basf2 Hands-on

From event generation to ntuple — one hour through the Belle II analysis chain

Belle II experiment · Analysis software for beginners

Speaker: Junhao Yin

Outline

#	Topic	Practice
0	Opening: what we are going to get running	
1	**Part 1** Environment & toolchain	Exercise 1
2	**Part 2** Five concepts worth knowing	Exercise 2
3	**Part 3** Hands-on A: MC generation	Exercise 3
4	**Part 4** Hands-on B: the analysis script	Exercise 4
5	**Part 5** Hands-on C: ntuple → plots	
6	**Part 6** Going further: B2BII / gbasf2	
7	**Part 7** AI agents in the loop	
8	Summary + Q&A	

> Only two scripts today: one generates MC, one does the analysis. The goal is to get both running in your hands.

Ground rules

- Account: **KEKCC** or **IHEP-farm**
- The scripts live in `analysis/examples`; just run `basf2 xxx.py`
- Convention: lines starting with `$` are shell commands, everything else is python
- **No C++ required** — a little python is enough: import, call a function, change a parameter
- Every part ends with a 2–6 minute exercise; ask anytime, or later on https://questions.belle2.org
- All material (slides + runnable scripts) is in today's course directory

Scripts: from $BELLE2_RELEASE_DIR/ or checkout from Git: https://gitlab.desy.de/belle2/software/basf2

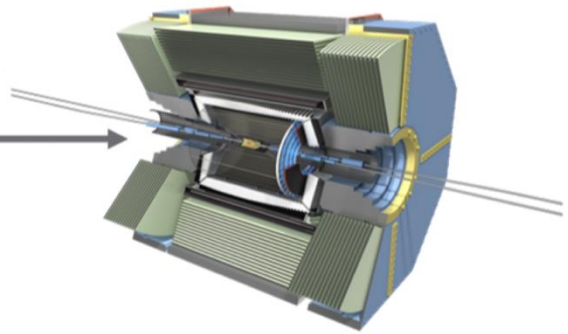
> We are not trying to understand everything — just to get two commands running end to end.

The big picture

Nature



PYTHIA

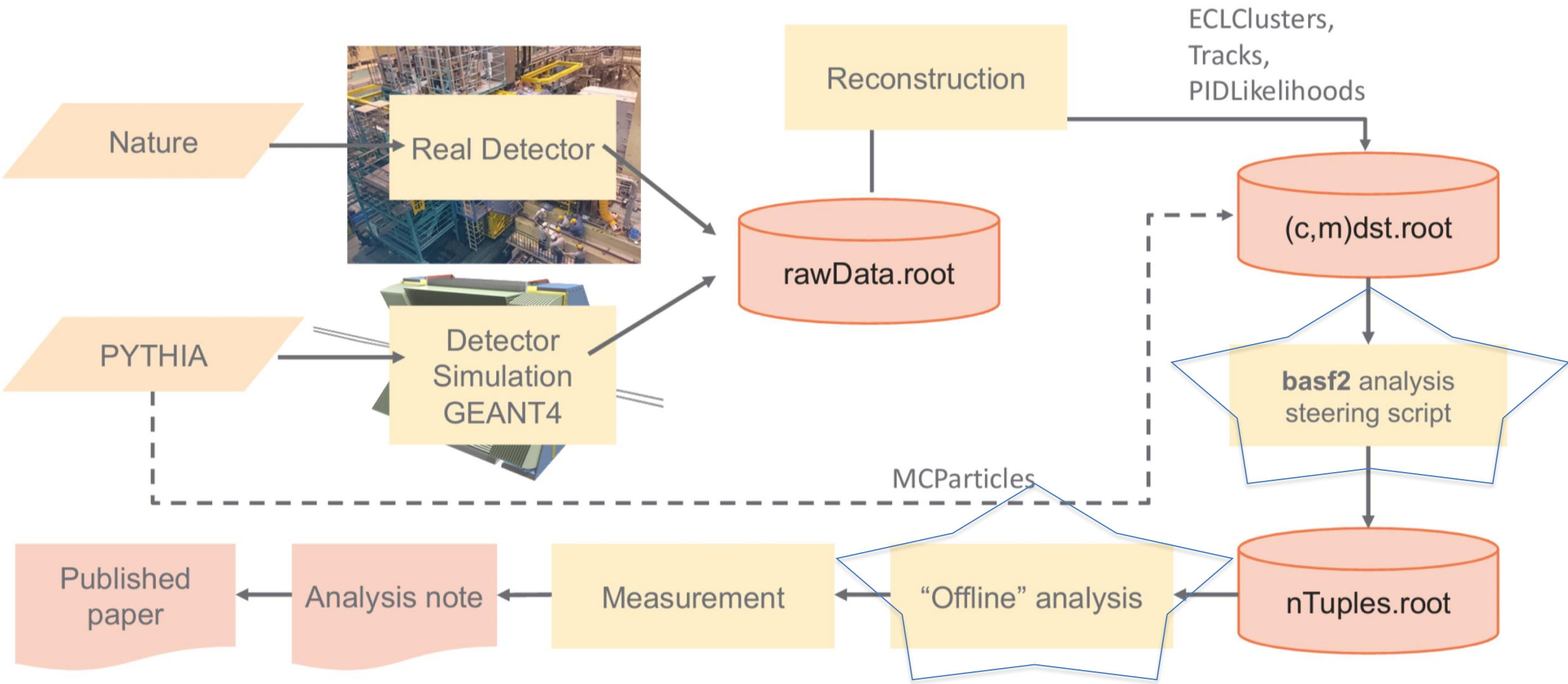


Published paper

What do you need?

Data storage → root (trees) files
Data processing → C++ code
Scripting → Python

The big picture



Part 1

Environment & toolchain

- What basf2 is, and what it is not
- One command to set up the environment; release vs light
- Where to look when something breaks

What basf2 is

- The official Belle II analysis software framework
- You write **python**; the script describes a **data flow** (`b2.create_path()`), and the framework handles scheduling, parallelism and I/O
- Not "a pile of ROOT macros": modules exchange objects through the **DataStore**, not by reading files by hand
- The same script runs locally and on the grid: `basf2 script.py` vs `gbasf2 script.py` differ only in the executor
- The design goal is exactly **easy to read and use** — reading an analysis script should tell you what it does

> One sentence: an analysis = a data flow + variables read off that flow.

Step one: set up the environment

```
# log in to KEKCC
ssh <username>@login.cc.kek.jp
ssh <username>@sshcc1.kek.jp

# once per new shell: set up the environment
source /cvmfs/belle.cern.ch/tools/b2setup release-09-00-15     # full release
# or
source /cvmfs/belle.cern.ch/tools/b2setup light-2505-deimos    # light release

# not sure what exists / what is recommended?
b2setup --help          # list all available releases
b2help-releases         # show the officially recommended release

# check that the environment is sane
basf2 --info
```

One setup line sets every `BELLE2_*` variable and sources ROOT and Geant4 along the way.

