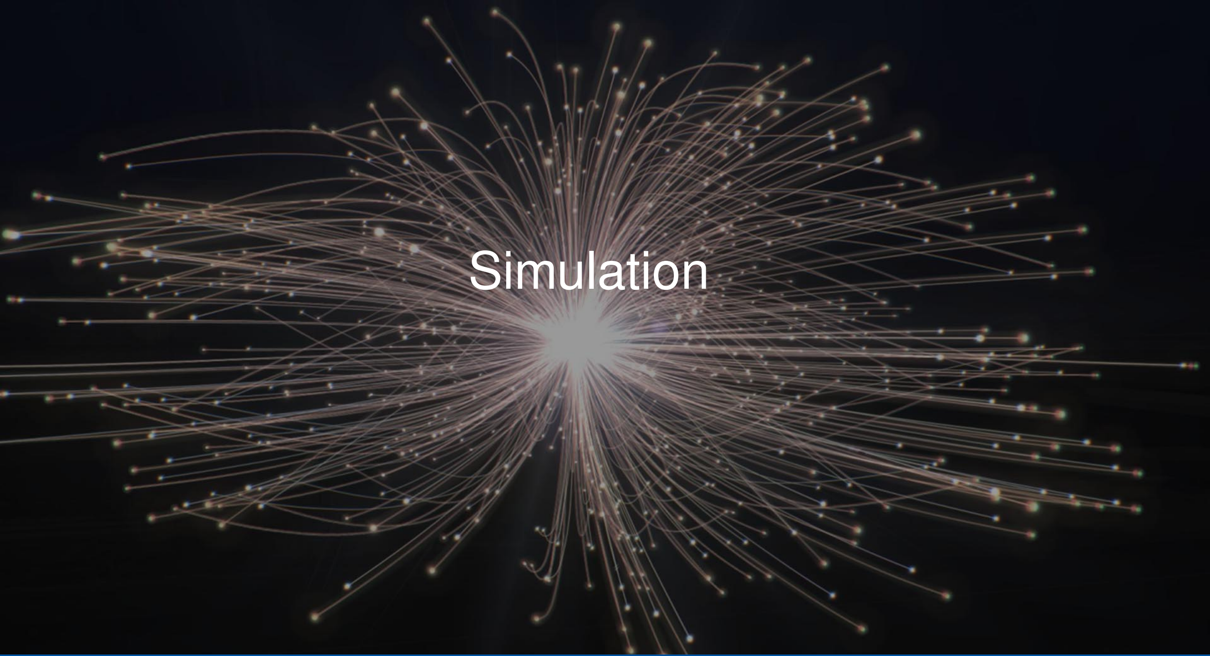
- To use EDM4hep files as primary input, have to convert EDM4hep to LCIO and back at run time
- Integrate iLCSoft processors with Gaudi-based processors
- Configurable which collections to convert for which processor

```
# EDM4hep to LCIO
edmConvTool = EDM4hep2LcioTool("VXDBarrelEDM4hep2lcio")
edmConvTool.Parameters = [
  "VertexBarrelCollection", "VertexBarrelCollection",
  ]
edmConvTool.OutputLevel = DEBUG
VXDBarrelDigitiser.EDM4hep2LcioTool = edmConvTool
```

```
# LCIO to EDM4hep
VXDBarrelDigitiserLCIOConv =
   Lcio2EDM4hepTool("VXDBarrelDigitiserLCIOConv")
VXDBarrelDigitiserLCIOConv.Parameters = [
  "VXDTrackerHits", "VXDTrackerHits",
  "VXDTrackerHitRelations", "VXDTrackerHitRelations"
  ]
VXDBarrelDigitiserLCIOConv.OutputLevel = DEBUG
VXDBarrelDigitiser.Lcio2EDM4hepTool =
    VXDBarrelDigitiserLCIOConv
```

Running with Gaudi

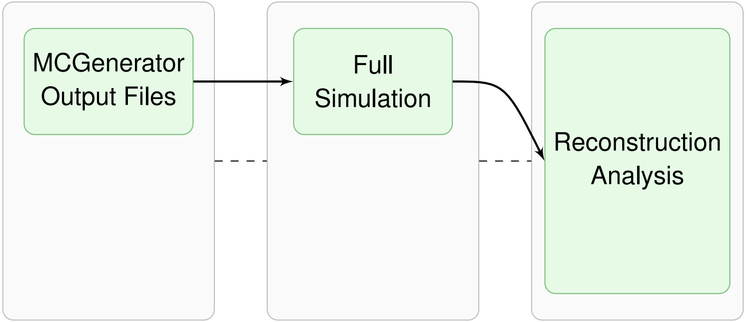
- After conversion of the Marlin steering file with `convertMarlinSteeringToGaudi.py`, and maybe some tweaks for the EDM4hep to LCIO conversions
- Run the workflow with: `k4run muonRec.py`



