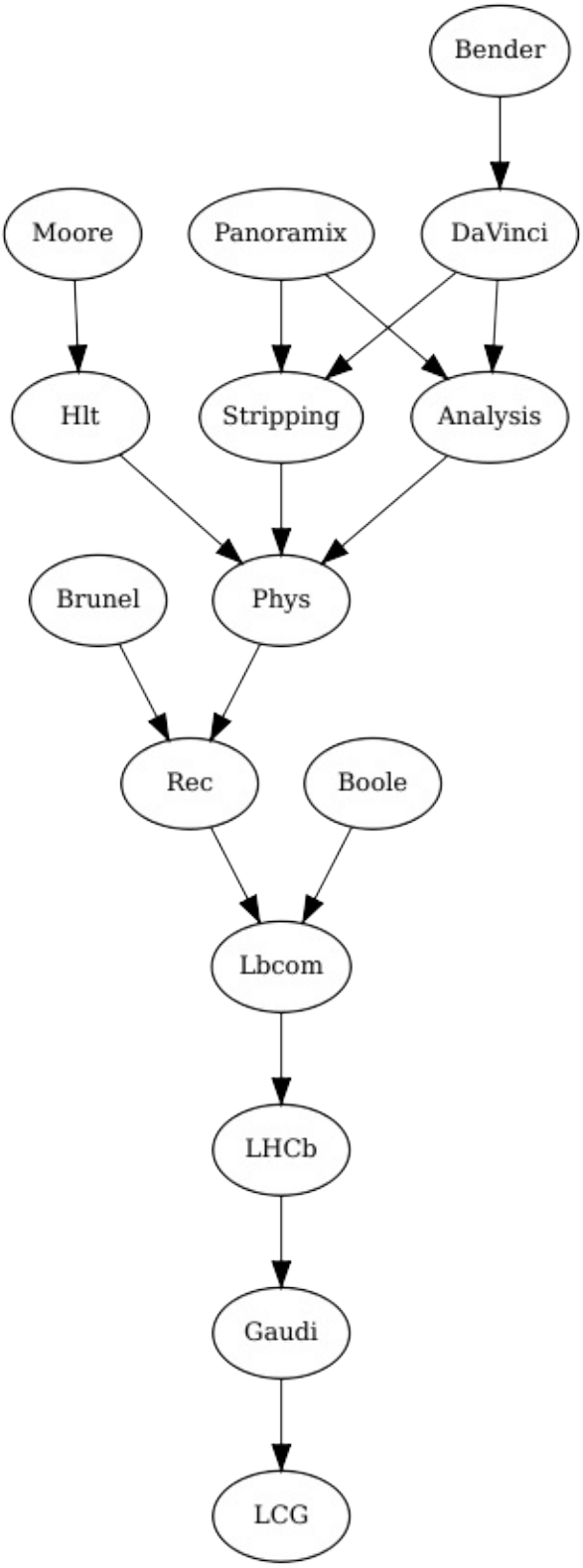
Table of Contents

The Bender tutorials 

These are tutorials for Bender application: "User-friendly python analysis environment for LHCb".

It is the first attempt to convert existing TWiki-based tutorials on Bender and BenderScript to GitHub platform, inspired by the great success of LHCb StarterKit lessons.

Bender is LHCb Python-based Physics Analysis Environment. It combines the physics content of DaVinci-project with the interactive python abilities provided by GaudiPython. It also could be considered as "Interactive LoKi". The major functionality comes from ROOT/Reflex dictionaries for the basic C++ classes and the interfaces. These dictionaries are used primary for POOL persistency and effectively reused for interactivity. The main purpose of top-level scripts is the coherent orchestration of the Reflex dictionaries and the proper decoration of the available interfaces.



Bender dependencies are sketched here:

Doxigen documentation for Bender is accessible here.

It is assumed that users are already has *some* knowledge of LHCb software, in particular DaVinci and are familiar with [LHCb Starterkit] [starterkit].

Getting started

Learning objectives

- Understand the overal structure of Bender module and the corresponding configuration
- Access to data, looping, histograming and n-tupling
- Combining particles with creation of compound particles

We'll start Bender tutorials with few very simple examples

- (trivial) *DoNothing* and *DaVinci* examples
- (less trivial) *HelloWorld* examples
- access to data, looping, histograming and n-tupling
- combining particles with creation of compound particles

The first two *almost useless*, but *very important* examples

Learning objectives

- Understand the overall structure of Bender *module* and the configuration of the application

Do-nothing

Learning objectives

- Understand the overall structure of Bender *module* using the oversimplified example

Any *valid* Bender module must have two essential parts

- function `run` with the predefined signature
- function `configure` with the predefined dignature

For the most trivial (*"do-nothing"*) scenario function `run` is

```
def run ( nEvents ) :
    # some fictive event loop
    for i in range( 0 , min( nEvents , 10 ) ) :  print ' I run event %i ' % i
    return 0
```

In a similar way, the simplest *"do-nothing"*-version of `configure` -function is

```
def configure (  datafiles ,  catalogs = [] , castor = False , params = {} ) :
    print 'I am configuration step!'
    return 0
```

As one clearly sees, these lines do nothing useful, but they are perfectly enough to be classified as the first Bender code. Moreover, the python module with these two function can already be submitted to Ganga/Grid, and Ganga will classify it as valid Bender code. Therefore this code is already *"ready-for-Ganga/Grid"*!

The details for the curious students: how Ganga/Grid treat Bender modules?

Actually Ganga executes at the remote node the following wrapper code

```
files    = ... ## this one comes from DIRAC
catalogs = ... ## ditto
params   = ... ## extra parameters (if needed): this comes from the user
nevents  = ... ## it comes from Ganga configuration

import USERMODULE   ## here it imports your module!
USERMODULE.configure ( files , catalogs ,  params = params )
USERMODULE.run       (  nevents )
```

Thats all! From this snippet you see:

- the code must have the structure of python *module*, namely no executable lines should appear in the main body of the file
  - (note the difference with respect to the *script*)
- it must have two functions `run` and `configure`
  - (everythnig else is not used)

The whole module is here:

```
## 1) some user code :
def run ( nEvents ) :
    for i in range( 0 , min( nEvents , 10 ) ) : print ' I run event %i ' % i
    return 0

## 2) configuration step
def configure (  datafiles ,  catalogs = [] , castor = False , params = {} ) :
    print 'I am configuration step!'
    return 0
```

DoNothing.py hosted with ❤ by **GitHub** **view raw**

In practice, before the submission the jobs to Ganga/Grid, the code needs to be tested using some test-data. This, formally unnesessary, but very important step can be easily embedded into your module using python's `__main__` clause:

```
if '__main__' == __name__ :
    print 'This runs only if module is used as the script! '
    configure ( [] , catalogs = [] , params = {} )
    run ( 10 )
```

Note that these lines effectively convert the *module* into *script*, and finally one gets:

```
## 1) some user code :
def run ( nEvents ) :
    for i in range( 0 , min( nEvents , 10 ) ) : print ' I run event %i ' % '
    return 0
## 2) configuration step
def configure (  datafiles ,  catalogs = [] , castor = False , params = {} ) :
    print 'I am configuration step!'
    return 0
## 3) steer the job
if '__main__' == __name__ :
    print 'This runs only if module is used as the script!'
    run ( 10 )
```

DoNothing.py hosted with ❤ by **GitHub** **view raw**

How to run it interactively?

The answer is trivial:

```
lb-run Bender/prod python DoNothing.py
```

That's all. Make a try and see what you get!

Unnesessary but very useful decorations:

It is highly desirable and *recommended* to put some *"decorations"* a top of this minimalistic lines:

- add magic `#!/usr/bin/env python` line as the top line of the module/script
- make the script executable: `chmod +x ./DoNothing.py`
- add a python documentation close to the begin of the script
  - fill some useful python attributes with the proper informaton
    - `__author__`
    - `__date__`
    - `__version__`
  - do not forget to add documenation in Doxygen-style and use in comments following tags
    - `@file`
    - `@author`
    - ...

With all these decorations the complete module is here

For all subsequent lessons we'll gradually extend this script
with the additional functionality, step-by-step converting it to something much more useful.

In practice, ...

In practice, the prepared and *ready-to-use* function `run` is imported from some of the main Bender module `Bender.Main`, and the only one really important task for the user is to code the function `configure`.

DaVinci

Learning objectives

- Understand the internal structure of the *configure* function

For the *typical* case in practice, the function *configure* (as the name suggests) contains three parts

1. *static configuration*: the configuration of `DaVinci` configurable (almost unavoidable)

2. *input data and application manager*: define the input data and instantiate Gaudi's application manager (mandatory)
3. *dynamic configuration*: the configuration of `GaudiPython` components (optional)

Static configuration

For the first part, the instantiation of DaVinci configurable is almost unavoidable step:

```
from    Configurables import DaVinci
rootInTES = '/Event/PSIX'
dv = DaVinci ( DataType   = '2012'    ,
               InputType  = 'MDST'    ,
               RootInTES  = rootInTES )
```

Here we are preparing application to read `PSIX.MDST` - uDST with few useful selections for B&Q Working Group. Note that in this part one can use all power of DaVinci/Gaudi `Congifurables` . In practice, for physics analyses, it is veyr convinient to use here `Selection` framework, that allows to configure `DaVinci` in a very compact, safe, robust and nicely readable way, e.g. let's get from Transient Store some `selection` and print its content

```
from PhysConf.Selections import AutomaticData,  PrintSelection
particles = AutomaticData  ( 'Phys/SelPsi2KForPsiX/Particles' )
particle  = PrintSelection ( particles )
```

As the last sub-step of (1), one needs to pass the selection object to `DaVinci`

```
dv.UserAlgorithms.append ( particles )
```

Where is `SelectionSequence` ?