A new terminal means running setup again.

release or light?

	full release release-AA-BB-CC	**light release** light-YYMM-<name>
Contains	all packages: analysis, mdst, skim, pxd, svd, trg, simulation, reconstruction …	only analysis-related packages: analysis, mdst, skim, b2bii, mva …
Can do	simulation + reconstruction + analysis: **produce your own MC**	read mdst / udst and run high-level analysis
Cannot do	large on disk, slow to start	**no simulation / reconstruction**; crashes on cdst
Support policy	latest patch of the last two major releases	the last two light releases
Used today for	Hands-on A (MC generation)	Hands-on B / C (analysis)

> Before generating MC, make sure you are on a full release — otherwise the import fails immediately.

Check the environment: basf2 --info

```
$basf2 --info
Release:            release-09-00-15
Release dir:        /cvmfs/belle.cern.ch/el9/releases/release-09-00-15
Local release dir:                      <- empty = you have not patched basf2
Externals:          /cvmfs/belle.cern.ch/el9/externals/v02-02-04
Python:             3.11
ROOT:               6.28/04

# before running anything, check these three: release / externals / ROOT
```

About 90% of "the environment is broken" problems are visible in this one screen.

Where things live

- Copy examples from `$BELLE2_RELEASE_DIR/<package>/examples/` — today mostly `analysis/examples/tutorials/`
- Find modules in `$BELLE2_RELEASE_DIR/<package>/modules/` — but **you rarely call modules directly**
- Find ready-made functions in `$BELLE2_RELEASE_DIR/<package>/scripts/`, e.g. `stdCharged.stdPi()`, `stdPi0s.stdPi0s()`, `generators.add_evtgen_generator()`
- Look up a variable: `b2help-variables` (all) or `b2help-variables Mbc` (one)
- Look up a particle name / PDG code: `b2help-particles`
- Documentation: https://software.belle2.org — sphinx, searchable, genuinely good
- Ask: https://questions.belle2.org — **anything, not just analysis**; search first, then post
- Source: https://gitlab.desy.de/belle2/software/basf2

> Suggested order: search sphinx / questions, then read examples, and only then open the source.

Exercise 1 · 3 minutes

Walk through the environment

1. Set up a light release, run `basf2 --info`, note the ROOT version

2. `b2help-variables`: do you see `Mbc`, `deltaE`, `chiProb`?
   What do you see with `b2help-` ?

3. `ls $BELLE2_RELEASE_DIR/analysis/examples/tutorials/` — what do the official tutorials cover?

`b2help-releases` with no argument prints the officially recommended release — use that from now on. `release-09-00-15` / `light-2505-deimos` in these slides are just examples.

Part 2

Five concepts worth knowing

- What mdst / cdst / udst actually are
- Why a plain ROOT macro cannot read an mdst
- The ParticleList model and the skeleton of an analysis script

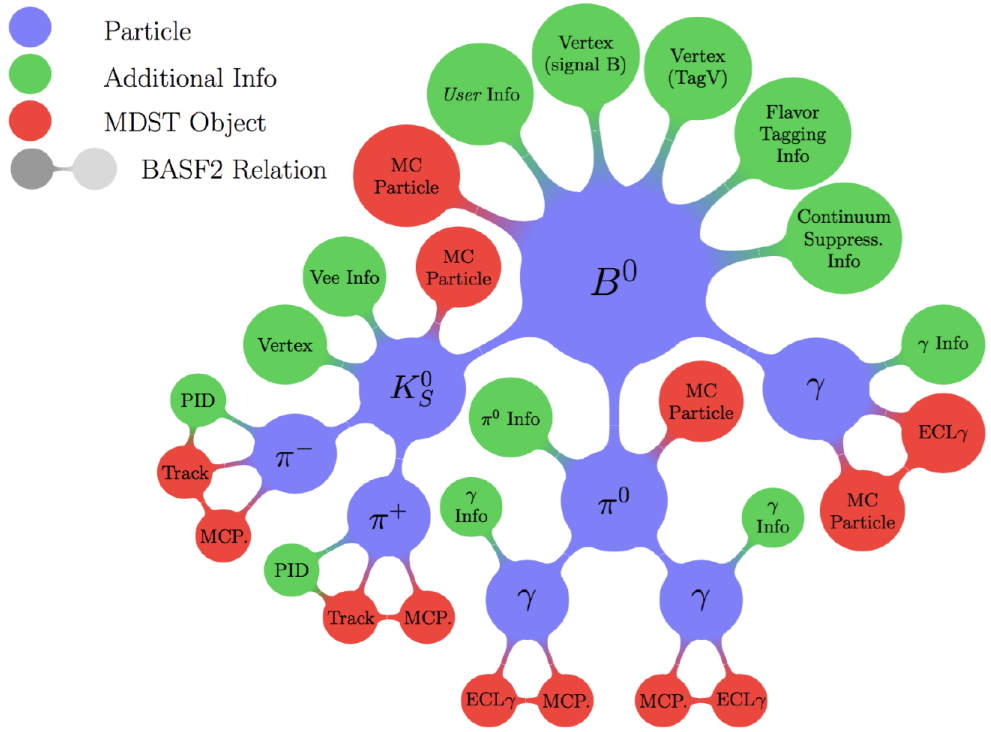
DataStore: know your file types first

Suffix	Full name	What is inside	Who uses it
dst	data summary table	full reconstruction (all detector objects)	production output, you rarely touch it
mdst	mini data summary table	the objects you need **+ the relations between them**	**your analysis input**
udst	user data summary table	skimmed mdst, i.e. an mdst after preselection	Skimmed intermediate files
cdst	calibration data summary table	mdst + digits (closer to raw)	calibration and performance studies

> All of them are ROOT files, but what they contain is **objects + relations**, not "a bunch of branches".

Can I read an mdst with my own ROOT macro?

- **No.** Write basf2 code.
- An mdst stores **relations** between objects (Particle ↔ Track ↔ ECLCluster): hand-rolled ROOT loops get this wrong e
- One particle may have **sev** experiment / momentum / ma
- **An ECLCluster is not a pr** by detector-expert modules
- Relation containers are loa
- Want ROOT plots? Do it at yourself



- At each stage we build relations between the dataobjects
  - Like vertex information, ContinuumSuppression → all related to Particles
  - Particles themselves related to primitive mdst objects (clusters, tracks)

> In short: the mdst is for basf2, not for ROOT macros. ROOT only draws the final plots.

