Intro to basf2

Logan Benninghoff (UMiss)

Alex Gale (UC)

Belle II Summer Workshop 2025

at Virginia Tech

Overview

- Intro to basf2
- Physics Analysis (Broadly)
- Converting this to code
- Specific code we use
- Wrap up

Introduction to basf2

basf2 ~ “Belle II Analysis Software Framework”

basf2 is the software that handles most data processing tasks at Belle II. Most notably used for,

- Generating simulated data
- Processing raw data from detectors
- Reconstructing events

(Final analysis of data after these steps is often handled outside of basf2.)

What is basf2?

basf2 is a collection of C++ modules that we can call using a python interface (steering scripts).

It is also a set of command line tools that we can use to handle data files.

The basf2 documentation is an excellent resource,

https://software.belle2.org/

Reference the sphinx documentation for the release you are using (often the latest light release).

basf2 versions

“release-08-01-07” ~ Format for basf2 releases

major release ~ New version once a year, contains all new software changes.

minor release ~ Adds limited set of new features.

patch ~ Primarily bug fixes.

In addition to the full releases, there are light releases of the software, that incorporate data analysis changes.

basf2 versions

For the hands on activity we can work with a light release. The format for a light release is,

“light-2505-deimos”

25 ~ Release year

05 ~ Release month

deimos ~ “codename” for this light release

Light releases contain a focused subset of basf2 packages, suitable for physics analysis tasks.

basf2 Key Terminology

basf2 Module ~ A collection of code (usually C++ code) that performs a specific data processing task.

Path ~ Modules are added to a sequence that we call a path, then are run in order.