The underlying `SelectionSequence` object will be created automatically. You should not worry about it.

Input data and application manager

This part is rather trivial and almost always standard:

```
from Bender.Main import setData, appMgr
## define input data
setData  ( inputdata , catalogs , castor )
## instantiate the application manager
gaudi = appMgr()  ## NOTE THIS LINE!
```

while `setData` can appear anywhere inside `configure` function, the line with `appMgr()` is very special. After this line, no *static configuration* can be used anymore. Therefore all the code dealing with `Configurables` and `Selections` must be placed above this line.

Dynamic configuration

For this particular example, it is not used, but will be discussed further in conjunction with other lessons.

The complete `configure` function is:

```
1  from Bender.Main import setData, appMgr, SUCCESS
2  ## The configuration of the job
3  def configure ( inputdata        ,    ## the list of input files
4                  catalogs = []    ,    ## xml-catalogs (filled by GRID)
```

```
                castor   = False ,    ## use the direct access to castor/EOS ?
                params   = {}    ) :

    ## import DaVinci & configure it!
    from Configurables import DaVinci
    ## delegate the actual configuration to DaVinci
    rootInTES = '/Event/PSIX'
    dv = DaVinci ( DataType   = '2012'    ,
                   InputType  = 'MDST'    ,
                   RootInTES  = rootInTES )

    from PhysConf.Selections import AutomaticData, PrintSelection
    particles  = AutomaticData( 'Phys/SelPsi2KForPsiX/Particles' )
    particles  = PrintSelection ( particles )
    dv.UserAlgorithms.append ( particles  )

    ## define the input data
    setData  ( inputdata , catalogs , castor )

    ## get/create application manager
    gaudi = appMgr()
    return SUCCESS
```

DaVinciEx.py hosted with ❤ by **GitHub** **view raw**

The prepared and *ready-to-use* function `run` is imported `Bender.Main`:

```
from Bender.Main import run
```

Now our Bender module (well, it is actually pure `DaVinci`, no real Bender here!) is ready to be used with Ganga/Grid. For local interactive tests we can use the trick with `__main__` clause: The `__main__` clause in our case contains some input data for local tests:

```
if __name__ == '__main__' :
    inputdata = [
        '/lhcb/LHCb/Collision12/PSIX.MDST/00035290/0000/00035290_00000221_1.psix.mdst' ,
        '/lhcb/LHCb/Collision12/PSIX.MDST/00035290/0000/00035290_00000282_1.psix.mdst' ]
    configure( inputdata , castor = True )
    ## the event loop
    run(10000)
```

The complete moodule can be accessed here

How to run it?

Again, the answer is trivial (and universal):

```
lb-run Bender/prod python DoNothing.py
```

That's all. Make a try and see what you get!

Challenge

Try to convert any of your existing `DaVinci` simple *script* into Bender *module* and run it interactively. You can use the result of this excersize for subsequent lessons.

What is `castor` ? Why `LFN` is used as input file name?

Bender is smart enough, and for many cases it can efficiently convert input `LFN` into the real file name.

1. First, if you have Grid proxy enabled (`lhcb-proxy-init`) is uses internally `LHCbDirac` to locate and access the file. This way is not very fast, but for all practial cases this look-up is almost always successful, however for some cases certain hints could be very useful. In particular, you can specify the list of Grid sites to look for data files:

   ```
   ## define input data
   setData  ( inputdata , catalogs , castor = castor ,  grid = ['RAL','CERN','GRIDKA'] )
   ```

2. Second, for CERN, one can use option `castor = True` , that activates the local look-up on input files at CERN-CASTOR and CERN-EOS storages (`root://castorlhcb.cern.ch` and `root://eoslhcb.cern.ch`). This look-up is much faster than the first option, but here the success is not guaranteed, since not all files have their replicas at CERN.
3. Third, for access to special locations, e.g. some local files, Bender also makes a try to look into directories specified via the environment variable `BENDERDATAPATH` (column separated list of paths) and also try to contruct the file names using the content of environment variable `BENDERDATAPREFIX` (semicolumn separated list of prefixes used for constryction the final file name). The file name is constructed using all `(n+1)*(m+1)` variants, where `n` is number of items in `BENDERDATAPATH` and `m` is number of items in `BENDERDATAPREFIX` . Using the combination of `BENDERDATAPATH` and `BENDERDATAPREFIX` variables one can make very powerful matching of *short* file names (e.g. LFN) to the actual file. Using these variables one can easily perform a local and efficient access to Grid files from some *close* Tier-1/2 center.

Keypoints

Wth these two examples, you should aready be able to

- code sove *valid* (but useless) Bender modules
- run them interactively

The first Bender algorithms

Prequisites

- One needs to undertand the stucture of Bender *module* : `run` , `configure` functions and the `__main__` clause
- One needs to know the sctructure and the content of `configure` function

Learning objectives

- Understand Bender algorithms
  - How to code them?
  - How to embedd them into the application?

Hello, world!

Traditionally for tutorials, the first algorithm prints `Hello, world` . The Bender algorithm inherits from the class `Algo` , imported from `Bender.Main` module. This python base is indeed a `C++` -class, that inherits from `LoKi::Algo` class, that in turn inherits from `DaVinciTupleAlgorithm` . The simplest algorithm is rather trivial:

```
from Bender.Main iport Algo, SUCCESS
class HelloWorld(Algo):
    """The most trivial algorithm to print 'Hello,world!'"""
    def analyse( self ) :   ## IMPORTANT!
        """The main 'analysis' method"""
        print       'Hello, world! (using native Python)'
        self.Print( 'Hello, World! (using Gaudi)')
        return SUCCESS      ## IMPORTANT!!!
```

Important note:

- one *must* implement the method `analyse` that gets no argument and returns `StatusCode`

Optionally one can (re)implement other important methods, like `__init__` , `initialize` , `finalize` , etc... In particular `initialize` could be used to locate some *tools* and or pre-define some useful code fragments, e.g. some *expensive* or non-trivial LoKi-functors.

Where to put the algorithm code?

It is recommended to put the algorithm code directly in the main body of your module, outside of `configure` function. It allows to have visual separation of the algorithmic and configuration parts. Also it helps for independent reuse of both parts.

How to embedd the algorithm into the application ?

There are two approaches *brute-force*, that works nicely with such primitive code as `HelloWorld` algorithm above and the

intelligent/recommended approach, that smoothly insert the algorithm into the overall flow of algorithms, provided by `DaVinci`

Brute-force

One can instantiate the algorithm in **configure** method **after** the instantiation of application manager, and add the algorithm, into the list of top-level algorithms, known to Gaudi:

```
gaudi = appMgr()
alg   = HelloWorld('Hello')
gaudi.addAlgorithm( alg )
```

For this particular simple case one can also just replace the list of top-level Gaudi algorithms with a single `HelloWorld` algorithm:

```
gaudi = appMgr()
alg   = HelloWorld('Hello')
gaudi.setAlgorithms( [ alg ] )
```

More on an optional _dynamic configuration_

As it has been said earlier, the part of `configure` function, placed after `gaudi=appMgr()` line corresponds to *dynamic configuration*, and here one can continue the further configuration of the algorthm, e.g.

```
gaudi    = appMgr()
alg      = HelloWorld('Hello')
alg.QUQU = 'qu-qu!'  ## define and set some "parameter"
gaudi.setAlgorithms( [ alg ] )
```

Later, this new *parameter* can be accessed e.g. in `analyse` function:

```
class HelloWorld(Algo):
    """The most trivial algorithm to print 'Hello,world!'"""
    def analyse( self ) :   ## IMPORTANT!
        """The main 'analysis' method"""
        print       'Hello, world! (using native Python)', self.QUQU ## use "parameter"
        self.Print( 'Hello, World! (using Gaudi)')
        return SUCCESS      ## IMPORTANT!!!
```

Such trick is in general a bit fragile, but it is often useful if one has several instances of the algorithm that differ only by some configuration parameter.

```
alg1 = MyALG ( ... )
alg2 = MyALG ( ... )
alg3 = MyALG ( ... )
alg1.decay_mode = '[D0 -> K-  pi+]CC'
alg2.decay_mode = '[D0 -> K-  K+ ]CC'
alg3.decay_mode = '[D0 -> pi- pi+]CC'
```

This approach is very easy and rather intuitive, but is not so easy to insert the algorithm into existing non-trivial flow of algorithms without a danger to destroy the flow. In this way one destroys various standard actions, like (pre)filtering, luminosity calculation etc., therfore it could not be recommended for the real physics analyses, but it could be used for some simple special cases.