ParticleList: the basf2 analysis model

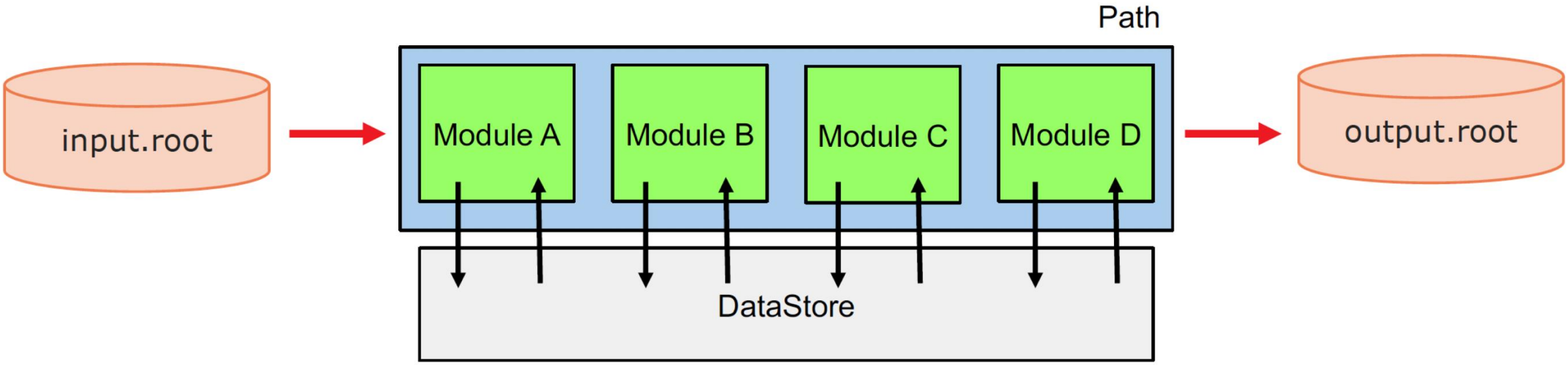
- An analysis is building blocks on ParticleLists: take lists → combine kids into parents → get **candidates**
- Every `reconstructDecay` tries all combinations, so **one event often has several candidates**
- basf2 **keeps all of them**; it never silently picks "the best one"
- For the best-candidate problem the usual recommendation is: don't pick — store all candidates in the ntuple and deal with it statistically (arXiv:1809.02296)
- Bonus: intermediates are reusable and inspectable, so you can tell *which level* gained or lost candidates

> Assume "many candidates per event" from the start — your ntuple is a multi-candidate structure by design.

Path of Data Processing

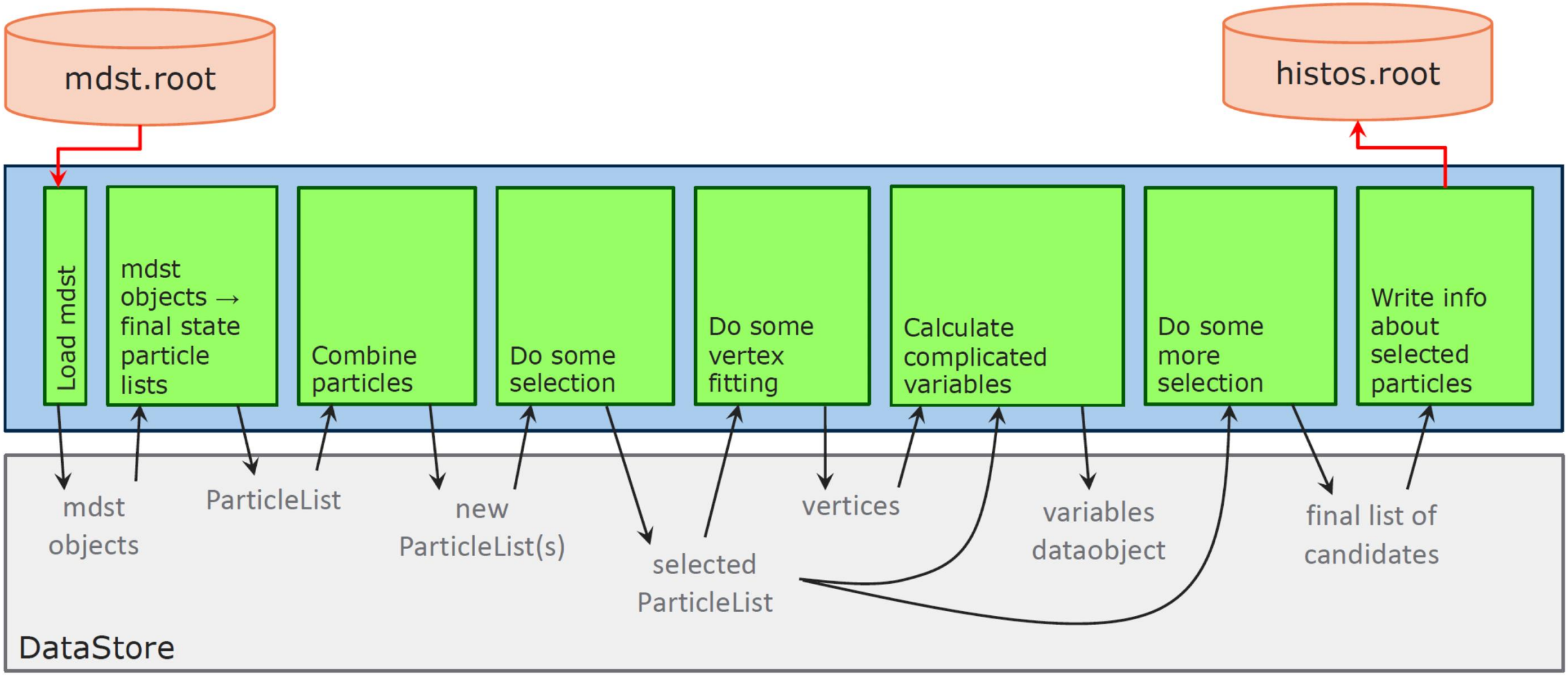
1) A set of classes (modules) that process the data
   → **BASF2 module**

2) A set of classes (dataobjects) that hold the data and allow module to pass thing one to the other
   → **BASF2 dataStore**