Simulation

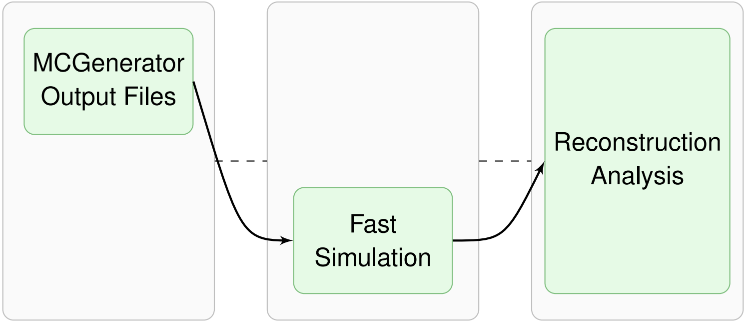
Simulation Integration in the Software Stack

- The simulations can be run in a stand-alone mode using the output from a Generator as input
- Create its own input via “particle gun”, at least for full simulation
- Run as part of a chain inside a framework, where k4Gen calls a MC Generator, or reads an input file
- In all cases, the following step of (high level) reconstruction or analysis should be usable in the same way



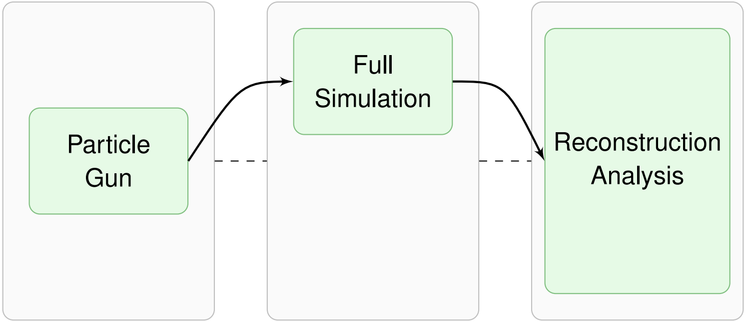
Simulation Integration in the Software Stack

- The simulations can be run in a stand-alone mode using the output from a Generator as input
- Create its own input via “particle gun”, at least for full simulation
- Run as part of a chain inside a framework, where k4Gen calls a MC Generator, or reads an input file
- In all cases, the following step of (high level) reconstruction or analysis should be usable in the same way



Simulation Integration in the Software Stack

- The simulations can be run in a stand-alone mode using the output from a Generator as input
- Create its own input via “particle gun”, at least for full simulation
- Run as part of a chain inside a framework, where k4Gen calls a MC Generator, or reads an input file
- In all cases, the following step of (high level) reconstruction or analysis should be usable in the same way



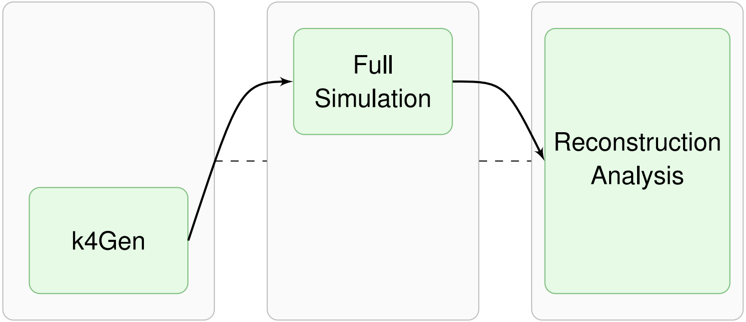
Particle Gun

Full Simulation

Reconstruction Analysis

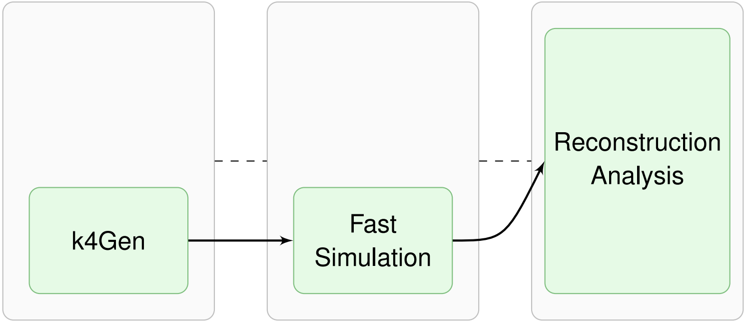
Simulation Integration in the Software Stack