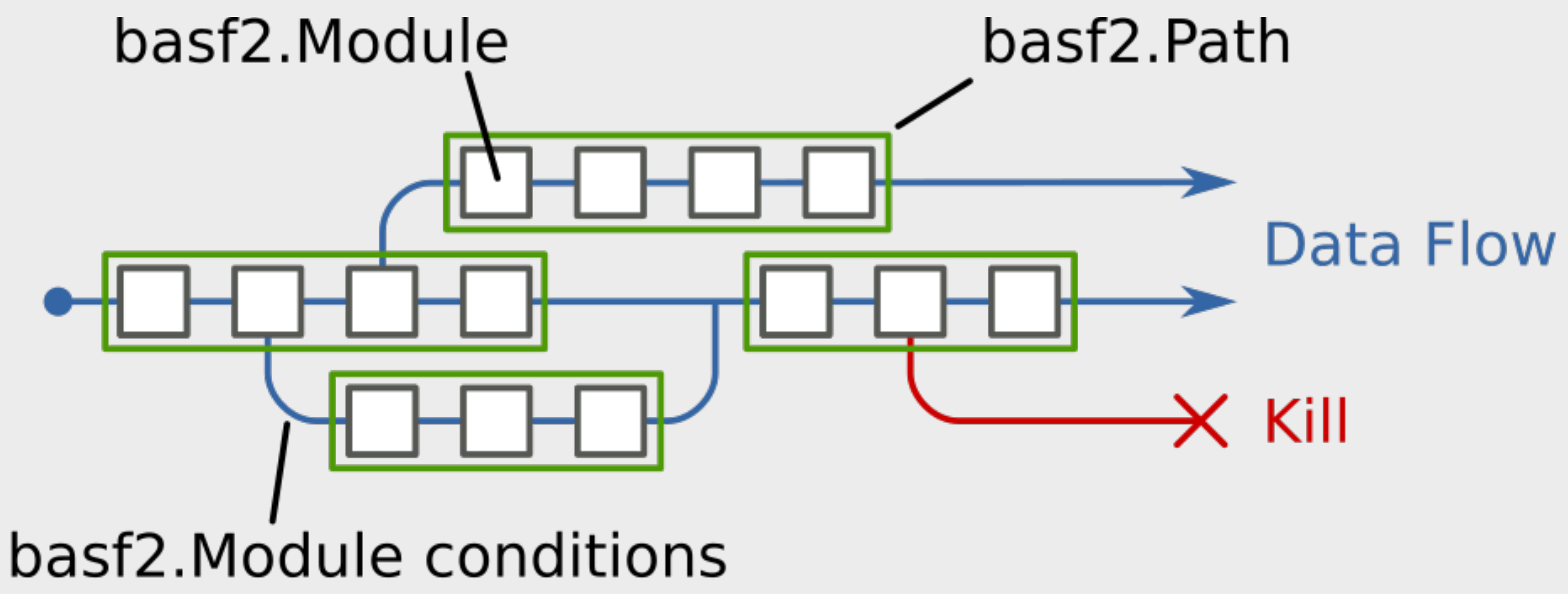

Image courtesy of basf2 sphinx documentation.

basf2 Key Terminology

Steering file/script ~ A python script used to create a path, add the desired modules to it, then run the sequence.

Packages ~ Collections of related modules and python scripts.

Package examples: analysis, tracking, reconstruction, ecl, klm…

You can see the list on gitlab, each directory on the page is a package ~ https://gitlab.desy.de/belle2/software/basf2

The Data Store

[Note ~ As a beginner you will not need to use this subject directly. Just good to know the terminology.]

How do modules save/load information as the path is running?

Modules retrieve and save data objects to the globally accessible data store.

Data objects ~ These are C++ objects that contain collections of variables relevant to that data object.

The Data Store

[Note ~ As a beginner you will not need to use this subject directly. Just good to know the terminology.]

Ex. ~ The KLMMuidHits data object contains the position of a hit in KLM layer 5 (and all other hit info).

We connect data objects with relations, so we build more complex objects out of lower level objects.

Ex. ~ Cluster objects constructed with info from Hits.

For efficiency/storage reasons, low-level objects are deleted after reconstruction-level objects are formed.

mDST

This will be discussed more in data production talks later this week, so just a brief intro,

mDST (mini Data Summary Table) ~ These are .root files containing only data objects necessary for a physics analysis. (~Tracks, PID Likelihoods, Clusters)

For the hands-on session we will work with a prepared and provided mDST file.

(Note ~ ROOT is a software framework started at CERN, we use .root files to store data.)

Physics Analysis (Broadly)

Load a subset of events from an mdst, that contains the necessary data objects (tracks, clusters, PID likelihoods, etc.)

We reconstruct final state particles from data objects.

Note ~ The construction of object -> particle is not one-to-one.

Ex. ~ One track object can have equal likelihoods of being a muon or pion hypothesis, and we may construct both a muon and a pion out of this track.

Physics Analysis (Broadly)

Final state particles and some of their object sources (in conjunction with PID objects),

data object	particle
Track	e, μ, π, K, p, d
neutral ECL Cluster	γ, K_L^0, n
Neutral KLM Cluster	K_L^0, n
V0	$\gamma, K_S^0, \Lambda, \bar{\Lambda}$

From final state -> reconstruct more energetic particles that decayed in the event.

For all particle reconstructions, we include cuts (some are more standard, some analysis specific).

Physics Analysis (Broadly)

Note ~ Our physics analyses are run on simulated data while being developed. Gives insight into the process studied, and means we add a step here,

We need to match the event reconstructed with the known parameters of the simulated event.

Ex. ~ Check how many pions we mistakenly reconstructed as muons.

Built in tools exist, but there will also be a hands-on MC truth matching session with better methods.

Physics Analysis (Broadly)

Finally, we need to write out our variables of interest from the event into an Ntuple (.root file).

Now we convert this abstract plan into basf2 code.

Physics Analysis (Broadly)

Finally, we need to write out our variables of interest from the event into an Ntuple (.root file).

Now we convert this abstract plan into basf2 code.

What basf2 code do I need to write? (Show of fingers, 1-4. Pick the best option, they're not all strictly "wrong".)

1) A module

2) A package

3) A steering file

4) A data object

Physics Analysis (Broadly)

Finally, we need to write out our variables of interest from the event into an Ntuple (.root file).

Now we convert this abstract plan into basf2 code.

What basf2 code do I need to write? (Show of fingers, 1-4. Pick the best option, they're not all strictly "wrong".)