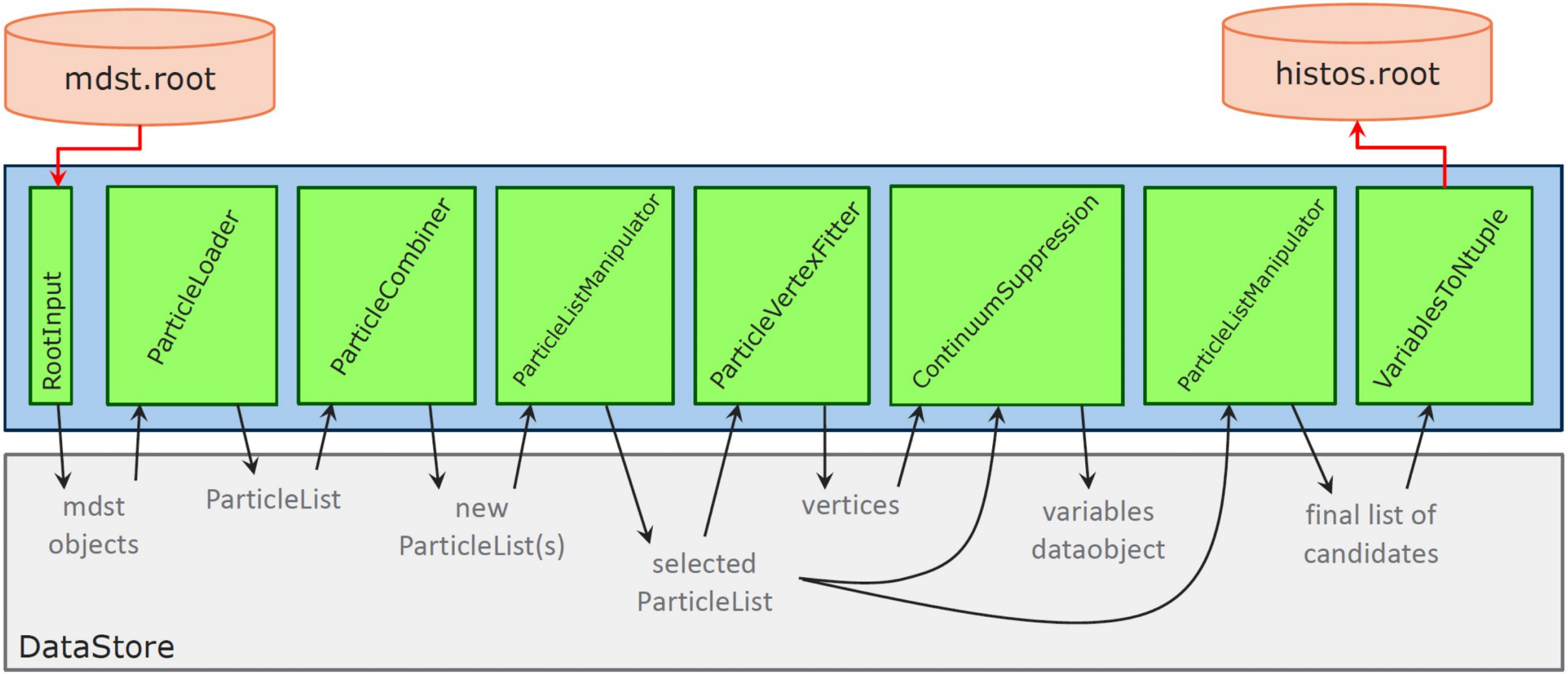
3) An order in which the modules must be executed
   → **BASF2 path**

A typical path for an analysis job



A typical path for an analysis job

Now with the real names for the modules



Skeleton of an analysis script: seven steps

1. ① **Path** b2.create_path()
2. ② **Input** ma.inputMdst(...) / inputMdstList(...)
3. ③ **Final-state lists** fill them yourself with ma.fillParticleList(...), or use stdc.stdPi(...)
4. ④ **Reconstruct bottom-up** ma.reconstructDecay(...), from daughters to the mother
5. ⑤ **Optional processing** ma.matchMCTruth(...) truth matching, vx.treeFit(...) kinematic fit
6. ⑥ **Collect variables** collections (vc.*) + aliases (vu.*, vm.addAlias)
7. ⑦ **Write and run** ma.variablesToNtuple(...) → b2.process(path)

```
my_path = b2.create_path()
ma.inputMdst(filename=..., path=my_path)
ma.fillParticleList('pi+:myloose', cut='...', path=my_path)
ma.reconstructDecay('D0:sig -> K-:loose pi+:loose', cut='1.8 < M < 1.9', dmID=1, path=my_path)
ma.reconstructDecay('D*+:sig -> D0:sig pi+:loose', cut='0.0 < Q < 0.2', path=my_path)
ma.variablesToNtuple(decayString='D*+:sig', variables=..., path=my_path)
b2.process(my_path)
```

Exercise 2 · 2 minutes

Learn to look things up

1. `b2help-particles` — find the name of you interested particle and PDG code

2. `b2help-variables` — what type is `chiProb` (double / int / bool)?

3. Find `create_aliases_for_selected` in `$BELLE2_RELEASE_DIR/analysis/examples/tutorials/B2A301-Dstar2D0Pi-Reconstruction.py`

These two commands answer about 80% of your future "what is this variable / particle called" questions. Worth practising now.

Part 3 · Hands-on A

Run one MC production

- Goal: 100 inclusive ccbar events containing D*±, through simulation and reconstruction
- Understand the four moves: create path / add modules / chain them / process
- Know when you should **not** produce MC yourself

Hands-on A: goal and deliverable

- **Goal**: 100 inclusive ccbar events, each with at least one D*±, through full Geant4 simulation + trigger simulation + reconstruction
- **Deliverable**: `B2A102-ccbar.mdst.root` — an mdst-style file you can feed straight into the Hands-on B script
- **Requirement**: a **full release** (simulation and reconstruction are not in light releases)
- **Runtime**: a few minutes for 100 events on KEKCC; while practising, set `evtNumList` to 10 first
- **Script**: `analysis/examples/simulations/B2A102-ccbarEventGeneration.py`, run `basf2 B2A102-ccbarEventGeneration.py`

> A first run that fails is normal: get 10 events through, then scale up.

MC generation script (1/2)

```
import basf2 as b2
from beamparameters import add_beamparameters
from generators import add_inclusive_continuum_generator
from simulation import add_simulation
from reconstruction import add_reconstruction

# reproducible + log level
b2.set_random_seed(12345)
b2.set_log_level(b2.LogLevel.INFO)

# 1. create your own path
main = b2.create_path()

# 2. event info: expList=0 / runList=1 is the MC convention; 100 events here
main.add_module("EventInfoSetter", expList=0, runList=1, evtNumList=100)

# 3. progress bar: essential for long jobs
main.add_module("Progress")

# 4. beam parameters: Y(4S) energy and spread are set for you
add_beamparameters(main, "Y4S")
```

For independent MC (MCri) use `expList=1003, runList=0`.

