Karlsruhe Institute of Technology

SCC

Software Workshop

Giacomo De Pietro *giacomo.pietro@kit.edu*

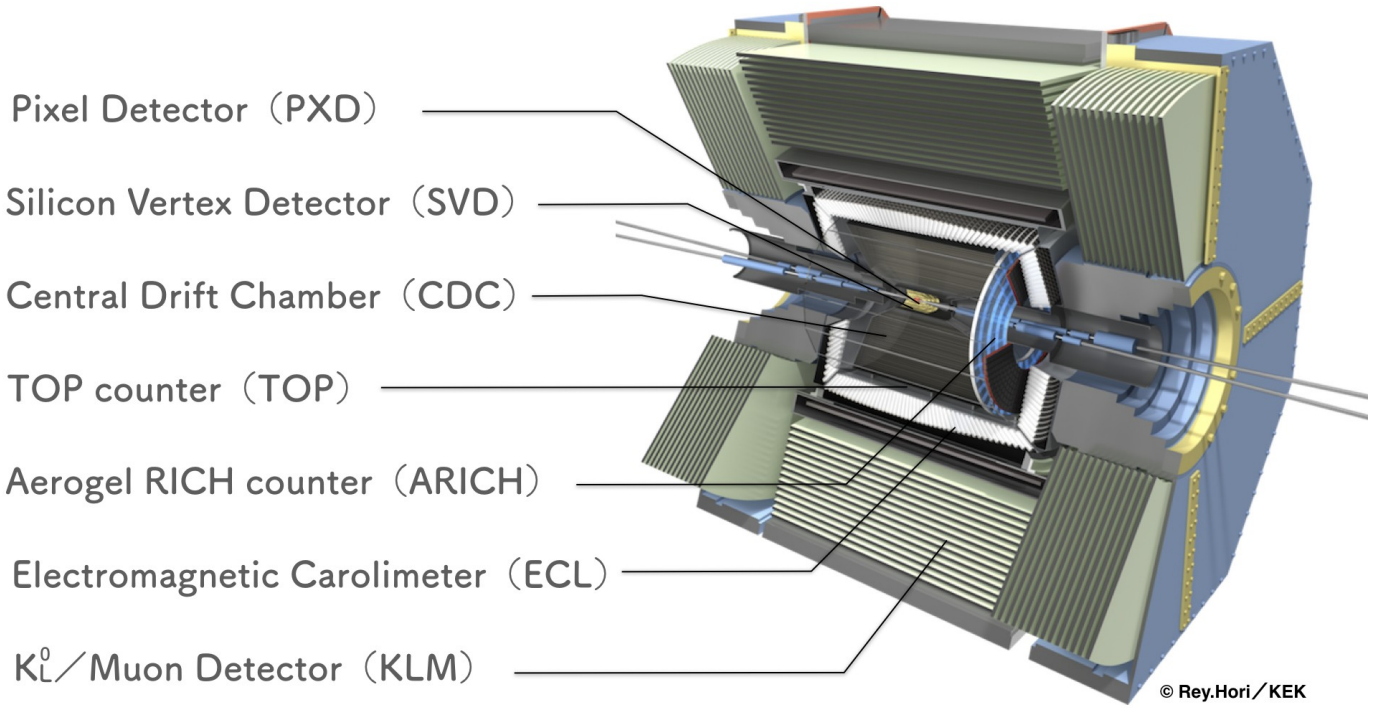
04/04/2026 – Belle II Academy @ Blaubeuren, BW

Disclaimer

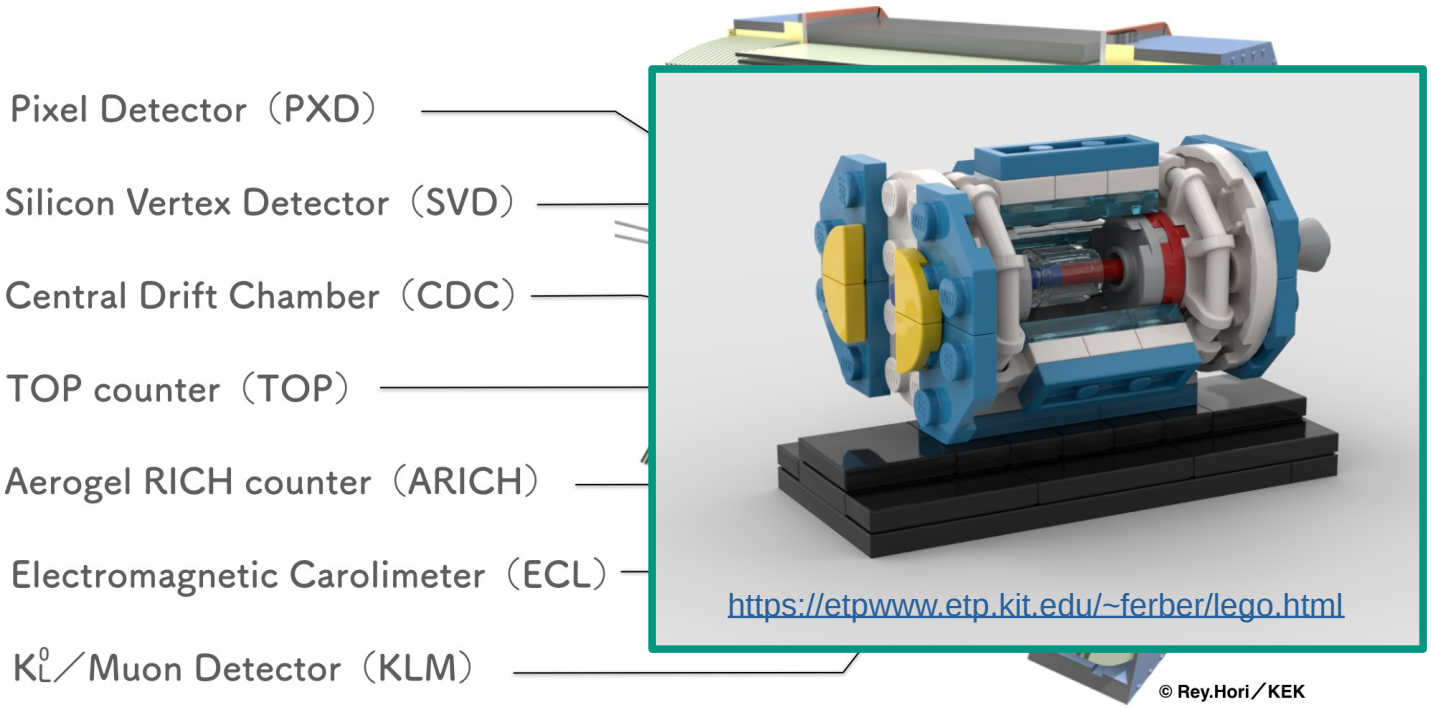
- I have been working with the Belle II software for (too?) many years…
- I might take some things for granted that are actually not at all obvious!
- Please stop me at any time if:
  - I use some jargon you don’t understand
  - If anything is unclear
  - Whenever you have a question