1) A module

2) A package

3) A steering file

4) A data object

Steering File Anatomy (Incomplete)

import basf2 python module (different usage of "module" terminology)

Create the path

Read in data file(s)

Make final state particle list

Form composite particles, truth match

Save variables to an output file

Process the path

Steering File Anatomy (Analysis)

import basf2 python module (different usage of “module” terminology) + other python modules

Create the path (and set other initial condition info)

Read in data file(s)

Make final state particle list (with cuts)

Form composite particles, truth match (with cuts)

Save variables to an output file (with alias names)

Process the path (print statistics)

Steering File Anatomy (Analysis)

import basf2 python module + others

Create the path (and set initial condition info)

Read in data files

Make final state particle list, with cuts

Form composite particles, with cuts

Truth match

Save variables to an output file (with alias names)

Process the path, print statistics

Imports

Need to import any python modules that we need. Starting out we want,

- `import basf2 as b2`
- `import modularAnalysis as ma`

modularAnalysis ~ Python module containing wrapper functions for C++ modules. We can call most of the modules we need for physics analyses with this.

Remember to add the import for python modules you add later on.

Steering File Anatomy (Analysis)

✓ import basf2 python module + others

Create the path (and set initial condition info)

Read in data files

Make final state particle list, with cuts

Form composite particles, with cuts

Truth match

Save variables to an output file (with alias names)

Process the path, print statistics

Creating Path

We then want to setup our path and give it a name ~ main_path here,

- `main_path = b2.Path()`

Note ~ The Belle II detector calibration values and configuration of components change over time.

Some modules need this information, so it is stored as payloads in the conditions database.

If we need this information, we insert an additional line of code before creating our path.

Globaltag

Payloads have an interval of validity (iov), which is the range of runs they are valid for.

A globaltag collects the payloads and iov's for a dataset, and we add it in our steering file with,

```
b2.conditions.append_globaltag(
ma.getAnalysisGlobaltag()
)
```

This line would precede creating our path.

For the hands-on activity we won't need to make use of the globaltag, but it is important to discuss.

Globaltag

Bonus note,

At the moment (June 2025), in order to make use of Neural Network variables (Ex. ~ PionIDNN), we need,

- `b2.conditions.append_globaltag(`
- `‘pid_nn_release08_v1’`
- `)`

The ordering of globaltags matters for some applications ~ This is why we have both “append” and “prepend” globaltag options.

Steering File Anatomy (Analysis)

✓ import basf2 python module + others

✓ Create the path (and set initial condition info)

Read in data files

Make final state particle list, with cuts

Form composite particles, with cuts

Truth match

Save variables to an output file (with alias names)

Process the path, print statistics

Input files

To read in a data file, we use inputMdst(), or inputMdstList() for multiple files,

- `ma.inputMdst(“~/fakename.root", path=main_path)`

Note ~ For most modules, you will need to fill in your path name.

For the hands-on session we’ll replace the file name string.

Steering File Anatomy (Analysis)

✓ import basf2 python module + others

✓ Create the path (and set initial condition info)

✓ Read in data files

Make final state particle list, with cuts

Form composite particles, with cuts

Truth match

Save variables to an output file (with alias names)

Process the path, print statistics

Forming Particles

To build final state particles we call the fillParticleList() module, and provide a string for the particle type,

- ▸ `ma.fillParticleList(‘e-:all’,‘’, path=main_path)`

Note ~ For charged particles, fillParticleList by default will create a list for both the given particle, and it’s charge conjugate.

Creates two lists, one for e-, one for e+.

Particle Grammar I

We provided fillParticleList with a decayString that gives the final state particle, with a label after a colon.

There are two reserved labels,

“e-:all” ~ Label for no cuts

“e-:MC” ~ Label for generated particles (MCparticle)

The particle name before the colon is converted to the PDG code for that particle when the module runs.

Adding cuts

We use the decayString labels to indicate what cuts we've applied to a list of particles.

The cuts chosen depend on your analysis and the particle type, but there are some common cuts.

Convenient way to do this, define a string,

```
goodTrack = 'thetaInCDCAcceptance and
dr < 0.5 and abs(dz) < 2’
```

```
ma.fillParticleList(‘e-:all’,
goodTrack, path=main_path)
```

What cuts to use?