MC generation script (2/2)

```
# 5. generator: KKMC makes the ccbar pair -> PYTHIA8 fragments -> EvtGen decays
#    particles=["D*+"]: every event must contain a D*+ (its charge conjugate is implied)
add_inclusive_continuum_generator(main, finalstate="ccbar", particles=["D*+"])

# 6. full Geant4 simulation + L1/HLT trigger simulation
#    to overlay beam background: add_simulation(main, bkgfiles=my_bkg_files)
add_simulation(main)

# 7. reconstruction: digit -> track / ECLCluster / Particle
add_reconstruction(main)

# 8. write an mdst-style ROOT file
main.add_module("RootOutput", outputFileName="B2A102-ccbar.mdst.root")

# 9. run!
b2.process(main)
print(b2.statistics)     # time spent per module, at a glance
```

Four moves, done: **create path → add modules → chain them (order = data flow) → `b2.process(main)`**.

Four moves & three lessons

- **Move ①** `b2.create_path()` — an empty data flow; everything hangs off it
- **Move ②** `main.add_module("Xxx", param=value)` — the parameters are the module's C++ properties
- **Move ③** Add generator → simulation → reconstruction → output **in that order**; reversed order gives "missing object" errors
- **Move ④** `b2.process(main)` actually runs it; `print(b2.statistics)` shows where the time went
- Lesson 1: the beam energy is set by `add_beamparameters(main, [c.m. energy])` — Not necessary if you are working on Y4S.
  - …but other energy points have to be configured by hand
- Lesson 2: overlay beam background with `add_simulation(main, bkgfiles=my_bkg_files)`
  - …or deliberately run without it
- Lesson 3: pick the wrong generator and everything downstream is wasted — agree on the sample with your conveners / supervisor first

> The script should read like a pipeline: the order *is* the physics process.

Choosing a generator

Generator	What it produces	When to use it
EvtGen	a specified decay chain from a `.dec` file	most common; paired with KKMC / continuum generators
KKMC	u/d/s/c continuum with γ radiation	ccbar / uds background samples
PHOKHARA	ISR, $e^+e^- \to$ hadrons + γ	initial-state radiation, luminosity
BabaYaga	Bhabha / μμ / γγ / ττ	low-multiplicity background
BHWide	small-angle Bhabha (< 5°)	small-angle luminosity
Tauola	τ production and τ decays	τ physics
AAFH	two-photon processes	two-photon background
others	—	check `generators/scripts/generators.py`

> You rarely add these modules directly — call the ready-made `add_xxx_generator()` helpers in `generators/scripts/generators.py`.

Three MC FAQs (and one warning)

- **Q: Where is the beam energy set?** A: Automatically to Y(4S), with a realistic spread
- **Q: What if I want a custom beam energy?** A: Call `beamparameters.add_beamparameters()`, then change the module parameters
- **Q: Should I overlay beam background?** A: `add_simulation(main, bkgfiles=bg)`
- **Warning: do not produce the MC sample for your final result yourself.**
- Reason 1: getting the **Global Tag (GT)** completely right is not trivial — the GT carries every parameter your analysis needs
- Reason 2: the signal MC you want has usually been produced already; redoing it is pure waste
- Reason 3: MC samples are huge — storing all of them on KEKCC is not realistic
- **What to do instead**: ask your conveners / DP liaisons whether the sample exists, and ask them to produce new ones

> Self-produced MC is for practice and private checks. Official results use official samples.

A small trick

- **Sometimes, we would like to generate “own designed” MC**

A: Two ways in my head:

1. **Call `` GeneratorPreselection ``. Find more details on [sphynx]**

   Example:

```
def add_cut(name, minParticles, maxParticles, minTheta, maxTheta=None):
    """Add a generator level cut and kill the event if the cut is not passed.  In
    this case the cut is on the min/max charged particles which have a
    center-of-mass theta angle between minTheta and maxTheta. If maxTheta is not
    given assume it to be 180-minTheta for a symmetric window"""

    # if only one angle make it symmetric
    if maxTheta is None:
        maxTheta = 180 - minTheta
    selection = main.add_module('GeneratorPreselection')
    selection.param('applyInCMS', True)
    selection.param('nChargedMin', minParticles)
    selection.param('nChargedMax', maxParticles)
    selection.param('MinChargedTheta', minTheta)
    selection.param('MaxChargedTheta', maxTheta)
    selection.param('MinChargedP', 0.)
    selection.param('MinChargedPt', 0.)
    selection.if_value("!=11", kill)
    selection.set_name("generator cut: " + name)
```

2. **Insert module analysis right after generator then you can `ApplyCut`**

Exercise 3 · 6 minutes

Get the MC running

1. Set `evtNumList` to 10 and run `basf2 B2A102-ccbarEventGeneration.py` on a full release
2. Read `b2.statistics`: which modules dominate the runtime? (hint: Geant4 simulation, always)
3. Change `particles=["D*+"]` to `particles=["D*+", "D+"]`, run again, and explain the change in runtime

If it fails, read the **first line** of the error: an `import` failure is almost always the wrong release; a missing object is usually a wrong module order. `b2 -i script.py` drops you into an interactive prompt for inspection.

Part 4 · Hands-on B

From mdst to ntuple

- Fill ParticleLists yourself → reconstruct D^{*+} bottom-up → fit → variables → ntuple
- Leave with your first analysis ntuple

Hands-on B: goal and input

- **Goal**: from an mdst, reconstruct $D^{*+} \to D^0(\to K^-\pi^+)\pi^+$ and store the candidates in an ntuple
- **Input**: `B2A102-ccbar.mdst.root`, the file produced in the previous step
- **Deliverable**: `anaScript_custom.root`, containing a plain TTree called `dsttree`
- **A light release is enough**: we only read mdst, no simulation
- **Script**: `hands-on/anaScript_custom.py`, run `basf2 anaScript_custom.py`

> In a real analysis you keep this code and only change the `decayString` and the cuts.

Analysis script (1/3): input and particle lists

```
import basf2 as b2
import modularAnalysis as ma
import variables.collections as vc
import variables.utils as vu
import vertex as vx
from variables import variables as vm

my_path = b2.create_path()

# 1. mdst input: every analysis script starts here
ma.inputMdst(filename=b2.find_file('Dst2D0pi.root', 'examples', False),
         path=my_path)

# 2. fill the final-state lists yourself (better than the standard lists)
#    "particle:label" — the label after the colon is yours to choose
ma.fillParticleList('pi+:myloose',
              cut='pt > 0.1 and abs(dz) < 2.0 and abs(drho) < 0.5',
              path=my_path)

ma.fillParticleList('K+:myloose',
              cut='pt > 0.1 and abs(dz) < 2.0 and abs(drho) < 0.5 '
                  'and kaonID > 0.6',
              path=my_path)
```

In a real analysis use `ma.inputMdstList(filelist=my_files, path=my_path)` to read many files.