Introduction: software

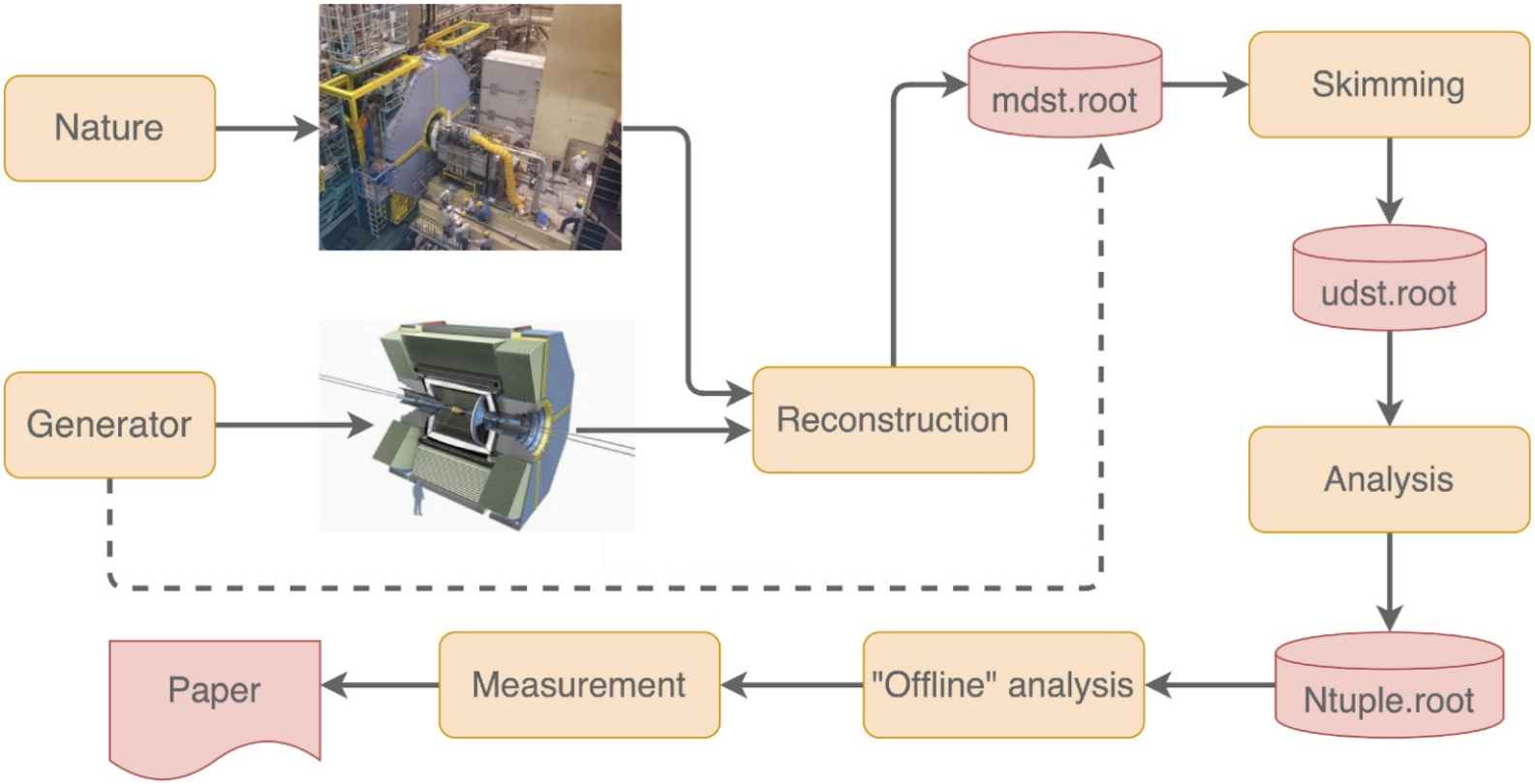
Belle II


Pixel Detector（PXD）
Silicon Vertex Detector（SVD）
Central Drift Chamber（CDC）
TOP counter（TOP）
Aerogel RICH counter（ARICH）
Electromagnetic Carolimeter（ECL）
K^0_L／Muon Detector（KLM）
© Rey.Hori／KEK

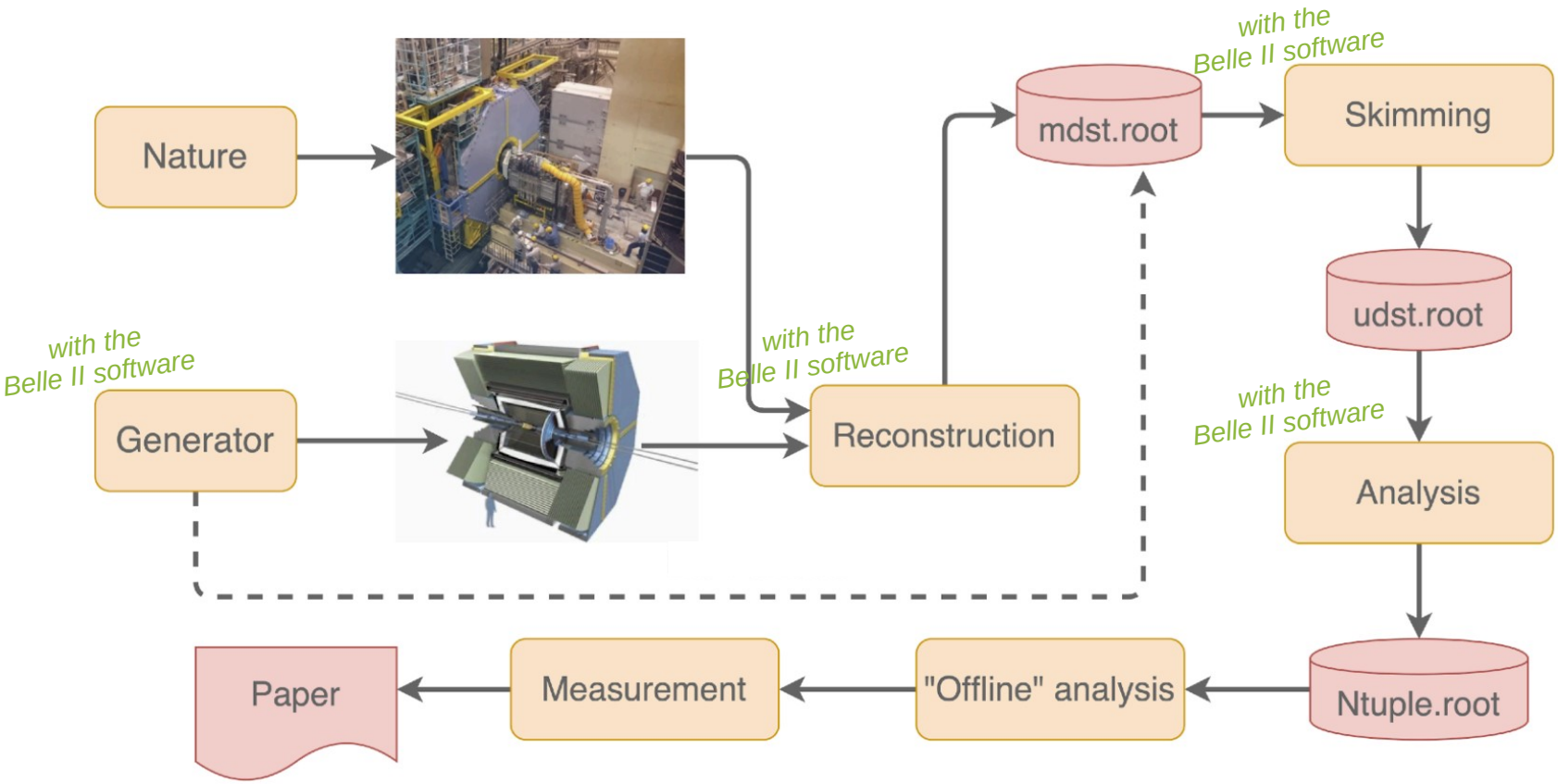
Belle II


Pixel Detector（PXD）
Silicon Vertex Detector（SVD）
Central Drift Chamber（CDC）
TOP counter（TOP）
Aerogel RICH counter（ARICH）
Electromagnetic Carolimeter（ECL）
K^0_L／Muon Detector（KLM）
https://etpwww.etp.kit.edu/~ferber/lego.html
© Rey.Hori／KEK