Intelligent approach

For *intelligent* approach one uses `Selection` wrapper for Bender algotithm, `BenderSelection`. This wrapper behaves as any other

selection-objects, and it lives in *static configuration* part of `configure` function:

```
from PhysConf.Selections import AutomaticData,  PrintSelection
particles = AutomaticData  ( 'Phys/SelPsi2KForPsiX/Particles' )
particle  = PrintSelection ( particles )

## configuration object for Bender algorithm:
#                             name   , input selections
hello = BenderSelection ( 'Hello' , inputs = [ particle ]  )
dv.UserAlgorithms.append ( hello )
```

As the next step in *dynamic configuration* part of `configure` function one instantiates the algoritm taking all the configuration from the selection-object:

```
gaudi = appMgr()
alg   = HelloWorld( hello )
```

To complete the module one (as usual) need to combine in the file

1. implementation of `HelloWorld` algorithm
2. `configure` function with proper *static* and *dynamic* configurations
3. `__main__` clause
4. (`run` function is imported from `Bender.Main` module)

The complete module can be accessed here

Get data, fill histos & n-tuples

Well, now your Bender algorithm knows how to print `Hello,world!` . Note that it also gets some data: in the previous example we fed it with `particles` -selection. Now try to get this data inside the algorithm and make first simpel manipulations with data.

`select` method

The method `select` is a heart of Bender algorithm. It allows to select/filter the particles that satisfies some criteria from the input particles. The basic usage is:

```
myB = self.select ( 'myB' , ('B0' ==  ABSID ) | ('B0' ==  ABSID ) )
```

The method returns collection filtered particles. The first argument is the tag, that will be associated with selected particles, the second argument is the selection criteria. The tag *must* be unique, and the selection criteria could be in a form of

- *LoKi predicate*: LoKi-functor that get the particle as argument and return the boolean value
- *decay descriptor* , e.g. `'Beauty --> J/psi(1S) K+ K-'` . Some components of the decay descriptor can be *marked*, and in this case, only the *marked* partcles will be selected:

  ```
  myB = self.select ( 'beauty' , 'Beauty --> J/psi(1S)  K+  K-')   ## get the heads of the decay
  myK = self.select ( 'kaons'  , 'Beauty --> J/psi(1S) ^K+ ^K-')   ## get only kaons
  ```

As soon as one gets some good, filtered particles there are many possible actions

- print it!

  ```
  myB = self.select ( 'myB' , ('B0' ==  ABSID ) | ('B0' ==  ABSID ) )
  print myB
  ```

- loop

```
myB = self.select ( 'myB' , ('B0' ==  ABSID ) | ('B0' ==  ABSID ) )
for b in myB :
 print 'My Particle:', p
 print 'some quantities: ', M(p) , PT(p) , P(p)
```

- fill histograms

```
myB = self.select ( 'myB' , ('B0' ==  ABSID ) | ('B0' ==  ABSID ) )
for b in myB :
 #          what        histo-ID    low-edge  high-edge  #bins
 self.plot( PT (p)/GeV , 'pt(B)'  , 0        , 20       ,  50  )
 self.plot( M  (p)/GeV , 'm(B)'   , 5.2      , 5.4      , 100  )
 self.plot( M1 (p)/GeV , 'm(psi)' , 3.0      , 3.2      , 100  )
 self.plot( M23(p)/GeV , 'm(KK)'  , 1.0      , 1.050    ,  50  )
```

- fill n-tuple:

```
myB = self.select ( 'myB' , ('B0' ==  ABSID ) | ('B0' ==  ABSID ) )
t = self.nTuple('TupleName')
for b in myB :
 t.column_float ( 'pt'    , PT (p)/GeV)
 t.column_float ( 'm'     , M  (p)/GeV)
 t.column_float ( 'm_psi' , M1 (p)/GeV)
 t.column_float ( 'm_kk'  , M23(p)/GeV)
 t.write()
```

For n-tuples...

Sicne n-tuples(ROOT's `TTree` objects) resides in ROOT-file, to use n-tuples, one also need to declare the output file for `TTree`s: The easiest way is to rely on `TupleFile` property of `DaVinci` :

```
dv = DaVinci ( DataType   = '2012'          ,
               InputType  = 'MDST'          ,
               TupleFile  = 'MyTuples.root' ,  ## SEE HERE !!!
               RootInTES  = rootInTES       )
```

Challenge

Add `select` statements, histos and tuples to your `HelloWorld` algorithm, created earlier, and run it.

Solution

The complete module is accesible here

Keypoints

Wth these two examples, you should aready be able to

- code Bender *algorithms* and insetr them into overall algorithm flow
- loop over data, fill historgrams and n-tuples

Create the compound particles in Bender

Prequisites

- One needs to understand the way how Bender accesses the data

Learning objectives

- Understand how Bender algorithm combines the particles and creates the compound particles

Make-B

The next example illustrates how one combines the particles and create the compound particles inside the Bender algorithm. Let's consider a simple case of creation of `B+ -> J/psi(1S) K+` decays.

The first step is rather obvious: before getting the combinations, we need to get the individual components. Here `select` function does the job:

```
## get J/psi mesons from the input
psis  = self.select ( 'psi' , 'J/psi(1S) -> mu+ mu-' )
## get energetic kaons from the input:
kaons = self.select ( 'k'  , ( 'K+' == ID ) & ( PT > 1 * GeV ) &  ( PROBNNk > 0.2 ) )
```

The loop over `psi k` combinations is ratehr trivial:

```
## make a loop over J/psi K combinations :
for b in self.loop( 'psi k' , 'B+' ) :
    ## fast evaluation of mass (no fit here!)
    m12 = b.mass ( 1 , 2 ) / GeV
    print 'J/psiK mass is %s[GeV]'  % m12
    p1  = b.momentum ( 1 ) / GeV
    p2  = b.momentum ( 2 ) / GeV
    p12 = b.momentum ( 1 , 2 ) / GeV
    print 'J/psiK  momentum is is %s[GeV]'  % p12
```

Looping object (`b` here), as the name, suggests, make a loop over all `psi k` combinations. The loop is done in CPU efficient way, and no expensive vertex fitting is performed. One can estimate various *raw* (no fit) kinematical quantities using functions `momentum` , `mass` , etc... (Note that indices starts from `1` . For all LoKi-based functions the index `0` is reserved for *self-reference*, the mother particle itself). These `raw` quantities can be used for quick reject of *bad* combinations before making CPU-expensive vertex fit.
If/when combination satisfies certain criteria, the vertex
fit and creatino of the compound particle is triggered automatically if any of particle/vertex information is retrieved (either directy via `particle/vertex` method, or indirectly, e.g. via call to any *particle/vertex LoKi-functor*. The good created *mother* particles can be saved for subsequent steps under some unique tag:

```
for b in self.loop( 'psi k' , 'B+' ) :
    ## fast evaluation of mass (no fit here!)
    m12 = b.mass ( 1 , 2 ) / GeV
    if not 5 < m12 <   6    : continue
    chi2vx = VCHI2 ( b )   ## indirect call for vertex fitr and creation of B+ meson
    if not 0<= chi2v <  20  : continue
    m = M ( p ) / GeV
    if not 5 <  m < 5.6     :  continue
    m.save('MyB')
```

Obviously the looping can be combnied with filling of historgams and n-tuples.

How to deal with charge conjugation?

One can make two loops:

```
psis    = self.select ( 'psi' , 'J/psi(1S) -> mu+ mu-' )
kplus   = self.select ( 'k+'  , ( 'K+' == ID ) & ( PT > 1 * GeV ) &  ( PROBNNk > 0.2 ) )
kminus  = self.select ( 'k-'  , ( 'K-' == ID ) & ( PT > 1 * GeV ) &  ( PROBNNk > 0.2 ) )

bplus   = self.loop( 'psi k+', 'B+' ) ## the first  loop object
bminus  = self.loop( 'psi k-', 'B-' ) ## the second loop object
for cc in ( bplus , bminus ) :
   for b in cc :
      m12 = b.mass(1,2) / GeV
      ...
      b.save('MyB')
```

The popular alternative is *charge-blind* loop, that is a bit simpler, but it requires some accuracy:

```
psis    = self.select ( 'psi' , 'J/psi(1S) -> mu+ mu-' )
## ATTENTION: select both K+ and K-, note ABSPID here
k       = self.select ( 'k'   , ( 'K+' == ABSID ) & ( PT > 1 * GeV ) &  ( PROBNNk > 0.2 ) )

for b in self.loop ( 'psi k' , 'B+' )
      m12 = b.mass(1,2) / GeV
      ...
      psi = b(1) ## get the first daughetr
      k   = b(2) ## get the second daughter

      ## ATTENTION: redefine PID on-flight
      if Q( k ) > 0 : b.setPID( 'B+' )
      else          : b.setPID( 'B-' )

      b.save('MyB')
```

The saved particles can be extracted back using the method `selected` :

```
myB = self.selected('MyB')
for b in myB :
   print M(b)/GeV
```