What particle:label actually does

- fillParticleList('pi+:myloose', cut=...) stores every π^+ passing the cut in a ParticleList called **`pi+:myloose`** (the :myloose is your own label)
- pi+ is particle name + charge (see b2help-particles); use pi- for the antiparticle, or names like anti-D0, D*-
- The cut uses **VariableManager** variables: pt, abs(dz), drho, kaonID, …
- **Why fill it yourself**: the standard list stdc.stdPi(listtype='loose') is a compromise for everyone; your analysis should tune for its own efficiency / signal-to-noise
- Naming tip: pi+:myloose, K+:tight, gamma:good — put the tightness in the name so the script stays readable in a year

> The label is global: reconstructDecay later refers to it by its full name pi+:myloose.

Analysis script (2/3): reconstruct bottom-up

```
# 3. reconstruct D0 -> K- pi+ (M in the cut = invariant mass of the mother)
ma.reconstructDecay(decayString='D0:sig -> K-:myloose pi+:myloose',
                    cut='1.8 < M < 1.9',
                    dmID=1,            # dmID = decay mode ID
                    path=my_path)

# 4. reconstruct D*+ -> D0 pi+
#    Q = M(D0 pi+) - M(D0) - m(pi+): the classic D* observable
ma.reconstructDecay(decayString='D*+:sig -> D0:sig pi+:myloose',
                    cut='0.0 < Q < 0.2',
                    dmID=1,
                    path=my_path)

# 5. MC truth matching: adds isSignal and friends (MC only)
ma.matchMCTruth(list_name='D*+:sig', path=my_path)

# 6. kinematic fit
vx.treeFit(list_name='D*+:sig',
           conf_level=0,          # 0: keep only fitted candidates; -1: keep all
           ipConstraint=True,     # pin the B production vertex to the IP
           updateAllDaughters=False,
           path=my_path)
```

Reconstruct inside-out: $K^-\pi^+$ first makes the D^0, then $D^0\pi^+$ makes the D^{*+}.

Four things about reconstructDecay

- **decayString**: 'mother:label -> daughter1:label daughter2:label' — it references **existing ParticleList full names**, so a typo means "list does not exist"
- **cut**: applies to the new mother, using the mother's variables (M, Q, chiProb, …); daughter variables need daughter(i, var)
- **dmID**: decay mode ID. When a particle has several modes ($D^0 \to K^-\pi^+$ / $K^-\pi^+\pi^0$ / K^+K^-), use different dmIDs; it is stored in extraInfo(decayModeID) and alias-able later
- **Automatic charge conjugation**: writing D0:sig -> K- pi+ also reconstructs anti-D0 -> K+ pi- for you
- Trick: when you do not know the physical range, start with a very loose cut (0 < M < 10), store an ntuple, look at the distribution, then tighten

> Loose first, tight later: cuts exist to reduce candidates and runtime, not to be optimal on day one.

Analysis script (3/3): fit, variables, ntuple

```
# 7. variables + aliases: extraInfo(decayModeID) is long, give it a short name
vm.addAlias('modeID', 'extraInfo(decayModeID)')

dstar_vars = vc.mc_truth + vc.inv_mass + ['Q', 'chiProb', 'modeID']

# ^ marks which particle, prefix prepends to the alias -> d0_M, d0_isSignal ...
d0_vars = vu.create_aliases_for_selected(
   list_of_variables=vc.inv_mass + vc.mc_truth,
   decay_string='D*+ -> ^D0 pi+',
   prefix='d0')

# [] means a chained decay, so several ^ can be marked in one call
fs_vars = vu.create_aliases_for_selected(
   list_of_variables=vc.pid + vc.track + vc.mc_truth,
   decay_string='D*+ -> [D0 -> ^K- ^pi+] ^pi+')

# 8. write the ntuple: the end of the chain, a plain ROOT TTree
ma.variablesToNtuple(decayString='D*+:sig',
                     variables=dstar_vars + d0_vars + fs_vars,
                     filename='anaScript_custom.root',
                     treename='dsttree',
                     path=my_path)

# 9. run!
b2.process(my_path)
```

Variables and aliases: stop hand-writing 100 lines

- Variable collections live in `analysis/scripts/variables/collections.py` and are just **python lists**, so you can add them: `vc.mc_truth + vc.inv_mass + ['Q', 'chiProb']`
- Useful ones: `vc.kinematics`, `vc.inv_mass`, `vc.deltae_mbc`, `vc.pid`, `vc.track`, `vc.mc_truth`, `vc.cluster`
- `vm.addAlias('modeID', 'extraInfo(decayModeID)')` puts `modeID` in the ntuple instead of the long name
- `vu.create_aliases(vc.kinematics, 'daughter(0,{variable})', 'd0')` → `d0_px`, `d0_py`, `d0_pz` …
- `vu.create_aliases_for_selected(list_of_variables, decay_string, prefix=)` uses **`^`** to mark particles and generates a whole batch of prefixed names
- `[]` denotes a chained decay, so several particles can be marked at once: `'D*+ -> [D0 -> ^K- ^pi+] ^pi+'`
- **The order must match your `reconstructDecay`**, otherwise you select a different particle than you think
- Custom collections are fine: they are python lists, just append in a loop

> Aliases are for humans: the branch name must still make sense to you three years from now.

Variable cheat sheet

Variable	Meaning	Typical use
M	invariant mass of the current ParticleList	D^0 window $1.8 < M < 1.9$
Mbc	beam-energy-constrained mass	B signal $5.24 < Mbc < 5.29$
deltaE	energy difference E* − E_beam	B signal $abs(deltaE) < 0.05$
Q	M(D*) − M(D^0) − m(π)	D* signal, peak at 5.85 MeV
chiProb	p-value of the kinematic fit	background rejection, $chiProb > 0.001$
isSignal / mcErrors	MC truth-matching result	**MC only**, use it to check yourself
daughter(0, p)	momentum of the 0th daughter	daughter-level variables
extraInfo(decayModeID)	decay mode ID	tell merged decay modes apart

> No memorising: b2help-variables lists everything, add a keyword to search, and ask an AI/expert when you are stuck.

Exercise 4 · 6 minutes

Extend your analysis

1. Run `basf2 anaScript_custom.py`, open `anaScript_custom.root` in ROOT, and count the branches with `tree.Print()`

2. Loosen the D^0 cut to $1.7 < M < 2.0$; how do the candidate count and the ntuple size change?

3. Add a $D^0 \to K^-\pi^+\pi^0$ mode (`dmID=2`), merge both with `ma.copyLists` into `D0:all`, then reconstruct the D^{*+}

Merging modes: `ma.copyLists(outputListName='D0:all', inputListNames=['D0:ch1','D0:ch2'], path=my_path)`, then write `D0:all` in the decayString. This is exactly what `dmID` is for: after merging you can still tell which mode a candidate came from.