Belle II workflow



Belle II workflow



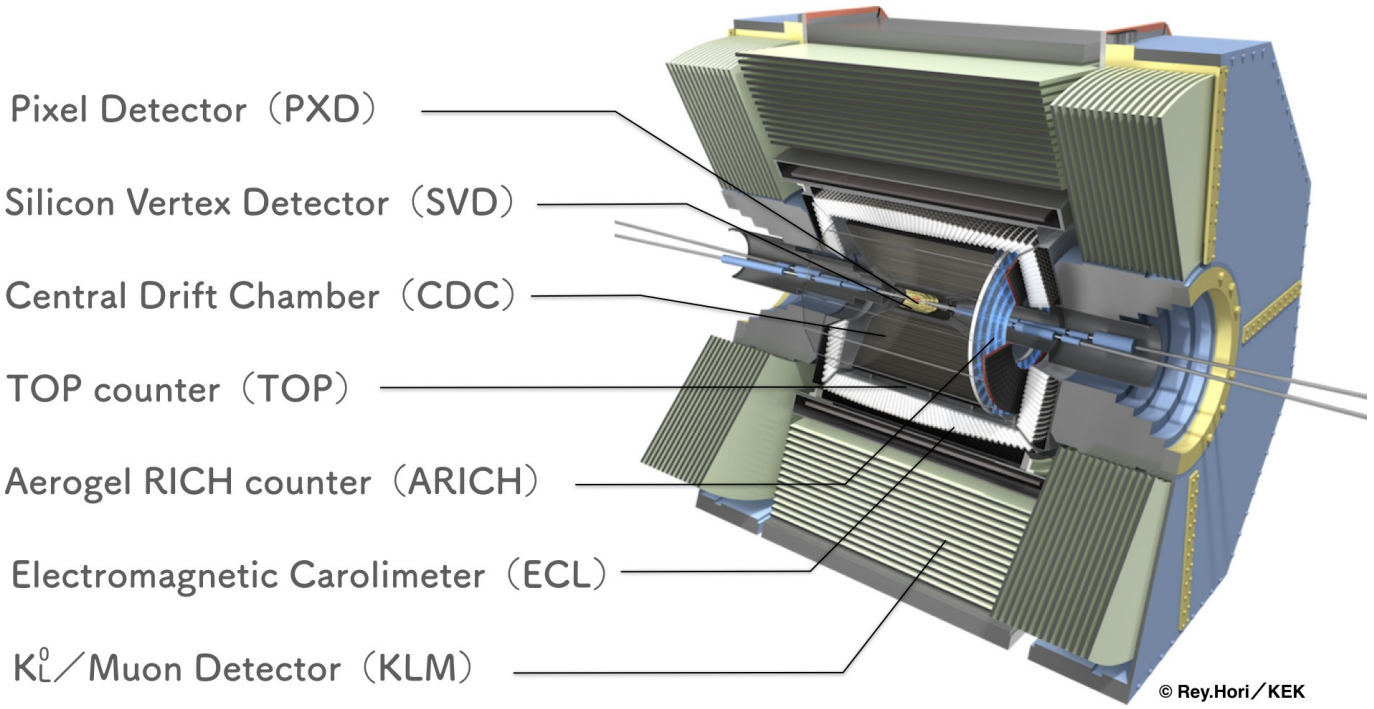
Belle II file formats

- We call our files “dst”
  - dst = data summary table
  - Basically: ROOT files where we store our “custom” objects
- The data for physics analysis are "mdst”
  - mdst = mini data summary table.
  - Same structure of a dst, but with much less information
  - Input to your analysis scripts
- At the end of your analysis chain you will write out a "normal" ROOT file containing “TTree”s, “TBranch”s, or histograms
  - Usually called ntuples
  - Input to your offline analysis

Belle II inputs for analysis

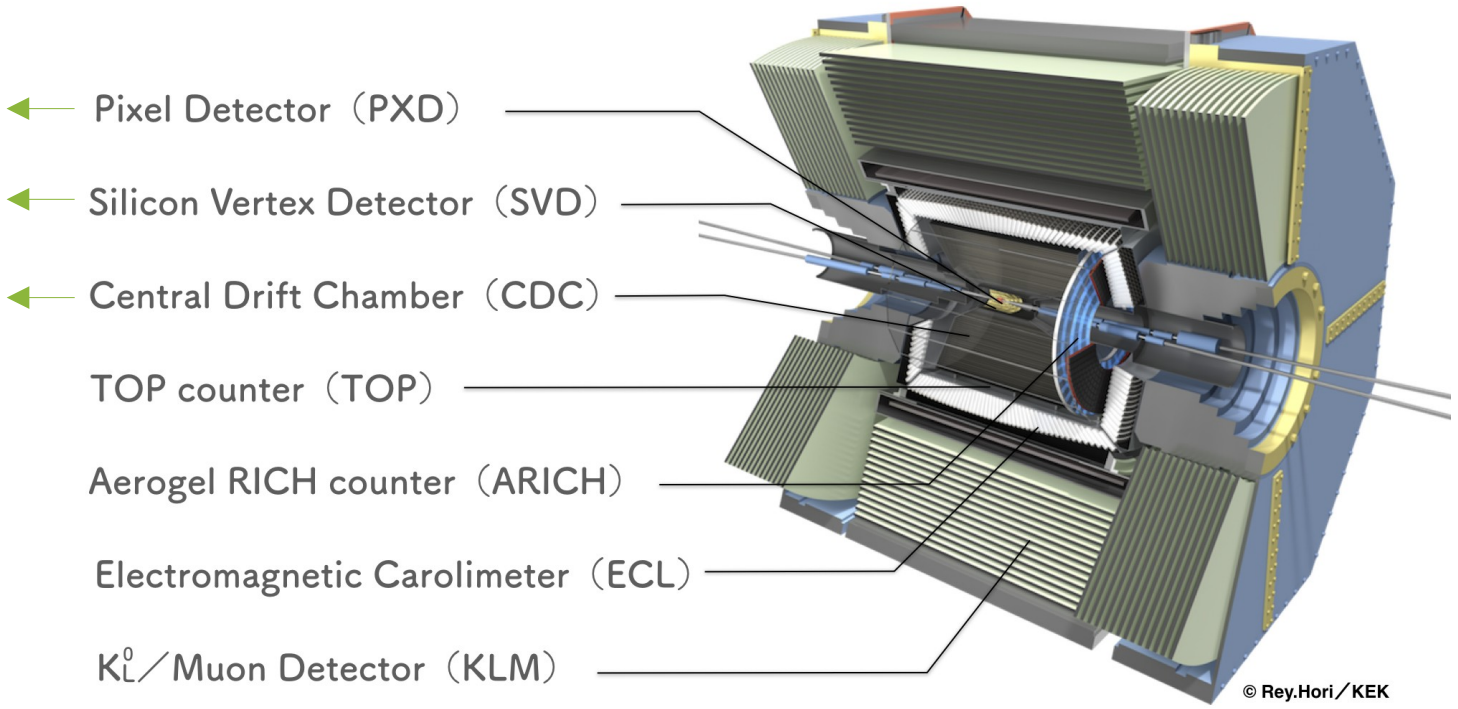
- The content of mdst files is:
  - tracks
  - ECL clusters
  - KLM clusters
  - particle identification (PID) likelihoods
  - MC particles (only in simulated events!)
  - relations between these objects
- These are the building blocks for all the Belle II physics analyses!