- The simulations can be run in a stand-alone mode using the output from a Generator as input
- Create its own input via “particle gun”, at least for full simulation
- Run as part of a chain inside a framework, where k4Gen calls a MC Generator, or reads an input file
- In all cases, the following step of (high level) reconstruction or analysis should be usable in the same way



Simulation Integration in the Software Stack

- The simulations can be run in a stand-alone mode using the output from a Generator as input
- Create its own input via “particle gun”, at least for full simulation
- Run as part of a chain inside a framework, where k4Gen calls a MC Generator, or reads an input file
- In all cases, the following step of (high level) reconstruction or analysis should be usable in the same way



Geant4 Full Simulation Interfaces

- `ddsim` standalone program and `k4SimGeant4` framework integrated solution
  - Also looking at integrating with LHCb's `Gaussino` (Using DD4hep geometry, gaudi based processing)
- Both approaches have to provide the same functionality: sensitive detectors, MC History, particle guns, physics list construction and configuration,
  - Ideally by the same implementation, but we are not there yet

ddsim: Full Simulation Example

- Only change needed to go from **LCIO** to EDM4hep output is the output file name

https://key4hep.github.io/key4hep-doc/examples/clic.html

```
source /cvmfs/sw-nightlies.hsf.org/key4hep/setup.sh
git clone https://github.com/iLCSoft/CLICPerformance
ddsim --compactFile $LCGEO/CLIC/compact/CLIC_o3_v14/CLIC_o3_v14.xml \
      --outputFile ttbar.slcio \
      --steeringFile clic_steer.py \
      --inputFiles ../Tests/yyxyev_000.stdhep \
      --numberOfEvents 3
```

ddsim: Full Simulation Example

- Only change needed to go from LCIO to **EDM4hep** output is the output file name

https://key4hep.github.io/key4hep-doc/examples/clic.html

```
source /cvmfs/sw-nightlies.hsf.org/key4hep/setup.sh
git clone https://github.com/iLCSoft/CLICPerformance
ddsim --compactFile $LCGEO/CLIC/compact/CLIC_o3_v14/CLIC_o3_v14.xml \
      --outputFile ttbar_edm4hep.root \
      --steeringFile clic_steer.py \
      --inputFiles ../Tests/yyxyev_000.stdhep \
      --numberOfEvents 3
```

Configuring and Running k4SimGeant4

```
from Configurables import (SimG4Alg, SimG4SaveTrackerHits, SimG4UserLimitPhysicsList, GeoSvc, SimG4Svc,
                           SimG4FullSimActions)
# parse the given xml file
geoservice = GeoSvc("GeoSvc")
geoservice.detectors = [os.path.join(path_to_detectors, 'Detector/DetFCCeeIDEA/compact/FCCee_DectMaster.xml')]
# configure sensitive detector
savetrackertool_DCH = SimG4SaveTrackerHits("saveTrackerHits_DCH")
savetrackertool_DCH.readoutNames = ["DriftChamberCollection"]
savetrackertool_DCH.SimTrackHits.Path = "positionedHits_DCH"
SimG4Alg("SimG4Alg").outputs += [savetrackertool_DCH]
# Setup for physicslist
physicslisttool = SimG4UserLimitPhysicsList("Physics");     physicslisttool.fullphysics = "SimG4FtfpBert"
# enable MC history
actions = SimG4FullSimActions();      actions.enableHistory=True
# configure geant4
geantservice = SimG4Svc("SimG4Svc")
geantservice.detector = 'SimG4DD4hepDetector'
geantservice.physicslist = physicslisttool;      geantservice.actions = actions
geantservice.magneticField = field
geantservice.g4PostInitCommands +=["/process/eLoss/minKinEnergy 1 MeV", "/tracking/storeTrajectory 1"]
```

- Execute with `k4run simGeant4.py`
- Some steering files available for **FCC detectors**.