Part 5 · Hands-on C

Turn the ntuple into plots

- Read your own ntuple with ROOT / pyroot
- Plot M($K^-\pi^+$) and Q, fit the peak once
- A troubleshooting table worth bookmarking

Plotting: finally, ROOT

```
import ROOT
ROOT.gROOT.SetBatch(True)      # no interactive window, just write files

f = ROOT.TFile.Open("anaScript_custom.root")
tree = f.Get("dsttree")
print("candidates in ntuple:", tree.GetEntries())
tree.Print()                   # see which branches actually exist

# 1. D0 candidate mass: there should be a peak at 1.865 GeV
c1 = ROOT.TCanvas("c1", "D0 mass", 900, 700)
tree.Draw("d0_M", "isSignal == 1", "hist")
c1.SaveAs("dstar_md0.pdf")

# 2. Q = M(D0 pi+) - M(D0) - m(pi+): peak near 5.85 MeV, flat background
c2 = ROOT.TCanvas("c2", "Delta M", 900, 700)
tree.Draw("Q >> hQ(100, 0.0, 0.02)", "isSignal == 1")
hQ = ROOT.gDirectory.Get("hQ")
hQ.Fit("gaus", "S")
print("sigma = %.3f MeV" % (hQ.GetFunction("gaus").GetParameter(2) * 1000))
c2.SaveAs("dstar_q_fit.pdf")
```

`python3 plot_dstar.py` is all it takes — ROOT ships with the basf2 environment, nothing to install.

Reading the result

- **$M(K^-\pi^+)$**: the D^0 peak sits at **1.865 GeV**; with `isSignal == 1` you look at truth-matched candidates only, the cleanest sample
- **$Q = M(D^0\pi^+) - M(D^0) - m(\pi^+)$**: the peak is at **5.85 MeV** and the background is almost flat, which is why Q beats M in D* analyses (better resolution, smaller systematics)
- Fit it: peak position and width (σ) are the **resolution** of your chain — use them to quantify every later change
- Background sources: random combinations, real D^0 paired with a wrong π^+, continuum — each has its own suppression tool (PID, ΔE / Mbc, event shape)
- **Histograms directly?** You can: write your own module that fills histograms inside `b2.process()`. Most people do not — ntuple then ROOT is the default

> The ntuple is your data product: store everything, plot it, and only then optimise.

Troubleshooting table (bookmark this)

Symptom	Cause	Fix
`No module named 'generators'`	running simulation on a light release	switch to a full release: `b2setup release-09-00-15`
`ParticleList ... does not exist`	the list referenced in decayString was never built	check the order: fill / reconstruct daughters first
`Variable ... not found` in a cut	wrong name, or wrong level	look it up with `b2help-variables <name>`
a branch is missing from the ntuple	not added to the variable list, alias not registered	use `tree.Print()` to see the real branch names
running very slowly	input is not mdst, or candidate explosion	make sure it is mdst; tighten cuts before adding variables
`Q` distribution is empty	wrong units (GeV, not MeV)	write `0.0 < Q < 0.2`, not `0 < Q < 200`

> Read the **first line** of the error: `import` failures are environment problems; missing objects or lists are script-order problems.

Part 6

Going further

- B2BII: putting Belle data on the same chain
- Generating Belle MC inside basf2
- Scaling up: gbasf2 and b2luigi

B2BII: Belle → Belle II

- Reads Belle mdst files
- Applies corrections to the Belle objects ("fix mdst"), with the HadronB(J) skim on by default
- Converts the Belle mdst into a Belle II mdst
- After that: **every basf2 analysis module works as usual**
- B2BII analyses keep their own particle-list conventions, but the usage is identical

```
from b2biiConversion import convertBelleMdstToBelleIIMdst

convertBelleMdstToBelleIIMdst(inputBelleMDSTFile, path=my_path)

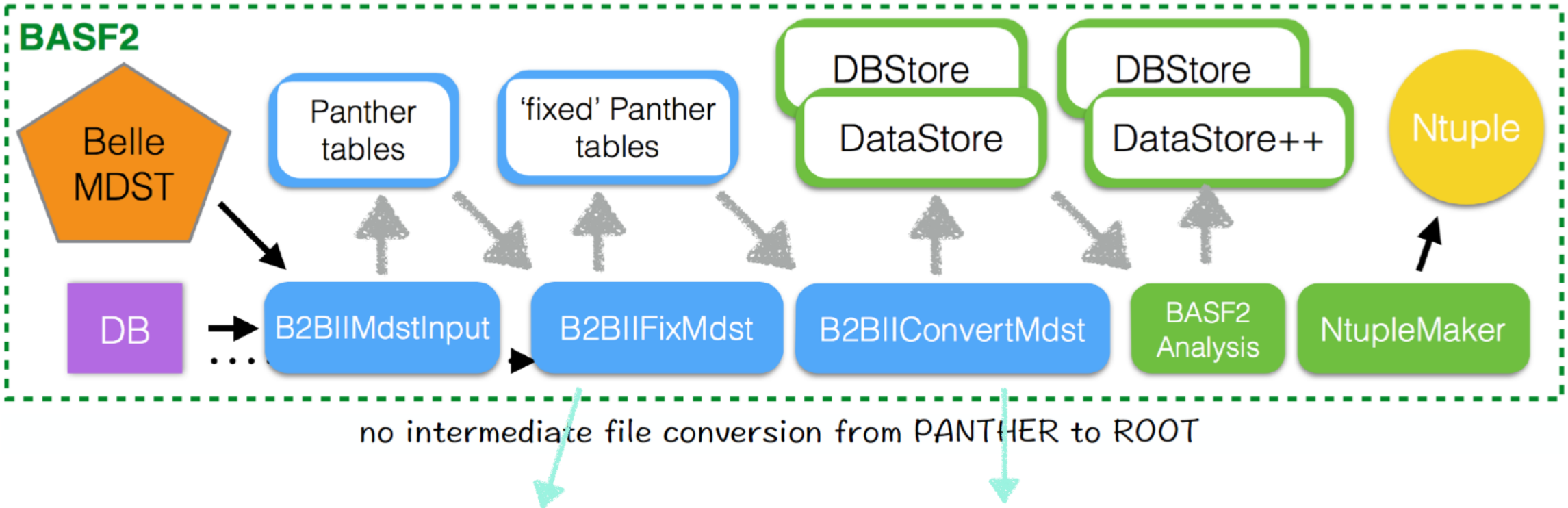
# from here on it is an ordinary Belle II mdst: ParticleList / reconstructDecay
# / ntuples all work unchanged, every basf2 analysis module applies
```

> One conversion line brings Belle data — and Belle's analysis experience — into the same basf2 workflow.