Belle II inputs for analysis

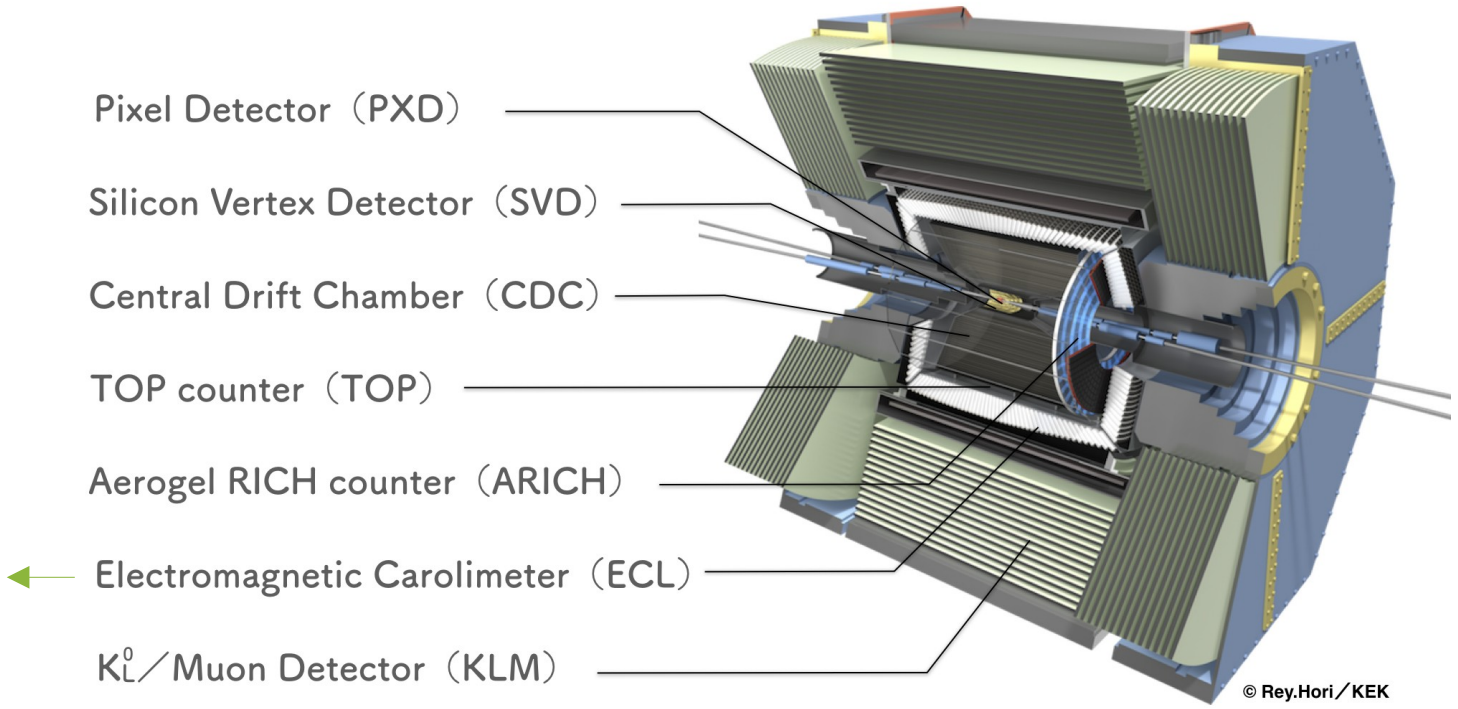


Belle II inputs for analysis

Tracks & vertexing

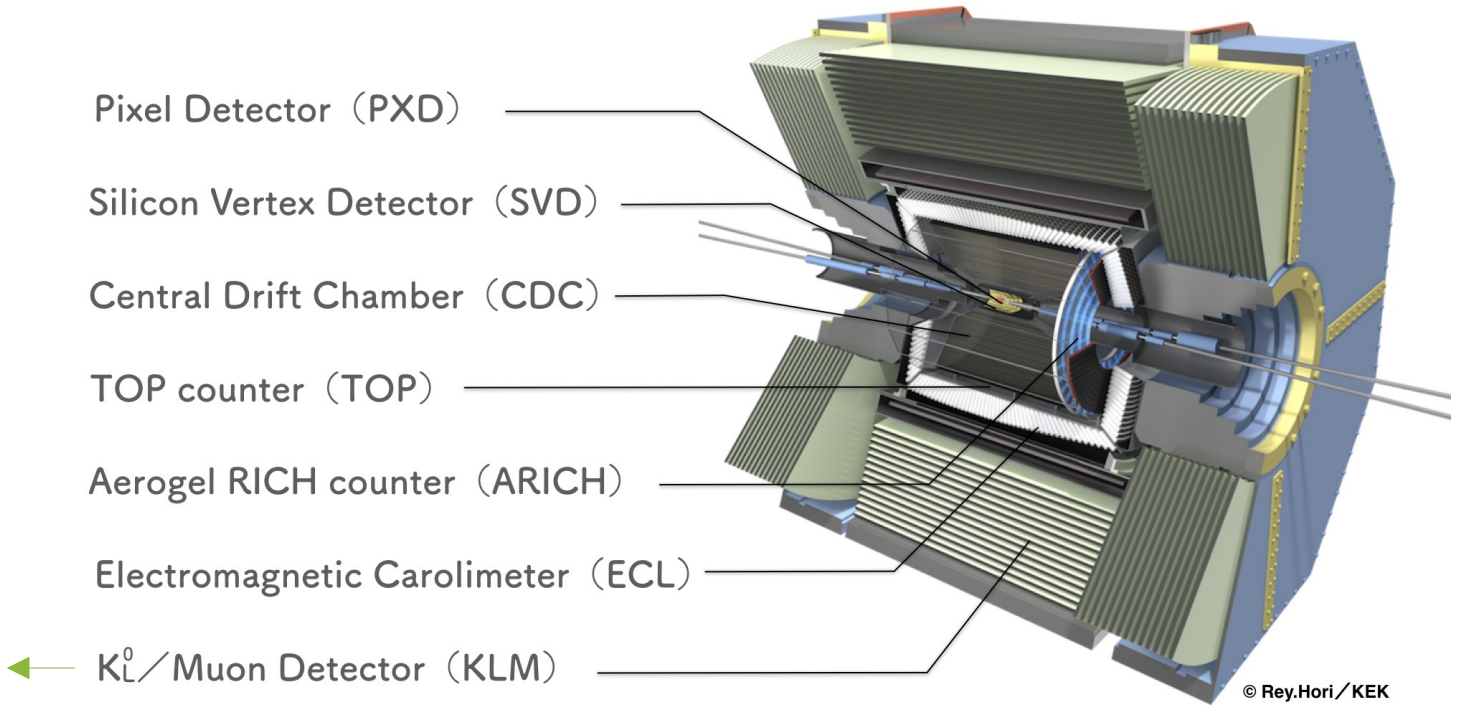


Belle II inputs for analysis



ECL clusters

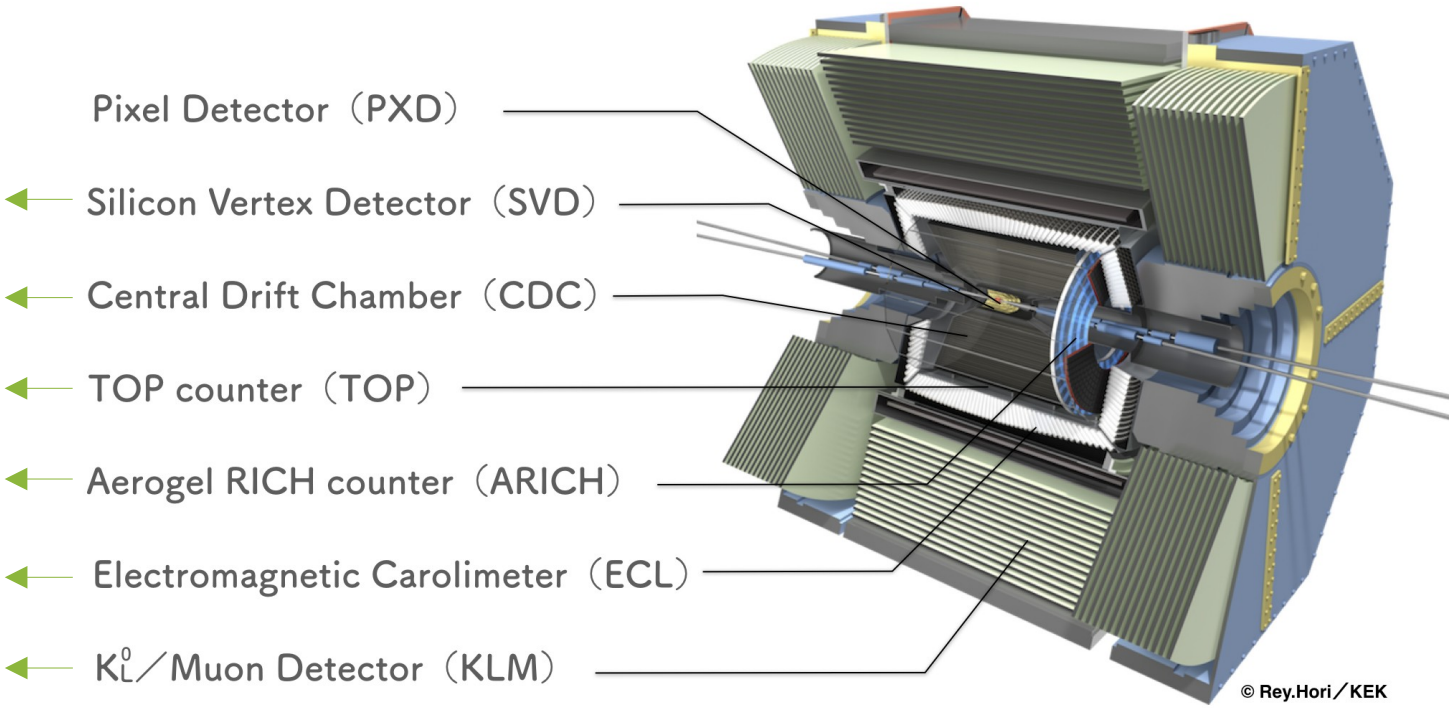
Belle II inputs for analysis



KLM clusters

Belle II inputs for analysis

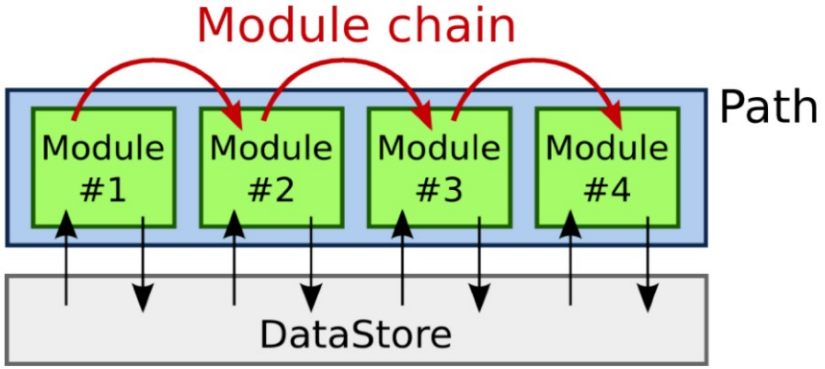
PID likelihoods



basf2: Belle II Analysis Software Framework

basf2: Belle II Analysis Software Framework

- basf2: Belle II Analysis Software Framework
  - arXiv:1809.04299
- More of a software framework than an “analysis framework” (name is historic)
  - It performs the unpacking of raw data, detector simulation, tracking, clustering, …
- The executable is a wrapper for IPython 3, which controls the setup and configuration of a path
- Modules are blocks of code that do a specific unit of data processing
  - They are added to the path calling them inside a so-called “steering file”
- User analysis is performed using the analysis package, with mdst files as input



basf2: few technical details…

- basf2 is mostly written in C++, with Python used to provide an easy user interface
- It is open source and publicly available on GitHub: https://github.com/belle2
  - Licensed under LGPLv3
- The documentation is also publicly available: https://software.belle2.org
  - We also provide a shortcut for the beginners' tutorials: https://training.belle2.org
- Development is done on DESY GitLab: https://gitlab.desy.de/belle2
  - But the access it limited to the Belle II members…
  - (GitHub repository is a mirror of the one hosted on GitLab)

basf2: few technical details…

- We rely on a lot of (open source) software packages not developed by us
  - Compilers, debuggers, Python, etc.
  - HEP-specific software like ROOT, Geant4, XRootD, etc.
    - Including some MC generators like EvtGen, Pythia, MadGraph, etc.
- We “package” all these packages ourselves in the so-called “externals”
  - https://github.com/belle2/externals
  - One version of the externals has a specific version of compilers, Python, ROOT, etc.