Delphes Fast Simulation

- “Delphes is a modular framework that simulates the response of a multipurpose detector in a parameterised fashion”
  - See **M. Selvaggi: Progress in DELPHES**
- Delphes integration to Key4hep framework: **key4hep/k4SimDelphes** and its **documentation**
- To integrate Delphes to Key4hep, we need to obtain EDM4hep output from it

Using Delphes Fast Simulation: Standalone

- Pick the Delphes card of your chosen detector
- Configuration for EDM4hep output: `edm4hep_output_config.tcl` which collections to store in the output file
- Pythia8 configuration: `p8_noBES_ee_ZH_ecm240.cmd`
- output file name: `delphes_events_edm4hep.root`

```
DelphesPythia8_EDM4HEP ${DELPHES_DIR}/cards/delphes_card_IDEA.tcl \
                       ${K4SIMDELPHES}/edm4hep_output_config.tcl \
                       p8_noBES_ee_ZH_ecm240.cmd \
                       delphes_events_edm4hep.root
```

There are other standalone programs in Key4hep to run for different input sources:

- `DelphesPythia8EvtGen_EDM4HEP`
- `DelphesROOT_EDM4HEP`
- `DelphesSTDHEP_EDM4HEP`

Using Delphes Fast Simulation: Framework Integrated

- Configure Delphes 'Algorithm' with similar arguments as standalone

```
#...
from Configurables import k4SimDelphesAlg
delphesalg = k4SimDelphesAlg()
delphesalg.DelphesCard = "delphes_card_IDEA.tcl"
delphesalg.DelphesOutputSettings = "edm4hep_output_config.tcl"
delphesalg.GenParticles.Path = "GenParticles"
#...
```

- Execute complete steering file: `k4run simDelphes.py`
  - Example **steering file**

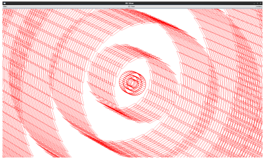
Developments

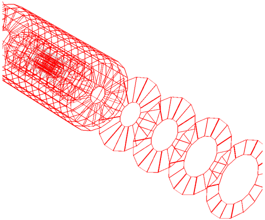
k4Clue

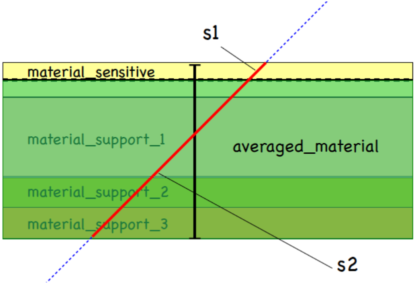
- Investigating use of the GPU friendly algorithm **CLUE** (CLUstering of Energy) as part of particle flow reconstruction
- CLUE Gaudi algorithm created: **k4Clue** and run as part of the CLIC reconstruction chain
- Validation and use of the clusters pending

k4ActsTracking

- Started work towards integration of the ACTS tracking toolkit with Key4hep:
  - Planning to create thin Gaudi Algorithm(s) converting necessary information for ACTS and tracks back to EDM4hep
- Try to use information provided by `dd4hep::rec::Surface` class to ACTS
  - Surfaces can be added after the fact to the geometry instantiation



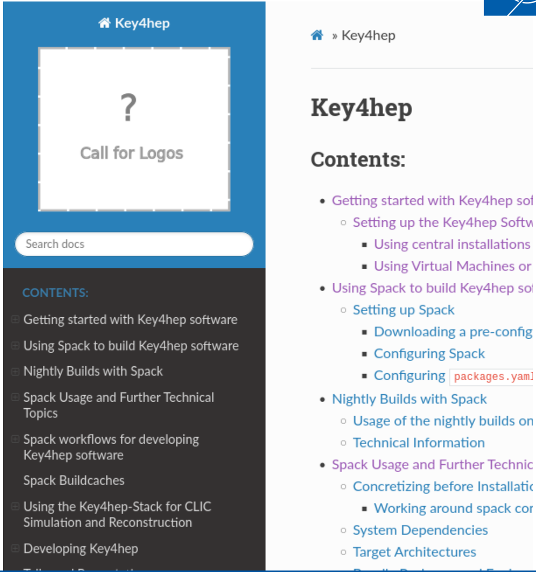




Documentation

Documentation

- Main documentation page key4hep.github.io based on GitHub pages https://github.com/key4hep/key4hep-doc
- Test the examples in the documentation via `notedown`
- Doxygen, e.g., EDM4hephttps://edm4hep.web.cern.ch/
- CLIC simulation and reconstruction example
  - Would be nice to add the CEPC workflows as well
- Restructuring of documentation in the works
  - Separate *User*, *Developer*, *Librarian* content