B2BII

Belle to Belle II

✦ Read and analyze Belle MDST format in basf2.



basf	basf2	In basf2 analysis script
Mdst_charged	Track	*Create ParticleLists (w/ or w/o cuts) of Final State Particles:* *fillParticleList('K+:mine','atcPIDBelle(3,2)>0.6 and abs(d0)<2 and abs(z0)<4', path=mypath)*
Mdst_gamma	gamma:mdst ECLCluster	*Use already created 'gamma:mdst' ParticleList:* *cutAndCopyList('gamma:mine,'gamma:mdst','E > 0.1', path=mypath)*
Mdst_pi0	pi0:mdst (daughters link to gamma:mdst)	*Use already created 'pi0:mdst' ParticleList:* *reconstructDecay('rho+:myRho -> pi+:all pi0:mdst', '0.6 < M < 1.0', path=mypath)*
Mdst_vee2	K_S0:mdst Lambda0:mdst gamma:v0mdst	*Use already created 'K_S0:mdst', 'Lambda0:mdst', and 'gamma:v0mdst' ParticleLists:* *cutAndCopyList('K_S0:good', 'K_S0:mdst', cut='extraInfo(goodKs)==1', path=mypath)*
Mdst_klong	K_L0:mdst	*Use already created 'K_L0:mdst' ParticleList:* *reconstructDecay('B0:jpsikl-> J/psi:my K_L0:mdst', 'Mbc>5.0', path=mypath)*

Generating Belle MC inside basf2

- **Step 1**: generate events with a generator (PHOKHARA, EvtGen, PYTHIA…)
- **Step 2**: run gsim (simulation + reconstruction)
- Replace step 1 with basf2 generators and write Belle format out: that is Belle MC — it only needs the `BelleMCOutput` module

```
main.add_module(phokhara)
main.add_module(evtgendecay)
main.add_module("BelleMCOutput", outputFileName=output_mdst)
main.add_module("PrintMCParticles", logLevel=basf2.LogLevel.DEBUG,
                onlyPrimaries=False)
main.add_module("Progress")
basf2.process(main)
```

> Compare with the MC script from Part 3: identical structure, only the output module changes to `BelleMCOutput`.

Part 7

AI agents in the loop

- Where an agent genuinely saves you time
- A workflow that keeps you in control
- What you must never delegate

Where agents actually help

Task	Fit	Why
Explain an unfamiliar basf2 API	★★★	the docs are large; an agent summarises `pkg/scripts/*.py` in seconds
Draft boilerplate (path, module params, ntuple wiring)	★★★	the pattern is stable across analyses
Decode a traceback	★★★	most basf2 errors are one of about ten shapes
Translate a colleague's script or comments	★★★	cheap and low risk
Refactor a 200-line script into functions	★★	good, but read the diff carefully
Suggest physics cuts	★	needs your efficiency and background budget
Set the Global Tag / production config	✗	a wrong GT silently ruins the sample
Write your systematic uncertainties	✗	needs your own understanding of detector effects

> Use the agent where the answer is **verifiable in one minute**; keep the physics to yourself.

A workflow that works

1. Give **context**, not a bare question: release, package, goal, input file, and the exact error text
2. Ask for a **minimal** script first (path + one module), then grow it step by step
3. Run it on **10 events** before believing anything
4. Paste the **first lines** of the traceback back — not the whole log
5. Verify every variable and module name with `b2help-variables` / `tree.Print()` / the source tree
6. Read the diff before you accept it: **you own the script, not the agent**

> Treat the agent as a fast junior colleague: excellent at drafts, unreliable on physics.

Prompt recipes

```
# 1) Error triage
I am running basf2 release-09-00-15 (light release) on this script:
<paste script>
It fails with: <paste the first 20 lines of the traceback>.
What is the most likely cause? Give me one fix, not five.

# 2) API question
In basf2 release-09-00-15, what arguments does
variables.utils.create_aliases_for_selected take?
Show a minimal example for the decay 'D*+ -> ^D0 pi+'.

# 3) Debug a cut
This cut returns an empty ParticleList:
ma.reconstructDecay('D*+:sig -> D0:sig pi+:loose', cut='0.0 < Q < 0.2')
Is the unit or the variable wrong? What should I print to find out?

# 4) Refactor
Split this 200-line script into a config dict, a build_path() and a main().
Keep every basf2 call byte-for-byte identical.
```

Each prompt names the release, the package and the observable — that is what makes the answer checkable.

Guardrails: what not to delegate

- **Physics is yours**: selection, cuts, efficiencies, background estimates, systematic uncertainties
- **Never trust an invented API**: suggested module names and parameters are plausible and often wrong — check against `$BELLE2_RELEASE_DIR/Sphinx`
- **Global Tags and production settings** always come from the official sample or your DP liaison
- **Data policy**: no internal or unpublished data into public models; use a local model for anything sensitive
- **Reproducibility**: if an agent wrote it, you still pin the release and the random seed — a script you cannot re-run is not a result
- **Disclose**: if a tool materially shaped your analysis, say so where your collaboration requires it

> The agent is a tool in your path — it is not part of the data flow, and it is not an author.

Tools and how to wire them up

Tool	Best for	Watch out for
VS Code + Copilot / Cursor	inline edits while you write a script	keep completions short, review each one
Claude Code / Codex CLI	repo-level work: read the package source, run tests	give it read access to `$BELLE2_RELEASE_DIR`
Local models (LM Studio etc.)	anything you cannot send out	slower and weaker, but stays on your machine
MCP / tool servers	letting the agent read docs and files itself	more autonomy means more careful review
`b2help-*` + questions.belle2.org	authoritative answers	**always the final word**

> Rule of thumb: the more autonomous the tool, the smaller the change you let it make at once.

After the course

Stage	What to do	Where
Today	get MC generation + analysis + ntuple running	`hands-on/` in this course directory
Next	prepare your own script for your own analysis	`$BELLE2_RELEASE_DIR/analysis/examples/tutorials/`
Variables	`b2help-variables` + the sphinx variables chapter	https://software.belle2.org
Scale	gbasf2 tutorial, b2luigi starter kit	software.belle2.org / b2luigi.belle2.org
Stuck	search questions first, then post; QQ / WeChat groups	https://questions.belle2.org
Signal MC	ask conveners / DP liaisons	—

> The path is clear: copy a tutorial → adapt one channel → go to production. The real difficulty is never the syntax, it is the physics selection.

Thank you!

- Three commands: `basf2 B2A102-ccbarEventGeneration.py` → `basf2 anaScript_custom.py` → `python3 plot_dstar.py`
- Three links: https://software.belle2.org ／ https://questions.belle2.org ／ https://gitlab.desy.de/belle2/software/basf2
- One habit: search before asking, parameterise your I/O, always run 10 events locally first
- And: ask the AI first, then ask a human — then verify
- Questions?