  - One or more versions of basf2 are linked to a specific version of the externals

basf2: software releases

- We provide 2 different type of software releases: major releases and light releases
- Major releases:
  - Once a year + frequent patch releases
  - Used in data taking, offline reprocessing, MC production and calibration
  - “Boring” naming scheme: release-08-01-03, release-09-00-01, etc.
- Light releases:
  - One every ~2 months
  - It contains all the features for physics analyses
  - No unpacking or digitization: **only mdst and udst can be processed**
  - “Funny” naming scheme: light-2511-gacrux, light-2403-persian, etc.

basf2: software releases

- We provide 2 different type of software releases: major releases and light releases
- Major releases:
  - Once a year + frequent patch releases
  - Used in data taking and calibration
  - “Boring” naming scheme: release-09-00-01, etc.
- Light releases:
  - One every ~2 months
  - It contains all the features for physics analyses
  - No unpacking or digitization: **only mdst and udst can be processed**
  - “Funny” naming scheme: light-2511-gacrux, light-2403-persian, etc.

We are going to use release-09-00-15

basf2: a steering file

- A simple steering file to generate, simulate and reconstruct 100 generic $B^+ B^-$ events:

```
import basf2 as b2
import generators as ge
import simulation as si
import reconstruction as re
import mdst

main = b2.Path()

main.add_module('EventInfoSetter', evtNumList=100)
main.add_module('Progress')
ge.add_evtgen_generator(path=main, finalstate='charged')
si.add_simulation(path=main)
re.add_reconstruction(path=main)
mdst.add_mdst_output(path=main, filename='mdst.root')

b2.print_path(path=main)
b2.process(path=main)
```

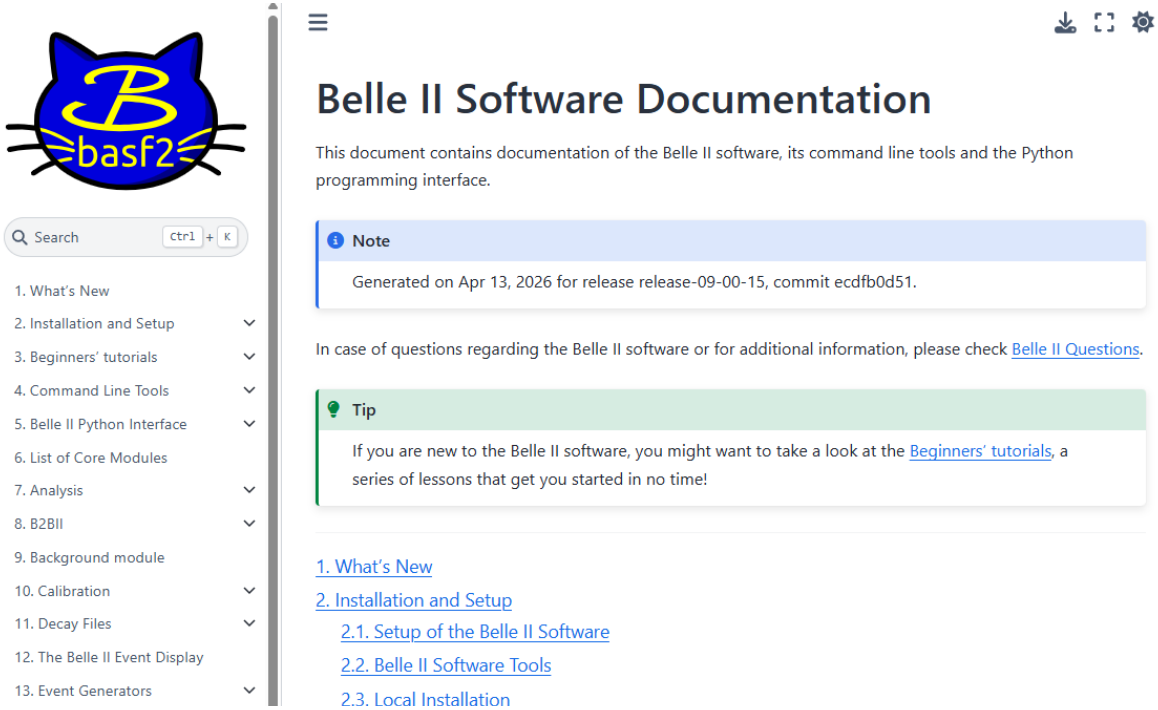
- (It works only with a major release)
- It adds 200+ modules to the path
- It can be executed from the terminal with `basf2 steering.py`

Setup

Reminder…

- Our (very) comprehensive documentation can be found here: https://software.belle2.org/release-09-00-15/sphinx/index.html

your best friend!