Configuration

It is clear that to build `B+ -> J/psi(1S) K+` decays, one needs to
feed the algorithm with J/psi(1S)-mesond and kaons. Using `Selection` machinery is the most efficient ansd transaprent way to do it.

```
from PhysConf.Selections import AutomaticData
jpsi = AutomaticData( '/Event/Dimuon/Phys/FullDSTDiMuonJpsi2MuMuDetachedLine/Particles' )

from StandardParticles   import StdLooseKaons as kaons
bsel = BenderSelection ( 'MakeB' , [ jpsi , kaons ] )
```

The complete example of creation of `B+ -> J/psi(1S) K+` decays starting from `DIMUON.DST` is accessible from here

Keypoints

Wth these example, you should be able to do

- code Bender *algorithm* that perofem loopint, combianig anc creation of compound particles

Advanced Bender

Here we discuss some *advanced* features of Bender, namely

- the powefull *fill* of n-tuples provied by `BenderTools.Fill` module
- the treatment of `TisTos` information in Bender, provided by the module `BenderTools.TisTos`
- using of *tools*

Advanced fill of n-tuples

The n-tuple filling functionality, described above is drasticlaly extended using the functions from `BenderTools.Fill` module. The import of this module add following functions to the base class `Algo` :

Method	Short description
`treatPions`	add information about pions
`treatKaons`	add information about kaons
`treatProtons`	add information about protons
`treatMuons`	add information about muons
`treatPhotons`	add information about photons
`treatDiGammas`	add information about di-photons (pi0, eta,...)
`treatTracks`	add information about the tracks
`treatKine`	add detailed kinematic information for the particle
`fillMasses`	masses of sub-combinations
`addRecSummary`	add rec-summary information
`addGecInfo`	add some GEC-info

These methods can be considered as a kind of very light *tuple-tools*. All of them are (well) documented and one can easily inspect them:

```
import BenderTools.Fill
from Bender.Main import Algo
help(Algo.treatPions)
```

Also all these methods print detailed how-to infomratino in log-file at the moment of the first invoke, and it vasn be very helpful to understand the branches in n-tuple/tree, e.g.

```
# BenderTools.Fill          INFO    treatTracks: The method adds track-specific information into n-tuple
#     ...
#     tup = ... ## n-tuple
#     b   = ... ## the particle  (or looping object)
#     self.treatTracks ( tup , b , '_B' ) ## suffix is optional
#     ...
#     Following variables are added into n-tuple:
#     - deltaM2_min_track_ss/os[+suffix]:
#     Minimal value of delta_m2(track1, track2) for all pairs of same-sign (``_ss'')
#     and opposite sign ``_os'' tracks, where function minm2 is
#     delta_M2(p1,p2) = (m^2(p1+p2) - 2*m^2(p1)-2*m^2(p2) )/m^2(p1+p2)
#     see  LoKi::Kinematics::deltaM2
#     - deltaAlpha_min_track_ss/os[+suffix]:
#     Minimal value of the angle between two momenta for all pairs of same-sign (``_ss'')
#     and opposite sign ``_os'' tracks
#     see  LoKi::Kinematics::deltaAlpha
#     - overlap_max_track_ss/os[+suffix]:
#     Maximal value ``overlap'' for all pairs of same-sign (``_ss'')
#     and opposite sign ``_os'' tracks
#     ``Overlap'' is defined as fraction of common/shared hits between two tracks
#     see LHCb::HasIDs::overlap
#     - minPt_track[+suffix]
#     Minimal pT of the tracks
#     - min/maxEta_track[+suffix]
#     Minimal/maximal eta/pseudorapidity of the tracks
#     - maxChi2_track[+suffix]
#     Maximal chi2/ndf for the track
#     - minKL_track[+suffix]
#     Minimal value of Kullback-Leibler divergency for the tracks
#     - maxTrGh_track[+suffix]
#     Maximal value of Track Ghost probability for the tracks (track-based)
#     - maxAnnGh_track[+suffix]
#     Maximal value of       Ghost probability for the tracks (PID-based)
#     - n_track[+suffix]
#     Number of tracks in the decay
#
#     And then for each track in the decay:
#     - p_track[+suffux]     momentum of the track
#     - pt_track[+suffux]    transverse momentum of the track
#     - eta_track[+suffux]   eta/pseudorapidity  of the track
#     - phi_track[+suffux]   phi (azimuth angle) of the track
#     - chi2_track[+suffux]  chi2/ndf of the track
#     - PChi2_track[+suffux] fit probability calculated from chi2/ndf of the track
#     - ann_track[+suffix]   Ghost probability (PID-based)
#     - trgh_track[+suffix]  Track Ghost probability (Track-based)
```

The typical usage of these methods is:

```
tup = self.nTuple('MyTuple')
for p in particles :

    psi = p(1) ## the first daughter: J/psi

    ## fill few kinematic variables for the particles:
    self.treatKine   ( tup , p   , '_b'   )  ## use the suffix to mark variables
    self.treatKine   ( tup , psi , '_psi' )  ## use the suffix to mark variables

    self.treatKaons  ( tup , p ) ## fill some basic information for all kaons
    self.treatMuons  ( tup , p ) ## fill some basic information for all muons
    self.treatTracks ( tup , p ) ## fill some basic information for all charged tracks

    tup.write()
```

Challenge

1. Add (some of) these functions into your previous Bender module with n-tuples.
2. Run it and observe the detailed printout in log-file
3. Observe new variables in your n-tuple/tree and find their description in the log-file or via `help(Algo.<THEMETHOD>)`
    - Is the description for all new varibales clear enough?

Solution

The complete module is accessible here and the corresponsing log-file is here

Handling of `TisTos` -information in Bender

Bender offers set of methods to handle `TisTos` -information in (relatively) easy way. This functionality comes from `BenderTools.TisTos` module. In short, it adds three relates method in the base class `Algo` :

Method	Short description
`decisions`	collect the trigger decisions for the given particle
`trgDecs`	print the collected trigger statistics in readable way
`tisTos`	fill N-tuple with `TisTos` -information

All of them are (relatively well) documented and one can easily inspect them:

```
import BenderTools.TisTos
from Bender.Main import Algo
help(Algo.decisions)
```

How to know what trigger lines are relevant or the given decay/particle?

Often information about the releavant trigger lines are spread in corridors in a form of myths, general beliefs or, in the best case, references to ANA-notes for some similar analysis. However it is very simple to collect this infomation using Bender. The method `decisions` is our friend here. the usage of this method require some preparatory work for the algorithm, namely one needs to instrument `initialize` -method:

```
class TrgLines(Algo):
    """Collect infomation about the trigger lines relevant for certain decays/particles
    """
    def initialize ( self ) :

        sc = Algo.initialize ( self ) ## initialize the base class
        if sc.isFailure() : return sc

        #
        ## container to collect trigger information, e.g. list of fired lines
        #
        triggers = {}
        triggers ['psi'] = {} ## slot to keep the information for J/psi
        triggers ['K']   = {} ## slot to keep the information for kaons
        triggers ['B']   = {} ## slot to keep the information for B-mesons

        sc = self.tisTos_initialize ( triggers , lines = {} )
        if sc.isFailure() : return sc

        return SUCCESS
```

Here in this example we want to collect `TisTos` -information for `J/psi` , `K` and the whole `B+` -meson. Then in the main `analyse` -method one just needs to invoke the method `decisions` for each particles in interest:

```
def analyse( self ) :   ## IMPORTANT!
  """The main 'analysis' method """
  ...
  for b in particles :

      psi =b(1) ## the first  daughter: J/psi
      k   =b(2) ## the second daughter: K

      ## collect trigger information for J/psi
      self.decisions   ( psi , self.triggers['psi'] )

      ## collect trigger information for kaons
      self.decisions   ( k   , self.triggers['K']   )

      ## collect trigger information for B-mesons
      self.decisions   ( b   , self.triggers['B']   )

  ...
  return SUCCESS
```

Thats all. Then when jobs runs it dumps to the log-file the running trigger statistics, and the statistics is dumped into the file `TrgLines_tistos.txt` (`<ALGNAME>_tistos.txt` in general). The summary table looks like:

```
********************************************************************************************
 Triggers for  psi
********************************************************************************************
Hlt1_TIS psi   #lines:     7 #events 321
 (  0.62 +- 0.44 )       Hlt1DiMuonHighMassDecision
 (  0.62 +- 0.44 )       Hlt1DiMuonLowMassDecision
 ( 19.00 +- 2.19 )       Hlt1TrackAllL0Decision
 (  3.74 +- 1.06 )       Hlt1TrackAllL0TightDecision
 (  4.36 +- 1.14 )       Hlt1TrackMuonDecision
 (  0.93 +- 0.54 )       Hlt1TrackPhotonDecision
 (  0.31 +- 0.31 )       Hlt1VertexDisplVertexDecision
 (100.00 +- 0.31 )       TOTAL
Hlt1_TOS psi   #lines:     9 #events 321
 ( 72.90 +- 2.48 )       Hlt1DiMuonHighMassDecision
 ( 59.81 +- 2.74 )       Hlt1DiMuonLowMassDecision
 (  0.31 +- 0.31 )       Hlt1DiProtonDecision
 (  9.03 +- 1.60 )       Hlt1SingleMuonHighPTDecision
 (  0.31 +- 0.31 )       Hlt1SingleMuonNoIPDecision
 ( 44.24 +- 2.77 )       Hlt1TrackAllL0Decision
 ( 10.90 +- 1.74 )       Hlt1TrackAllL0TightDecision
 ( 74.77 +- 2.42 )       Hlt1TrackMuonDecision
 (  0.31 +- 0.31 )       Hlt1TrackPhotonDecision
 (100.00 +- 0.31 )       TOTAL
Hlt1_TPS psi   #lines:     6 #events 321
 ( 13.71 +- 1.92 )       Hlt1DiMuonHighMassDecision
 ( 10.90 +- 1.74 )       Hlt1DiMuonLowMassDecision
 (  0.31 +- 0.31 )       Hlt1DiProtonDecision
 (  3.12 +- 0.97 )       Hlt1TrackAllL0Decision
 (  0.93 +- 0.54 )       Hlt1TrackAllL0TightDecision
 (  4.67 +- 1.18 )       Hlt1TrackMuonDecision
 (100.00 +- 0.31 )       TOTAL
```