Key4hep

Call for Logos

Search docs

CONTENTS:

- Getting started with Key4hep software
- Using Spack to build Key4hep software
- Nightly Builds with Spack
- Spack Usage and Further Technical Topics
- Spack workflows for developing Key4hep software
- Spack Buildcaches
- Using the Key4hep-Stack for CLIC Simulation and Reconstruction
- Developing Key4hep

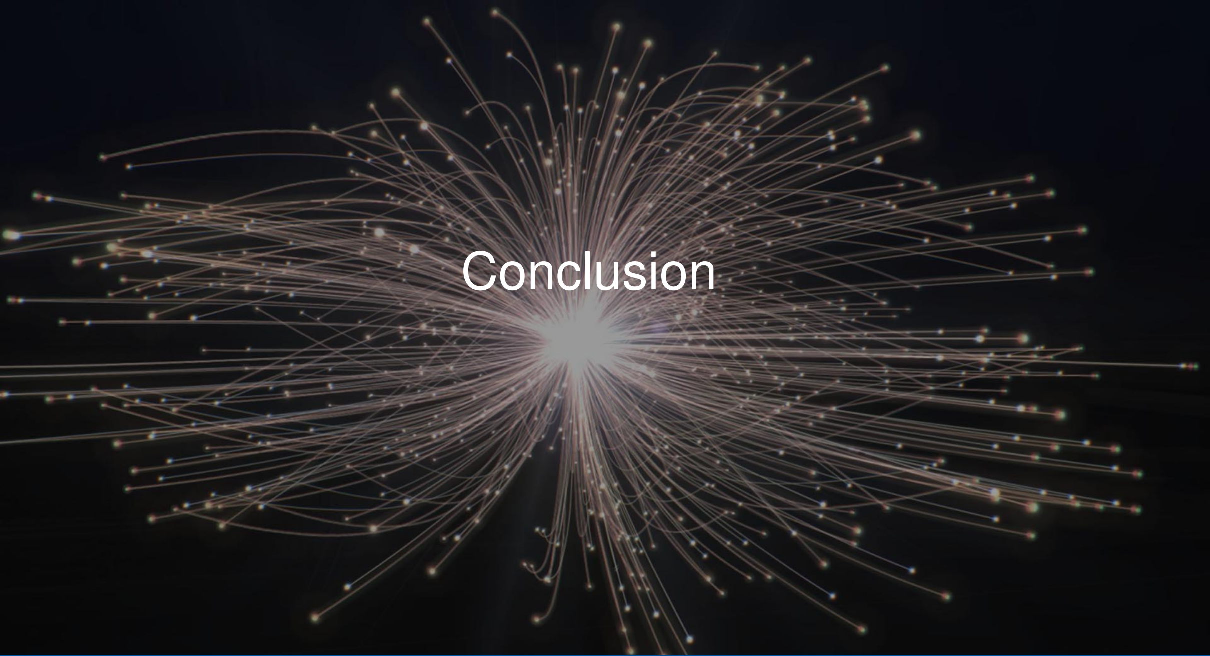
» Key4hep

Key4hep

Contents:

- Getting started with Key4hep sof
  - Setting up the Key4hep Softw
    - Using central installations
    - Using Virtual Machines or
- Using Spack to build Key4hep sof
  - Setting up Spack
    - Downloading a pre-config
    - Configuring Spack
    - Configuring `packages.yaml`
- Nightly Builds with Spack
  - Usage of the nightly builds on
  - Technical Information
- Spack Usage and Further Technic
  - Concretizing before Installatic
    - Working around spack cor
  - System Dependencies
  - Target Architectures

Conclusion



Conclusion

- Key4hep is the common framework of future Higgs factory studies: CEPC, CLIC, FCC, ILC
  - Parts of the stack also adopted by the EIC
  - New parties always welcome
- Essentially all functionality from iLCSoft preserved in the process
- On going developments to support multi-threading
- Integration of new common solutions: Gaussino, ACTS, CLUE

Thank you for your attention!

Acknowledgements

This work benefited from support by the CERN Strategic R&D Programme on Technologies for Future Experiments **(CERN-OPEN-2018-006)**.
This project has received funding from the European Union's Horizon 2020 Research and Innovation programme under Grant Agreement no. 101004761.
This project has received funding from the European Union's Horizon 2020 Research and Innovation programme under Grant Agreement no. 871072.