Setup

- If you are running at KEKCC, NAF or any other (decent) local cluster: basf2 is already available via the CernVM File System (CVMFS)
- To setup basf2:

```
source /cvmfs/belle.cern.ch/tools/b2setup
fatal: detected dubious ownership in repository at '/cvmfs/belle.cern.ch/tools'
To add an exception for this directory, call:
        git config --global --add safe.directory /cvmfs/belle.cern.ch/tools
Belle II software tools set up at: /cvmfs/belle.cern.ch/tools
```

  - (You can safely ignore the error message, or run the command it suggests to run)
- Then, you can run

```
basf2
command not found: basf2
```

  - We need to setup a release first :)

Setup

- Then, just setup release-09-00-15:

```
b2setup release-09-00-15
Environment setup for release: light-2511-gacrux
Central release directory    : /cvmfs/belle.cern.ch/ubuntu2204/releases/light-2511-
gacrux
```

- You can confirm it works with:

```
basf2 --info
…
  basf2 (Belle II Analysis Software Framework)
Copyright(C) 2010-2024 Members of the Belle II Collaboration
 (See "basf2 --license" for more information.)
           Release release-09-00-15
           Version release-09-00-15
…
```

Setup (better way…)

- Since >1 year we have an alternative (and better) way to setup basf2 using b2venv
- b2venv allows to correctly setup basf2 AND a Python virtual environment together
  - Creating a virtual environment in the usual way does not work well with basf2…
- See the full documentation at https://software.belle2.org/development/sphinx/build/tools_doc/b2venv.html
- TL;DR to have basf2 AND a correctly setup virtual environment:

```
source /cvmfs/belle.cern.ch/tools/b2setup
b2venv release-09-00-15
source .venv/bin/activate
```

Setup

- Your environment is ready to go: you can now use basf2!
- If you type `b2` in the terminal and then start pressing TAB, you will see all the available commands we provide with basf2
- Some useful commands are:
  - `b2help-*`
  - `b2file-*`
  - `b2code-style-check / b2code-style-fix`
    - You can always use the `--help` option to see the help message!

More on useful commands

- Help at the command line: `pydoc3 basf2.Path`
- List all the modular analysis convenience functions: `basf2 modularAnalysis.py`
- List the basf2 modules: `b2help-modules`
- List the basf2 analysis variables: `b2help-variables`
- List the basf2 particle names: `b2help-particles`
  - We use the same conventions used by ROOT, PDG, EvtGen, etc.

Before starting…

- The GitLab repository below contains the steering files that solve the exercises from this workshop: https://gitlab.desy.de/belleii-academy-2026/software-part-ii
- The repository is currently (almost) empty: the steering files will be added one by one, at the end of each corresponding exercise ;)

Let’s start!

A steering file

- All the steering files have a common structure:

```
import basf2 as b2
import generators as ge


main = b2.Path()

main.add_module('EventInfoSetter')
main.add_module('Progress')
ge.add_evtgen_generator(
  finalstate='charged',
  path=main
)
main.add_module('PrintMCParticles')

...

b2.print_path(path=main)

b2.process(path=main)

print(b2.statistics)
```

A steering file

- All the steering files have a common structure:

```
import basf2 as b2
import generators as ge


main = b2.Path()

main.add_module('EventInfoSetter')
main.add_module('Progress')
ge.add_evtgen_generator(
  finalstate='charged',
  path=main
)
main.add_module('PrintMCParticles')

...

b2.print_path(path=main)

b2.process(path=main)

print(b2.statistics)
```

packages to import

A steering file

- All the steering files have a common structure:

```
import basf2 as b2
import generators as ge


main = b2.Path()

main.add_module('EventInfoSetter')
main.add_module('Progress')
ge.add_evtgen_generator(
  finalstate='charged',
  path=main
)
main.add_module('PrintMCParticles')

...

b2.print_path(path=main)

b2.process(path=main)

print(b2.statistics)
```

packages to import

the path is created

A steering file

- All the steering files have a common structure:

```
import basf2 as b2
import generators as ge


main = b2.Path()

main.add_module('EventInfoSetter')
main.add_module('Progress')
ge.add_evtgen_generator(
  finalstate='charged',
  path=main
)
main.add_module('PrintMCParticles')

...

b2.print_path(path=main)

b2.process(path=main)

print(b2.statistics)
```

packages to import

the path is created

modules are added to the path:
- directly
- via convenient functions

A steering file

- All the steering files have a common structure:

```
import basf2 as b2
import generators as ge


main = b2.Path()

main.add_module('EventInfoSetter')
main.add_module('Progress')
ge.add_evtgen_generator(
  finalstate='charged',
  path=main
)
main.add_module('PrintMCParticles')

...

b2.print_path(path=main)

b2.process(path=main)

print(b2.statistics)
```

packages to import

the path is created

modules are added to the path:
- directly
- via convenient functions

the list of modules is printed (optional)

A steering file

- All the steering files have a common structure:

```
import basf2 as b2
import generators as ge


main = b2.Path()

main.add_module('EventInfoSetter')
main.add_module('Progress')
ge.add_evtgen_generator(
  finalstate='charged',
  path=main
)
main.add_module('PrintMCParticles')

...

b2.print_path(path=main)

b2.process(path=main)

print(b2.statistics)
```

packages to import

the path is created

modules are added to the path:
- directly
- via convenient functions

the list of modules is printed (optional)

the path is executed

A steering file

- All the steering files have a common structure:

```
import basf2 as b2
import generators as ge


main = b2.Path()

main.add_module('EventInfoSetter')
main.add_module('Progress')
ge.add_evtgen_generator(
  finalstate='charged',
  path=main
)
main.add_module('PrintMCParticles')

...

b2.print_path(path=main)

b2.process(path=main)

print(b2.statistics)
```

packages to import

the path is created

modules are added to the path:
- directly
- via convenient functions

the list of modules is printed (optional)

the path is executed

some statistics are printed (optional)

Exercise

- Create a Python file where you copy the steering file from the previous slide and execute it with basf2:
  - `basf2 steering_file.py -n 10`
  - Run `basf2 --help` to see all the available options
    - In this case: `-n 10` means “process 10 events”
- Execute the command multiple times: is the output always the same?