Only a short fragment is shown here, one gets similar fragments for all declared particles (`psi` , `K` and `B`) and for all trigger levels (`L0` , `Hlt1` and `Hlt2`). The full table is accessible here The content of the summaty table is rather intuitive: it summarizes the fire frequences for varios trugegr lines for three regimes `TIS` , `TOS` and `TPS` . Inspecting such table, one immediately concludes that the most relevan `Hlt1-TOS` -line is `Hlt1DiMuonHighMassDecision` . Other `Hlt1-TOS` -lines are less relevant here. But please note that here only very small statistics is used (321 event), and with larger statistics sthe conclusions could be corrected. E.g. due to small statistics here, for `Hlt1-TIS` -lines the choice is not evident: one clearly see that `Hlt1TrackAllL0Decision` line is important, but for importance of other lines one csn judge only after the significant increase of the statistics.

What is `_tistos.db` file?

In practice to make a decision, large statistics is required (for real data and/or for simulated samples). And here these files are very useful. The trigger statistics is saved not only in `<ALGNAME>_tistos.txt` -file but also in `shelve` -dbase `<ALGNAME>_tistos.db` .
If really large statistics is required there are some utilities to merge the information from these `<ALGNAME>_tistos.db` together, e.g. on the output of Ganga jobs.

Challenge

1. Add `decisions` -function for your previous Bender module with n-tuples.
   - (Do not forget to instrument the `initialize` method)
2. Run it and observe the output summary table
3. Identify the relevant `L0-TOS` , `Hlt1-TOS` and `Hlt2-TOS` lines for your decay
   - Does it correspond to your expectations?

Solution

The complete module is accessible here and the corresponsing summary table is here

How to add the `TisTos` -information to n-tuple/tree?

Now, when we have the lists of the relevant lines, one wants to add infrommation about the decisions of these lines into n-tuple/tree. The method `tistos` is out friend here. To use this method one needs to instrument `initialize` -method:

```
class TisTosTuple(Algo):
    """Enhanced functionality for n-tuples
    """
    def initialize ( self ) :

        sc = Algo.initialize ( self ) ## initialize the base class
        if sc.isFailure() : return sc

        #
        ## container to collect trigger information, e.g. list of fired lines
        #
        triggers = {}
        triggers ['psi'] = {} ## slot to keep information for J/psi

        #
        ## the lines to be investigated in details
        #
        lines    = {}
        lines    [ "psi" ] = {} ## trigger lines for J/psi

        ## six mandatory keys:
        lines    [ "psi" ][   'L0TOS' ] = 'L0(DiMuon|Muon)Decision'
        lines    [ "psi" ][   'L0TIS' ] = 'L0(Hadron|DiMuon|Muon|Electron|Photon)Decision'
        lines    [ "psi" ][ 'Hlt1TOS' ] = 'Hlt1(DiMuon|TrackMuon).*Decision'
        lines    [ "psi" ][ 'Hlt1TIS' ] = 'Hlt1(DiMuon|SingleMuon|Track).*Decision'
        lines    [ "psi" ][ 'Hlt2TOS' ] = 'Hlt2DiMuon.*Decision'
        lines    [ "psi" ][ 'Hlt2TIS' ] = 'Hlt2(Charm|Topo|DiMuon|Single).*Decision'

        sc = self.tisTos_initialize ( triggers , lines )
        if sc.isFailure() : return sc

        return SUCCESS
```

Here one defines six `regex`-patterns that describe the six sets of triggers lines: `L0`, `Hlt1`, `Hlt2` vs `TIS`, `TOS`. These expressions are coded according to the infomration, obtainer earlier. The next step is rather trivial: in `analyse` method one need to invoke the method `tistos` for the given particle, J/psi in our case:

```
def analyse( self ) :
    """The main 'analysis' method"""
    ...
    tup = self.nTuple('MyTuple')
    for b in particles :
        psi =b(1) ## the first daughter: J/psi
        ...
        ## fill n-tuple with TISTOS information for J/psi
        self.tisTos      (
            psi               , ## particle
            tup               , ## n-tuple
            'psi_'            , ## prefix for variable name in n-tuple
            self.lines['psi'] , ## trigger lines to be inspected
            verbose = True      ## good ooption for those who hates bit-wise operations
            )
        tup.write()
   ...
   return SUCCESS
```

This code adds several variables into n-tuple/tree, see log-file or use `help(Algo.tistos)`. Also `TisTos`-information for *global* and *physics* triggers is added.

In details,

The fragment from log-file:

```
# BenderTools.TisTos        INFO    tisTos: Fill TisTos information into n-tuple
#
#    # for d in particles :
#    #    self.tisTos ( d     ,
#    #                  tup   ,
#    #                 'd0_'  ,
#    #                 self.lines ['D0'] ,
#    #                 self.l0tistos     ,
#    #                 self.l1tistos     ,
#    #                 self.l2tistos     )
#
#    ``lines'' here is a dictionary of lines (or regex-patterns) with
#    following keys:
#    - L0TOS
#    - L0TIS
#    - Hlt1TOS
#    - Hlt1TIS
#    - Hlt2TOS
#    - Hlt2TIS
#
#    e.g.
#
#    # lines = {}
#    # lines ['L0TOS'  ] = 'L0HadronDecision'
#    # lines ['L0TIS'  ] = 'L0(Hadron|Muon|DiMuon)Decision'
#    # lines ['Hlt1TOS'] = ...
#    # lines ['Hlt1TIS'] = 'Hlt1Topo.*Decision'
#    # lines ['Hlt2TOS'] = ...
#    # lines ['Hlt2TIS'] = ...
#
#    Technically it is useful to keep it as ``per-particle-type'' dictionary
#
#    # def initialize ( self ) :
#    #     ...
#    #     self.lines           = {}
#    #     self.lines ['B'  ]   = {}
#    #     self.lines ['B'  ]['L0TOS'] = ...
#    #     self.lines ['B'  ]['L0TOS'] = ...
#    #     ...
#    #     self.lines ['psi']   = {}
#    #     self.lines ['psi']['L0TOS'] = ...
#    #     ...
#    #     return SUCCESS
#
#    # def analyse ( self ) :
#
#    #     particles = ...
#
#    #     for B in particles :
#    #         self.tisTos ( B     ,
#    #                       tup   ,
#    #                       'B0_' ,
#    #                       self.lines['B']   ,
#    #                       self.l0tistos     ,
#    #                       self.l1tistos     ,
#    #                       self.l2tistos     )
#
#    #     ...
#    #     return SUCCESS
#
#    The function adds few columns to n-tuple, the most important are
#    - ``<label>l0tos'' that corresponds to   'L0-TOS'
#    - ``<label>l0tis'' that corresponds to   'L0-TIS'
#    - ``<label>l1tos'' that corresponds to 'Hlt1-TOS'
#    - ``<label>l1tis'' that corresponds to 'Hlt1-TIS'
#    - ``<label>l2tos'' that corresponds to 'Hlt2-TOS'
#    - ``<label>l2tis'' that corresponds to 'Hlt2-TIS'
#    Additionally information for five predefined lists is stored:
#    - ``<label>l0phys'' : L0-physics lines,
#    - ``<label>l1phys'' : Hlt1-physics lines  (routing bit #46)
#    - ``<label>l2phys'' : Hlt2-physics lines  (routing bit #77)
```

```
#    - ``<label>l1glob'' : Hlt1Global decision
#    - ``<label>l2glob'' : Hlt2Global decision
#
#    The stored value is unsigned short, a bit-representaion of ITisTos::TisTosTob object
#    - see ITisTos
#    - see ITisTos::TisTosTob
#
#    Later, in processing of TTree one can use these flags as :
#
#    >>> tree = ...
#    >>> tree.Draw('M'               )  ## no trigger requirements
#
#    >>> tree.Draw('M', '(l0tos&2)==2')  ## require L0-tos   with respect to the list of 'L0TOS'-lines
#    >>> tree.Draw('M', '(l1tos&2)==2')  ## require Hlt1-tos with respect to the list of 'Hlt1TOS'-lines
#    >>> tree.Draw('M', '(l2tos&2)==2')  ## require Hlt2-tos with respect to the list of 'Hlt2TOS'-lines
#
#    >>> tree.Draw('M', '(l0tis&4)==4')  ## require L0-tis   with respect to the list of 'L0TIS'-lines
#    >>> tree.Draw('M', '(l1tis&4)==4')  ## require Hlt1-tis with respect to the list of 'Hlt1TIS'-lines
#    >>> tree.Draw('M', '(l2tis&4)==4')  ## require Hlt2-tis with respect to the list of 'Hlt2TIS'-lines
#
#    >>> tree.Draw('M', '(l0tos&3)==3')  ## require L0-tus   with respect to the list of 'L0TOS'-lines
#    >>> tree.Draw('M', '(l1tos&3)==3')  ## require Hlt1-tus with respect to the list of 'Hlt1TOS'-lines
#    >>> tree.Draw('M', '(l2tos&3)==3')  ## require Hlt2-tus with respect to the list of 'Hlt2TOS'-lines
#
#    >>> tree.Draw('M', '(l0tos&1)==1')  ## require L0-tps   with respect to the list of 'L0TOS'-lines
#    >>> tree.Draw('M', '(l1tos&1)==1')  ## require Hlt1-tps with respect to the list of 'Hlt1TOS'-lines
#    >>> tree.Draw('M', '(l2tos&1)==1')  ## require Hlt2-tps with respect to the list of 'Hlt2TOS'-lines
#
#    >>> tree.Draw('M', '(l0phys&2)==2') ## require L0-tos   with respect to L0-physics lines
#    >>> tree.Draw('M', '(l0phys&4)==4') ## require L0-tis   with respect to L0-physics lines
#
#    One can avoid bit-wise operations using ``verbose=True'' flag.
#    In this case one gets following boolean variables in n-tuple:
#    - ``<labeltag>_tos''  Trigger On Signal
#    - ``<labeltag>_tis''  Trigger Independently on Signal
#    - ``<labeltag>_tps''  Trigger Partially on Signal
#    - ``<labeltag>_tus''  Trigger Used Signal (== TOS || TPS )
#    - ``<labeltag>_dec''  Trigger decision
#
#    where ``<labeltag>'' is
#    - ``<label>l0tos''
#    - ``<label>l0tis''
#    - ``<label>l0phys'
#    - ``<label>l1tos''
#    - ``<label>l1tis''
#    - ``<label>l1phys'
#    - ``<label>l1glob'
#    - ``<label>l2tos''
#    - ``<label>l2tis''
#    - ``<label>l2phys'
#    - ``<label>l2glob'
```