The cuts I've chosen here are,

~ The track must be within the polar angle $17^\circ < \theta < 150^\circ$ (within the range of the CDC).

~ The track originates at a position within a radial distance 0.5 cm and z-component distance of 2 cm from the interaction point.

(These are relatively common.)

```
▶ goodTrack = 'thetaInCDCAcceptance and
▶ dr < 0.5 and abs(dz) < 2’
```

Adding cuts

We can also apply cuts after creating the particle list,

```
▶ goodTrack = 'thetaInCDCAcceptance and
▶ dr < 0.5 and abs(dz) < 2’
```

```
▶ ma.fillParticleList(‘mu-:select’,
▶ goodTrack, path=main_path)
```

```
▶ ma.applyCuts(“mu-:select",
▶ cut=‘p > 2.0’, path=main_path)
```

(ma.applyCuts will be useful later.)

What cuts to use?

There are standard particle lists for charged particles, but we generally do not use them past early stages.

It is recommended that you use the standard selections for V0 decays (neutral decays to two charged particles) and other neutral decays.

Usage for K_S^0 example, creates “K_S0:merged” list,

- `import stdV0s as s0`
- `s0.stdKshorts(path=main_path)`

Steering File Anatomy (Analysis)

✓ import basf2 python module + others

✓ Create the path (and set initial condition info)

✓ Read in data files

✓Make final state particle list, with cuts

Form composite particles, with cuts

Truth match

Save variables to an output file (with alias names)

Process the path, print statistics

“Combining” final state particles

From the final state particles, we reconstruct the particle they decayed from. We use reconstructDecay for this,

```
▶ ma.reconstructDecay(decayString,
▶ cut=‘X.XX < M < Y.YY’, path=main_path)
```

reconstructDecay calls the particleCombiner module.

By default, particleCombiner creates the charge conjugated list along with described decay. It also ensures no particles are reused in a decay.

What do we put for decayString?

Particle Grammar II

We write our decay string with the following structure,

General: “Mother” arrow “Daughter 0” “Daughter 1”…

For multiple decays we use brackets,

Ex. ~ Lambda_c+ -> Sigma+ [K_S -> pi+ pi-]

For reconstructDecay, we include associated labels,

Lambda_c+:select -> Sigma+:loose [K_S:loose -> pi+:loose pi-:loose]

- `ma.reconstructDecay('Xi_c+:sig ->`
- `Lambda_c+:pkpi pi0:eff40_May2020',`
- `cut=‘X < M < Y’, path=main_path)`

Vertex Fitting

After we've reconstructed the mother particle, we may want to re-fit the decay vertex. - Gives new vertex position and 4-momenta

The vertex python module contains fitting algorithms,

~ TreeFit: Global fitting algorithm, good for long decay chains

~ kFit: Fast, kinematics based fitter

~ More options available, not covered here (hands-on will use TreeFit)

Vertex Fitting

Using TreeFit,

-
  ```
  import vertex as vx
  ```

-
  ```
  vx.treeFit('Xi_c+:sig', conf_level=0,
  updateAllDaughters=True,
  massConstraint=['pi0'], path=main)
  ```

~ particleList: We give the mother particle in the decay chain that we're fitting.

~ conf_level: Minimum accepted confidence level. The value 0 here means we accept any successful fits. A value of -1 accepts all candidates.

Vertex Fitting

Using TreeFit,

- `import vertex as vx`

- `vx.treeFit('Xi_c+:sig', conf_level=0,`
- `updateAllDaughters=True,`
- `massConstraint=['pi0'], path=main)`

~ updateAllDaughters: Rewrites momenta/position of daughters after the fit.

~ massConstraint: We tell the fitting algorithm to use the known PDG mass of the particle given to constrain the fit.

Vertex Fitting

If we use updateAllDaughters=True, we may want to save the 4-momenta or vertex position before the fit,

- `ma.variablesToExtraInfo(‘pi0:eff40_May2020’,`
- `variables={‘M':'Mpi0_before_fit'},`
- `path=main)`
- `va.addAlias(‘pi0_MBeforeFit',`
- `'extraInfo(Mpi0_before_fit)')`

We’ll see some of these commands farther in.