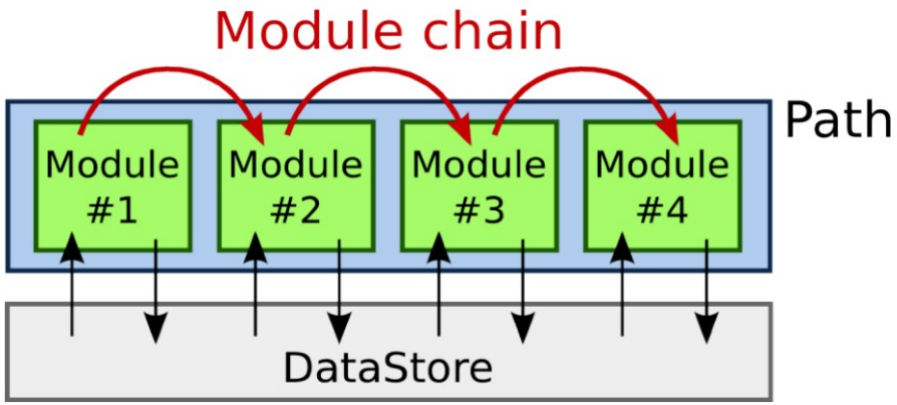
Path, modules and DataStore

- Almost at the beginning of the basf2 execution, you can find the output of `basf2.print_path(path=main)`:

```
[INFO] Modules and parameter settings in the path:
  1. EventInfoSetter
  2. Progress
  3. EvtGenInput
      userDECFile=/cvmfs/belle.cern.ch/el9/releases/release-09-00-15/data/generators/
evtgen/charged.dec
  4. PrintMCParticles
```

- Let’s understand what this means

Path, modules and DataStore



- A module is a block of code that performs a specific unit of data processing
- Multiple modules can be added to a path and run sequentially
  - A path is basically a list of modules
- The DataStore is a pool of objects valid within an event
- A module can add objects to the DataStore or retrieve them if already existing
- At the end of an event, the content of the DataStore is either (partially) stored in an output file or deleted

Path, modules and DataStore

- From our example:
  - EventInfoSetter
    - Read from the DataStore: -
    - Add to the DataStore: EventMetaData
  - Progress
    - Read from the DataStore: -
    - Add to the DataStore: -
  - EvtGenInput
    - Read from the DataStore: -
    - Add to the DataStore: array of MCParticles
  - PrintMCParticles
    - Read from the DataStore: array of MCParticles
    - Add to the DataStore: StringWrapper object

Path, modules and DataStore

- We can have in the DataStore:
  - Arrays of the objects from the same class
    - We call them StoreArray (in C++) or PyStoreArray (in Python)
    - MCParticles are stored as a StoreArray
      - `StoreArray<MCParticle>` or `PyStoreArray("MCParticles")` (note the s)
  - Single object of a given class
    - We call them StoreObjPtr (in C++) or PyStoreObj (in Python)
    - EventMetaData is stored as a StoreObjPtr
      - `StoreObjPtr<EventMetaData>` or `PyStoreObj("EventMetaData")`

Path, modules and DataStore

- A module is a class that can interact with the DataStore and has some specialized methods:
  - constructor:
    - It is called when the module is added to the path via `add_module`
    - Called only once (before `basf2.process`)
  - initialize:
    - This is called inside the basf2 process during the initialization phase
    - Called only once
  - event:
    - This is called during the event processing
    - Called once per event (multiple times in the process)
  - terminate:
    - Called at the end of the whole basf2 process to do some cleanup or print statistics (if necessary)
    - Called only once

Python modules

- In our documentation, we have a tutorial about how to write a Python module: https://software.belle2.org/release-09-00-15/sphinx/online_book/basf2/python_module.html
- Let’s quickly see here how to write it

Python modules

```
import basf2 as b2


class MyModule(b2.Module):

  def __init__(self):
    super().__init__()
    b2.B2INFO('Constructor')

  def initialize(self):
    b2.B2INFO('Initialize')

  def event(self):
    b2.B2INFO('Event')

  def terminate(self):
    b2.B2INFO('Terminate')


main = b2.Path()
main.add_module('EventInfoSetter')
main.add_module(MyModule())
b2.process(main)
print(b2.statistics)
```

Python modules

- Let's try to interact with the DataStore
- The key lines of code are:

```
import ROOT
emd = ROOT.Belle2.PyStoreObj('EventMetaData')
mcps = ROOT.Belle2.PyStoreArray('MCParticles')
```

- You should always check if a PyStoreObj is valid before accessing it:

```
if emd.isValid():
  # Treat emd as an object of type EventMetaData
  print(emd.getEvent())
```

- You can loop over a PyStoreArray in a fully pythonic way:

```
for mcp in mcps:
  # Treat mcp as an object of type MCParticle
  print(mcp.getLifetime())
```

Exercise (I)

- Using this steering file as baseline:

```
import basf2 as b2
import generators as ge

class MyModule(b2.Module):
  ...

b2.set_random_seed('academy')

main = b2.Path()
main.add_module('EventInfoSetter')
ge.add_evtgen_generator(
  finalstate='charged',
  path=main
)
main.add_module(MyModule())

b2.process(main)
print(b2.statistics)
```

write a module that counts how many charged pions are generated in each event and in total in the whole process

Exercise (I)

- You can find the Doxygen documentation of the EventMetaData and MCParticle classes at the following links:
  - https://software.belle2.org/release-09-00-15/doxygen/classBelle2_1_1EventMetaData.html
  - https://software.belle2.org/release-09-00-15/doxygen/classBelle2_1_1MCParticle.html

Exercise (II)

- Include again the module PrintMCParticles to the path and check its output on screen
- Try to replicate the output of PrintMCParticles (in a simplified way) in your Python module by printing the FULL decay tree of the Upsilon(4S), including the momentum of particles
  - You need to write a recursive function for that :)
  - You can use the pdg package from basf2 to render the particle names:

```
import pdg
pdg.to_name(211)  # this is a pi+
```

    - https://software.belle2.org/release-09-00-15/sphinx/framework/doc/modules/pdg.html
  - `MCParticle::getMomentum()` returns a `ROOT::Math::XYZVector` object:
    - `pydoc3 ROOT.Math.XYZVector` will show the available methods ;)

ROOT, Python and black magic

- ROOT allows you to use any C++ library from Python, profiting also of the just-in-time (JIT) compilation
- How is this done? Black magic…
  - Relevant ROOT documentation is here: https://root.cern/manual/python/
  - Technical details can be found here: https://cppyy.readthedocs.io/en/latest/

ROOT, Python and black magic

```
import ROOT

cpp = r'''
double add(double a, double b) {
  return a + b;
}

class MyClass {
public:
  double m_a = 0.;
  double m_b = 0.;
  void multiply(double c) { m_a *= c; m_b *= c;};
}
'''

ROOT.gInterpreter.ProcessLine(cpp)

c = ROOT.add(3.5, 2.1)
print(c)

my_class = ROOT.MyClass()
my_class.m_a = 3.5
my_class.m_b = 2.1
my_class.multiply(2)
print(my_class.m_a, my_class.m_b)
```

ROOT, Python and black magic

```
import ROOT

cpp = r'''
double add(double a, double b) {
  return a + b;
}

class MyClass {
public:
  double m_a = 0.;
  double m_b = 0.;
  void multiply(double c) { m_a *= c; m_b *= c;};
}
'''

ROOT.gInterpreter.ProcessLine(cpp)

c = ROOT.add(3.5, 2.1)
print(c)

my_class = ROOT.MyClass()
my_class.m_a = 3.5
my_class.m_b = 2.1
my_class.multiply(2)
print(my_class.m_a, my_class.m_b)
```

define a C++ function and a C++ class in a Python (raw) string

ROOT, Python and black magic

```
import ROOT

cpp = r'''
double add(double a, double b) {
  return a + b;
}

class MyClass {
public:
  double m_a = 0.;
  double m_b = 0.;
  void multiply(double c) { m_a *= c; m_b *= c;};
}
'''

ROOT.gInterpreter.ProcessLine(cpp)

c = ROOT.add(3.5, 2.1)
print(c)

my_class = ROOT.MyClass()
my_class.m_a = 3.5
my_class.m_b = 2.1
my_class.multiply(2)
print(my_class.m_a, my_class.m_b)
```

define a C++ function and a C++ class in a Python (raw) string

process the string with this function

ROOT, Python and black magic

```
import ROOT

cpp = r'''
double add(double a, double b) {
  return a + b;
}

class MyClass {
public:
  double m_a = 0.;
  double m_b = 0.;
  void multiply(double c) { m_a *= c; m_b *= c;};
}
'''

ROOT.gInterpreter.ProcessLine(cpp)

c = ROOT.add(3.5, 2.1)
print(c)

my_class = ROOT.MyClass()
my_class.m_a = 3.5
my_class.m_b = 2.1
my_class.multiply(2)
print(my_class.m_a, my_class.m_b)
```

define a C++ function and a C++ class in a Python (raw) string

process the string with this function

the function is available in Python!