Challenge

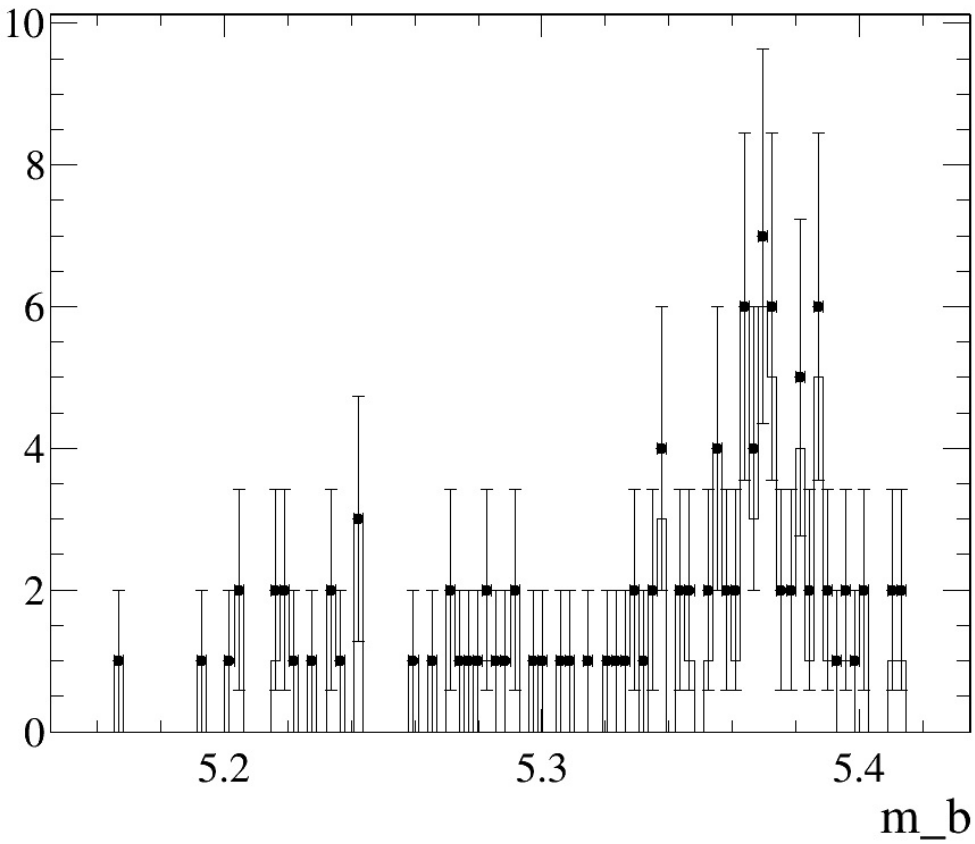
1. Add `tistos`-function for your previous Bender module with n-tuples.
   - (Do not forget to instrument the `initialize` method)
2. Run it and observe new variables in n-tuple/tree
3. Make a plot of B-mass for all candidates, and for candiated that are `L0-TOS`, `Hlt-TOS` and `Hlt2-TOS` with respect to the set of selected trigger lines.

Solution

The complete module is accessible here and the corresponsing log-file is here To make the corresponsing plot, e.g. start `(i)python` :

```
import ROOT
f = ROOT.TFile('TisTosTuples.root','READ')
t = f.Get('TisTos/MyTuple')
t.Draw ( 'm_b'  ,'' , 'e1' )
t.Draw ( 'm_b' , ' psi_l0tos_tos && psi_l1tos_tos && psi_l2tos_tos' , 'same hist' )
```

The result is here:



Using *tools* in Bender

Usage of *tools* and *services* in Bender is rather trivial

1. One aquires *tools* and *services* usin the method `tool` and `svc` (that are very smilar to thir C++ counterpartners `GaudiAlgorithm::tool` and `GaudiAlgorithm::service`). Typically *tools* and *services* are aquired in `initialize` method, e.g.

```
class TupleTools(Algo):
"""Demonstration how to use TupleTools with Bender"""
def initialize ( self ) :
   sc = Algo.initialize ( self ) ## initialize the base class
   if sc.isFailure() : return sc
   IPTT = cpp.IParticleTupleTool
   IETT = cpp.IEventTupleTool
   self.t1 = self.tool( IPTT,'TupleToolPid'      , parent = self ) ## <--- HERE!
   if not self.t1 : return FAILURE
   self.t2 = self.tool( IPTT,'TupleToolGeometry' , parent = self ) ## <--- HERE!
   if not self.t2 : return FAILURE
   self.e1 = self.tool( IETT,'TupleToolBeamSpot' , parent = self ) ## <--- HERE!
   if not self.e1 : return FAILURE
   return SUCCESS
```

2. Then *tools* and *services* can be direcly used in `analyse` -method:

```
## the main 'analysis' method
def analyse( self ) :   ## IMPORTANT!
...
tup = self.nTuple('MyTuple')
for b in particles :
   psi = b ( 1 ) ## the first daughter: J/psi
   ## Particle Tuple Tool
   sc = self.t1.fill ( b , psi , 'psi_' , tup )
   if sc.isFailure() : return sc
   ## Particle Tuple Tool
   sc = self.t2.fill ( b , b   , 'b_'   , tup )
   if sc.isFailure() : return sc
   ## Event Tuple Tool
   sc = self.e1.fill ( tup )
   if sc.isFailure() : return sc
   tup.write()
```

Note that many *standard tools* and *standard services* are direclty accessible via the base class `DVAlgorithm`, e.g.

```
vx_fitter = self.vertexFitter()
dcalc     = self.distanceCalculator()
pp        = self.ppSvc()  ##  particle property service
```

Using *tuple tools* in Bender

Challenge for the fans of tuple tools use

1. Add set of known *tuple tools* into your example
   - (Do not forget to instrument the `initialize` method to aquire tools)
2. Run it and observe new varianles into n-tuple.
   - Do their names correspond to your expectations?