Best Candidate Selection

There may be multiple sets of fit parameters that satisfy our decay reconstruction, or vertex fit, how do we pick one to use?

We chose a variable to rank by (chiProb here),

```
ma.rankByHighest(‘Xi_c+:sig',
variable=‘chiProb', allowMultiRank=True,
outputVariable=‘Xi_c+_rank’, path=main)
```

```
va.addAlias(‘Xi_c+_rank’,
‘extraInfo(Xi_c+_rank)’)
```

```
ma.applyCuts(‘Xi_c+:sig’,
‘Xi_c+_rank == 1’, path=main)
```

Steering File Anatomy (Analysis)

✓ import basf2 python module + others

✓ Create the path (and set initial condition info)

✓ Read in data files

✓Make final state particle list, with cuts

✓Form composite particles, with cuts

Truth match

Save variables to an output file (with alias names)

Process the path, print statistics

Truth Matching

We'll have a hands-on discussion for this later in the week.

For now, just use matchMCTruth, which will relate Particles and MCParticles,

```
ma.matchMCTruth(list_name='Xi_c+:sig',
path=main)
```

Notably, produces a binary isSignal variable, which is 1 for correctly reconstructed, 0 for incorrectly, and NaN for no related particle found.

Steering File Anatomy (Analysis)

✓ import basf2 python module + others

✓ Create the path (and set initial condition info)

✓ Read in data files

✓Make final state particle list, with cuts

✓Form composite particles, with cuts

✓Truth match

Save variables to an output file (with alias names)

Process the path, print statistics

Variables

basf2 has pre-defined sets of variables called variable collections.

Some examples so far have included variables from kinematics (InvM, M, p, …) and mc_truth (isSignal…)

The basf2 documentation is your best friend when it comes to finding useful variables.

https://software.belle2.org/development/sphinx/analysis/doc/Variables.html#variable-InvM

Aliases

addAlias should be used to simplify names of some information saved.

Ex. ~ Variables with arguments in parentheses.

```
va.addAlias(‘gamma0_e9e21',
'daughter(0,clusterE9E21)')
```

We can use create_aliases_for_selected from the variables.utils python module to make variables for multiple particles in a decay string.

Particle Grammar III

We use indicators in the decay string to pull variables for specific particles,

- `vu.create_aliases_for_selected(`
- `list_of_variables=['M'],`
- `decay_string='B0 -> ^J/psi ^K_S0',`
- `prefix=['daughters'])`

The marker chosen, "^" here, indicates which particles to create the listed aliased variables for.

There are other marker options that you could look up in the documentation.

Saving Variables

We need to write out the variables we are interested in to an output file.

Generally we create a list of our variables, optionally using our aliases.

We run variablesToNtuple to output a .root file with a tree name given,

```
▶ ma.variablesToNtuple(‘Xi_c+:sig’,
▶ variables=list_of_vars,
▶ filename=‘my_ntuple.root’,
▶ treename='particle', path=main)
```

Steering File Anatomy (Analysis)

✓ import basf2 python module + others

✓ Create the path (and set initial condition info)

✓ Read in data files

✓Make final state particle list, with cuts

✓Form composite particles, with cuts

✓Truth match

✓Save variables to an output file (with alias names)

Process the path, print statistics

Executing the path

Add processing the path,

- `b2.process(main_path)`

Print statistics line,

- `print(b2.statistics)`

Steering File Anatomy (Analysis)

- ✓ import basf2 python module + others
- ✓ Create the path (and set initial condition info)
- ✓ Read in data files
- ✓Make final state particle list, with cuts
- ✓Form composite particles, with cuts
- ✓Truth match
- ✓Save variables to an output file (with alias names)
- ✓Process the path, print statistics

Wrapping up

This is the extent of this overview, but only the beginning of the features available.

~ Continuum Suppression

~ Event level cuts

~ Full Event Interpretation (FEI)

~ Rest of Event (ROE), and ROE masking

~ Flavor Tagging

~ MVA package

~ And more…

End