ROOT, Python and black magic

```
import ROOT

cpp = r'''
double add(double a, double b) {
  return a + b;
}

class MyClass {
public:
  double m_a = 0.;
  double m_b = 0.;
  void multiply(double c) { m_a *= c; m_b *= c;};
}
'''

ROOT.gInterpreter.ProcessLine(cpp)

c = ROOT.add(3.5, 2.1)
print(c)

my_class = ROOT.MyClass()
my_class.m_a = 3.5
my_class.m_b = 2.1
my_class.multiply(2)
print(my_class.m_a, my_class.m_b)
```

define a C++ function and a C++ class in a Python (raw) string

process the string with this function

the function is available in Python!

the class is available in Python!

Exercise

- Try this black magic yourself!

ROOT, Python and black magic… and basf2!

- Can I use this black magic passing together with basf2 classes?

ROOT, Python and black magic… and basf2!

- Can I use this black magic passing together with basf2 classes?
- Yes!

```
cpp = r'''
#include <mdst/dataobjects/MCParticle.h>
#include <cmath>

namespace Belle2 {
  double getMomentum(const MCParticle* mcParticle) {
    // mcParticle is a pointer:
    // we access its methods using the operator ->
    return std::sqrt(mcParticle->getMomentum().Mag2());
  }
} // namespace Belle2
'''
...

import ROOT
ROOT.gInterpreter.ProcessLine(cpp)
self.mcps = ROOT.Belle2.PyStoreArray('MCParticles')
for mcp in self.mcps:
  # mcp is a pointer to an MCParticle (MCParticle*)
  print(ROOT.Belle2.getMomentum(mcp))
```

ROOT, Python and black magic… and basf2!

- Can I use this black magic passing together with basf2 classes?
- Yes!

```
cpp = r'''
#include <mdst/dataobjects/MCParticle.h>
#include <cmath>

namespace Belle2 {
  double getMomentum(const MCParticle* mcParticle) {
    // mcParticle is a pointer:
    // we access its methods using the operator ->
    return std::sqrt(mcParticle->getMomentum().Mag2());
  }
} // namespace Belle2
'''
...

import ROOT
ROOT.gInterpreter.ProcessLine(cpp)
self.mcps = ROOT.Belle2.PyStoreArray('MCParticles')
for mcp in self.mcps:
  # mcp is a pointer to an MCParticle (MCParticle*)
  print(ROOT.Belle2.getMomentum(mcp))
```

class headers must be included

ROOT, Python and black magic… and basf2!

- Can I use this black magic passing together with basf2 classes?
- Yes!

```
cpp = r'''
#include <mdst/dataobjects/MCParticle.h>
#include <cmath>

namespace Belle2 {
  double getMomentum(const MCParticle* mcParticle) {
    // mcParticle is a pointer:
    // we access its methods using the operator ->
    return std::sqrt(mcParticle->getMomentum().Mag2());
  }
} // namespace Belle2
'''
...

import ROOT
ROOT.gInterpreter.ProcessLine(cpp)
self.mcps = ROOT.Belle2.PyStoreArray('MCParticles')
for mcp in self.mcps:
  # mcp is a pointer to an MCParticle (MCParticle*)
  print(ROOT.Belle2.getMomentum(mcp))
```

class headers must be included

better enclose the code under the Belle2 namespace

ROOT, Python and black magic… and basf2!

- Can I use this black magic passing together with basf2 classes?
- Yes!

```
cpp = r'''
#include <mdst/dataobjects/MCParticle.h>
#include <cmath>

namespace Belle2 {
  double getMomentum(const MCParticle* mcParticle) {
    // mcParticle is a pointer:
    // we access its methods using the operator ->
    return std::sqrt(mcParticle->getMomentum().Mag2());
  }
} // namespace Belle2
'''
...

import ROOT
ROOT.gInterpreter.ProcessLine(cpp)
self.mcps = ROOT.Belle2.PyStoreArray('MCParticles')
for mcp in self.mcps:
  # mcp is a pointer to an MCParticle (MCParticle*)
  print(ROOT.Belle2.getMomentum(mcp))
```

class headers must be included

better enclose the code under the Belle2 namespace

voila'!

Exercise (I)

- Try to define some simple functions which accept at least a MCParticle (pointer) as argument and use them in your Python module

Exercise (II)

- Go back to your module that replicates the functionalities of PrintMCParticles and try to replace the core logic with the Python bindings from ROOT
- Tips: you can use the std::string class to create and manipulate strings:

```
#include <string>
std::string s;
s = "Hello " + std::to_string(42); // s contains "Hello 42"
```

Write your own variables

- In the analysis package, we provide a lot of variables to store information in your ntuples or apply cuts when running your steering file
- The full list of variables is here: https://software.belle2.org/development/sphinx/analysis/doc/Variables.html#variables-by-group
- Of course, despite we have a lot of variables, the list is not exhaustive: you might need a variable for your analysis which we do not provide in basf2!
- There are two solutions for this:
  - The variable is potentially useful for others: add it to basf2!
  - The variable is too specific to be useful also for others: you keep it in your analysis repository

Write your own variables

- We have a tutorial about how to write a variable: https://software.belle2.org/development/sphinx/analysis/doc/Variables.html#writing-your-own-variable
- What if... we define a new variable using this trick? Does it work?

Write your own variables

- We have a tutorial about how to write a variable: https://software.belle2.org/development/sphinx/analysis/doc/Variables.html#writing-your-own-variable
- What if... we define a new variable using this trick? Does it work?
- Yes!

Write your own variables

```
from variables import variables as vm
import ROOT

cpp = '''
#include <analysis/VariableManager/Manager.h>
#include <analysis/dataobjects/Particle.h>

namespace Belle2::Variable {
  double myVariable(const Particle* p) {
    return 0.;
  }
  VARIABLE_GROUP("My Variables");
  REGISTER_VARIABLE(
    "myVar",    // How you call the variable in your steering
    myVariable, // The name of the C++ function
    "A nice description."
  );
} // Belle2::Variable namespace
'''
ROOT.gInterpreter.ProcessLine(cpp)

print(vm.getVariable('myVar').name)
print(vm.getVariable('myVar').description)
```

you always need to include (at least) these two headers

Write your own variables

- This is a 100% portable solution to write your own variables:
  - If it works with a given release on a certain OS, it will work also on another OS using the same release
  - If you need to submit a job on grid including your own variables, this “trick” allows you not to specify the OS when submitting your project

Exercise

- Try to replicate the functionalities of an existing variable and verify that your variable returns the same values as the “official” variable
- You can use this “skeleton” steering file to create an ntuple file starting from MCParticles: https://gitlab.desy.de/belleii-academy-2026/software-part-ii/-/blob/main/my_variable_steering_skeleton.py
  - When you use ma.fillParticleListfromMC, basf2 creates a Particle with the same kinematics of the underlying MCParticle

What’s next?

- We just scraped the surface of the possibilities of what you can do with basf2…
  - E.g.: we did not cover how to check for the relations between objects and loop over the related objects
- Python modules are great for prototyping ideas, debugging purposes or tests, but: we can’t include them in simulation, reconstruction or HLT path for technical reasons
  - You can include them in your analysis path: make sure their execution time and memory consumption are under control!
- If you want to deploy your Python module in basf2 to be used in the official reconstruction… You need to rewrite it in C++ :D