Solution

The complete module is accessible here The structure of n-tuple is:

```
In [7]: from Ostap.Logger import multicolumn
In [8]: f = ROOT.TFile('TupleTools.root','read')
In [9]: t = f.TupleTools.MyTuple
In [10]: print multicolumn ( t.branches() )
BeamX              b__ENDVERTEX_COV_  b__ENDVERTEX_Y     b__FDCHI2_OWNPV    b__OWNPV_CHI2      b__OWNPV_XERR
 b__OWNPV_ZERR
BeamY              b__ENDVERTEX_NDOF  b__ENDVERTEX_YERR  b__FD_OWNPV        b__OWNPV_COV_      b__OWNPV_Y
 psi__ID
b__DIRA_OWNPV      b__ENDVERTEX_X     b__ENDVERTEX_Z     b__IPCHI2_OWNPV    b__OWNPV_NDOF      b__OWNPV_YERR
b__ENDVERTEX_CHI2  b__ENDVERTEX_XERR  b__ENDVERTEX_ZERR  b__IP_OWNPV        b__OWNPV_X         b__OWNPV_Z
```

Processing of simulated data in Bender

Processing of simulation data in Bender is rather simple, one just needs to inherit the algorithm from base class `AlgoMC`, this class can be imported from `Bender.MainMC` module.

```
from Bender.MainMC import *  # it imports also the whole content of Bender.Main module
class MyAlg(AlgoMC):
```

And the corresponsing wrapper for `Selection` -framework is `BenderMCSelection`

Important notes: `Simulation=True` and `DDDB/SIMCOND` -tags

- One needs to use `Simulation=True` flag for `DaVinci` -configurable

```
from Configurables import DaVinci
dv = DaVinci ( Simulation      = True           , ## <--- HERE!
           ...
           TupleFile       = 'MCtruth.root' )
```

- It is *very* important to specify the correct `DDDB/SIMCOND` -tags for the simulated data. It is very easy to get efficiencies wrong up to 30% if simulated data a processed with the wrong `DDDB/SIMCOND` -tags.

```
from Configurables import DaVinci
dv = DaVinci ( Simulation      = True                     , ##
           ...
           DDDBtag         = 'dddb-20130929-1'         , ## <--- HERE!
           CondDBtag       = 'sim-20130522-1-vc-mu100' , ## <--- HERE!
           ...
           TupleFile       = 'MCtruth.root' )
```

Correct `DDDB/SIMCOND` -tags can be retrived in several ways:

1. from `bookkeeping-DB` for the given production type

   ### Challenge (only for those who knows how to do it)

   Do you know how to do it? If so make a try to use this way.

   - Please use the timer for comparison.

2. using the helper Bender scripts `get-dbtags` or `get-metainfo` for the given file

   ### Challenge

   Try to use these scripts form the command line.

   - Start with `get-dbtags -h` and `get-metainfo -h` and follow the instructions.

   ### Solution

   ```
   1  [] % ./run get-dbtags /lhcb/MC/2012/ALLSTREAMS.DST/00033494/0000/00033494_(
   ```

```
# get-dbtags                INFO    ************************************************************
...
# get-dbtags                INFO    ************************************************************
# BenderTools.GetDBtags     INFO    Use the file PFN:root://eoslhcb.cern.ch//eos/lhcb/grid/p
# BenderTools.GetDBtags     INFO    ------------+------------------------------------------
# BenderTools.GetDBtags     INFO     Tags:
# BenderTools.GetDBtags     INFO    ------------+------------------------------------------
# BenderTools.GetDBtags     INFO           DDDB : dddb-20130929-1
# BenderTools.GetDBtags     INFO        SIMCOND : sim-20130522-1-vc-mu100
# BenderTools.GetDBtags     INFO    ------------+------------------------------------------
# BenderTools.GetDBtags     INFO     Input files scanned: 1 from 1
# BenderTools.GetDBtags     INFO     Last (successful) file "/lhcb/MC/2012/ALLSTREAMS.DST/00
# BenderTools.GetDBtags     INFO    ------------+------------------------------------------
```

dbtags.txt hosted with ❤ by **GitHub** **view raw**

```
[] % ./run get-metainfo /lhcb/MC/2012/ALLSTREAMS.DST/00033494/0000/00033494_00000013_1.allst
...
# get-metainfo              INFO    ************************************************************
# BenderTools.GetDBtags     INFO    Use the file PFN:root://eoslhcb.cern.ch//eos/lhcb/grid/p
# BenderTools.GetDBtags     INFO    --------------------------------+---------------------
# BenderTools.GetDBtags     INFO     MetaInfo:
# BenderTools.GetDBtags     INFO    --------------------------------+---------------------
# BenderTools.GetDBtags     INFO                                Boole : v26r3
# BenderTools.GetDBtags     INFO                               Brunel : v43r2p7
# BenderTools.GetDBtags     INFO                                DDDB : ['dddb-20130929-1']
# BenderTools.GetDBtags     INFO                               Event : 133751
# BenderTools.GetDBtags     INFO                             EvtType : 13246002
# BenderTools.GetDBtags     INFO                               Gauss : v45r6
# BenderTools.GetDBtags     INFO                                 Run : 3349836
# BenderTools.GetDBtags     INFO                             SIMCOND : ['sim-20130522-1-vc-m
# BenderTools.GetDBtags     INFO                                 TCK : 0x00000000
# BenderTools.GetDBtags     INFO                                Time : 0 (Thu Jan  1 00:00:0
# BenderTools.GetDBtags     INFO                                UUID : 0CC46A49-2B70-E311-A8
# BenderTools.GetDBtags     INFO    ------------------------------------------------------------
# BenderTools.GetDBtags     INFO     Input files scanned: 1 from 1
# BenderTools.GetDBtags     INFO     Last (successful) file "/lhcb/MC/2012/ALLSTREAMS.DST/00
# BenderTools.GetDBtags     INFO    ------------------------------------------------------------
```

metainfo.txt hosted with ❤ by **GitHub** **view raw**

3. using `dirac-bookkeeping-decays-path` script from `LHCbDirac/prod` for the given MC eventype:

```
lb-run -c x86_64-slc6-gcc49-opt LHCbDirac/prod dirac-bookkeeping-decays-path 13104231
```

Challenge

Make a try with this command (do not forget to obtain valid Grid proxy).

- Is the output clear enough?

Solution

The output is a list of record. Each record consists of

```
1. The path in `bookkeeping-DB`
2. `DDDB`-tag
3. `SIMCOND`-tag
4. Number of files
5. Number of events
6. Unique production ID, that coudl be used to get more detailed information
```

```
[pclbitep01]~% lb-run -c x86_64-slc6-gcc49-opt LHCbDirac/prod dirac-bookkeeping-decays-path
('/MC/2012/Beam4000GeV-JulSep2012-MagUp-Nu2.5-EmNoCuts/Sim06b/Trig0x40990042Flagged/Reco14/S
('/MC/2012/Beam4000GeV-JulSep2012-MagDown-Nu2.5-EmNoCuts/Sim06b/Trig0x40990042Flagged/Reco14
('/MC/2011/Beam3500GeV-2011-MagDown-Nu2-Pythia6/Sim08a/Digi13/Trig0x40760037/Reco14a/Strippi
('/MC/2011/Beam3500GeV-2011-MagUp-Nu2-Pythia6/Sim08a/Digi13/Trig0x40760037/Reco14a/Stripping
('/MC/2011/Beam3500GeV-2011-MagDown-Nu2-Pythia8/Sim08a/Digi13/Trig0x40760037/Reco14a/Strippi
('/MC/2011/Beam3500GeV-2011-MagUp-Nu2-Pythia8/Sim08a/Digi13/Trig0x40760037/Reco14a/Stripping
('/MC/2012/Beam4000GeV-2012-MagUp-Nu2.5-Pythia8/Sim08a/Digi13/Trig0x409f0045/Reco14a/Strippi
('/MC/2012/Beam4000GeV-2012-MagDown-Nu2.5-Pythia8/Sim08a/Digi13/Trig0x409f0045/Reco14a/Strip
('/MC/2012/Beam4000GeV-2012-MagUp-Nu2.5-Pythia6/Sim08a/Digi13/Trig0x409f0045/Reco14a/Strippi
('/MC/2012/Beam4000GeV-2012-MagDown-Nu2.5-Pythia6/Sim08a/Digi13/Trig0x409f0045/Reco14a/Strip
('/MC/2016/Beam6500GeV-2016-MagUp-Nu1.6-25ns-Pythia8/Sim09b/Trig0x6138160F/Reco16/Turbo03/St
('/MC/2016/Beam6500GeV-2016-MagDown-Nu1.6-25ns-Pythia8/Sim09b/Trig0x6138160F/Reco16/Turbo03/
('/MC/2015/Beam6500GeV-2015-MagUp-Nu1.6-25ns-Pythia8/Sim09b/Trig0x411400a2/Reco15a/Turbo02/S
('/MC/2015/Beam6500GeV-2015-MagDown-Nu1.6-25ns-Pythia8/Sim09b/Trig0x411400a2/Reco15a/Turbo02
('/MC/2016/Beam6500GeV-2016-MagUp-Nu1.6-25ns-Pythia8/Sim09b/Trig0x6138160F/Reco16/Turbo03/St
('/MC/2016/Beam6500GeV-2016-MagDown-Nu1.6-25ns-Pythia8/Sim09b/Trig0x6138160F/Reco16/Turbo03/
('/MC/2015/Beam6500GeV-2015-MagUp-Nu1.6-25ns-Pythia8/Sim09b/Trig0x411400a2/Reco15a/Turbo02/S
('/MC/2015/Beam6500GeV-2015-MagDown-Nu1.6-25ns-Pythia8/Sim09b/Trig0x411400a2/Reco15a/Turbo02
[pclbitep01]~%
```

bookkepinginfo.txt hosted with ❤ by **GitHub** **view raw**

4. for Ganga/Grid there is a way to combine the function `getBKInfo2/getBKInfo` to obtain the information on flight from `bookkeeping-DB` and to propagate this information to Bender using `params` -argument of the `configure` function. This way is built around (3)

In details,...

```
template = JobTemplate(
   application  = prepareBender (
    version      = 'v31r0'                ,
    module       = my_module              ,
    use_tmp      = True                   ) ,
    ...
    )
productions = getBKInfo2 ( 13104231 )
for entry in productions :
    print 'INFORMATION: %s' % entry
    path      = entry ['path'     ] ## "long path"
    dddbtag   = entry ['DDDBtag'  ]
    conddbtag = entry ['CondDBtag']
    year      = entry ['Year']

    j = Job (  template )
    j.name      = ... ## construct name here
    j.inputdata = BKQuery ( path ).getDataset()
    j.application.params = { 'DDBtag' : dddbtag , 'CondDBtag' : conddbtag , 'Year' : year }
    j.submit()
```

where it is assumed that `configure` -function is instrumented properly to accept `params` and to propagate the tags further to `DaVinci` -configurable. The function `getBKInfo2` comes from here:

```
# ============================================================================
## get meta-information (paths and tags) from bookeeping
#  @param evttype     the event type
#  The function return the list/tuple of prodcution summaries
#  @code
#  productions = getBKInfo2 ( 13104231 )
#  for entry in productions :
#     print 'INFORMATION: %s' % entry
#     path      = entry ['path'     ] ## "long path"
#     dddbtag   = entry ['DDDBtag'  ]
#     conddbtag = entry ['CondDBtag']
#  @endcode
#
#  The obtained path in bookkeeping database can be used for subsequent
#  BKQuery command:
#  @code
#  data = BKQuery( path ).getDataset()
#  job.inputdata = data
#  @endcode
#
#  More complete list of cofiguration parameters for   DaVinci-bases applications
#  can be accessed via <code>DaVinciConf</code> dictionary
#  @code
#  conf = entry['DaVinciConf']
#  @endcode
#
#  The tags can be transferred to the application in application-dependent way
#
#  - For GaudiExec-application one can construct some additional options file:
```

```
30  #  @code
31  #  from tempfile import NamedTemporaryFile
32  #  tag_file = NamedTemporaryFile ( delete = False , suffix = '.py')
33  #  tag_file.write('''
34  #  from Configurables import DaVinci
35  #  dv = DaVinci ( Simulation = True ,
36  #                 DDDBtag    = '%s' ,
37  #                 CondDBtag  = '%s' )
38  #  ''' % ( dddbtag , conddbtab ) )
39  #  tag_file.close()
40  #  job.application.options.append ( tag_file.name )
41  #  @endcode
42  #  or alternatively, with more parameters:
43  #  @code
44  #  config   = entry['DaVinciConf']
45  #  from tempfile import NamedTemporaryFile
46  #  tag_file = NamedTemporaryFile ( delete = False , suffix = '.py')
47  #  tag_file.write('''
48  #  config = %s
49  #  from Configurables import DaVinci
50  #  dv = DaVinci ( **config )
51  #  ''')
52  #  tag_file.close()
53  #  job.application.options.append ( tag_file.name )
54  #  @endcode
55  #
56  #  - A helper function <code>daVinciMCConf</code> is provided for GaudiExec/DaVinci applicati
57  #  @code
58  #  config   = entry['DaVinciConf']
59  #  the_file = daVinciMCConf ( **config )
60  #  job.application.options.append ( the_file )
61  #  @endcode
62  #
63  #  - Alternatively  one can use rely on ``extaOpts'' of GaudiExec:
64  #  @code
65  #  options  = entry['DaVinciExtraOpts']
66  #  job.application.extraOpts = options
67  #  @endcode
68  #
69  #  - For Bender-based applications, no need to deal with separate files,
70  #  assuming <code>configure</code> method in the Bender module treats
71  #  appropriately the argument <code>params</code>:
72  #  @code
73  #  conf = entry['DaVinci']       ## get configuration
74  #  job.applciation.params = conf ## move configuration parameters to Bender
75  #  @endcode
```

```
def getBKInfo2 ( evttype )  :
    """Get meta-information (paths and tags) from bookeeping databbase
    - evttype : the event type
    The function returns the list/tuple of production summaries:

    >>> productions = getBKInfo2 ( 13104231 )
    >>> for entry in productions :
    ...   print 'INFORMATION: %s '  % entry
    ...   path      = entry ['path'     ]      ## 'long path'
    ...   prod_path = entry ['ProductionPath'] ## production path
    ...   dddbtag   = entry ['DDDBtag'  ]      ## DDDB-tag
    ....  conddbtag = entry ['CondDBtag']      ## SIMCONDDB-tag

    The obtained path in bookkeeping database can be used for subsequent BKQuery command:

    >>> data = BKQuery( path ).getDataset()

    It is a bit better top rely on ``ProductionPath'' :

    >>> data = BKQuery( entry['ProductionPath'] , Type = 'Production').getDataset()

    More complete list of cofiguration parameters for DaVinci-bases applicaitions
    can be accessed via DaVinciConf dictionary

    >>> conf = entry['DaVinciConf']
    >>> print conf.keys()

    The tags can be transferred to the application in application-dependent way:

    - For Bender-based applications, no need to deal with separate files,
    assuming that ``configure''-method in the Bender module treats
    appropriately the argument ``params''

    >>> conf = entry['DaVinciConf']  ## get configuration
    >>> job.application.params = conf ## move configuration parameters to Bender

    - For GaudiExec/DaVinci-application one can construct some additional options file:

    >>> from tempfile import NamedTemporaryFile
    >>> tag_file = NamedTemporaryFile ( delete = False , suffix = '.py')
    >>> tag_file.write('''
    ... from Configurables import DaVinci
    ... dv = DaVinci ( Simulation = True ,
    ...                DDDBtag    = '%s' ,
    ...                CondDBtag  = '%s' )
    ... ''' % ( dddbtag , conddbtab ) )
```

```
122       >>> tag_file.close()
123       >>> job.application.options.append ( tag_file.name )
124
125       or, alternatively, with more predefined  parameters:
126
127       >>> config   = entry['DaVinciConf']
128       >>> from tempfile import NamedTemporaryFile
129       >>> tag_file = NamedTemporaryFile ( delete = False , suffix = '.py')
130       >>> tag_file.write('''
131       ... config = %s
132       ... from Configurables import DaVinci
133       ... dv     = DaVinci ( **config )
134       ... ''')
135       >>> tag_file.close()
136       >>> job.application.options.append ( tag_file.name )
137
138       - A helper function ``daVinciMCConf'' is provided for GaudiExec/DaVinci application:
139
140       >>> config   = entry['DaVinciConf']
141       >>> the_file = daVinciMCConf ( **config )
142       >>> job.application.options.append ( the_file )
143
144       - Alternatively  one can use rely on ``extaOpts'' of GaudiExec:
145
146       >>> options  = entry['DaVinciExtraOpts']
147       >>> job.application.extraOpts = options
148
149       """
150       ##
151       import os
152       from subprocess import Popen, PIPE
153
154       try :
155           arguments = [ 'lb-run'                       ,
156                         'LHCbDirac/prod'               ,
157                         'dirac-bookkeeping-decays-path' ,
158                         str(evttype)
159                         ]
160           pipe = Popen ( arguments          ,
161                          env    = os.environ ,
162                          stdout = PIPE       )
163
164       except OSError :
165           # most likely dirac script is not in the PATH!
166           raise
167
```

```
    ## case-insensitive  dictionary
    #  @see https://stackoverflow.com/questions/2082152/case-insensitive-dictionary/32888599#
    class CIDict(dict):
        """case-insensitive  dictionary
        - see https://stackoverflow.com/questions/2082152/case-insensitive-dictionary/3288859
        """
        @classmethod
        def _k(cls, key):
            return key.lower() if isinstance(key, basestring) else key
        def __init__(self, *args, **kwargs):
            super(CIDict, self).__init__(*args, **kwargs)
            self._convert_keys()
        def __getitem__(self, key):
            return super(CIDict, self).__getitem__(self.__class__._k(key))
        def __setitem__(self, key, value):
            super(CIDict, self).__setitem__(self.__class__._k(key), value)
        def __delitem__(self, key):
            return super(CIDict, self).__delitem__(self.__class__._k(key))
        def __contains__(self, key):
            return super(CIDict, self).__contains__(self.__class__._k(key))
        def has_key(self, key):
            return super(CIDict, self).has_key(self.__class__._k(key))
        def pop(self, key, *args, **kwargs):
            return super(CIDict, self).pop(self.__class__._k(key), *args, **kwargs)
        def get(self, key, *args, **kwargs):
            return super(CIDict, self).get(self.__class__._k(key), *args, **kwargs)
        def setdefault(self, key, *args, **kwargs):
            return super(CIDict, self).setdefault(self.__class__._k(key), *args,
        def update(self, E={}, **F):
            super(CIDict, self).update(self.__class__(E))
            super(CIDict, self).update(self.__class__(**F))
        def _convert_keys(self):
            for k in list(self.keys()):
                v = super(CIDict, self).pop(k)
                self.__setitem__(k, v)

    result = []

    stdout = pipe.stdout
    for line in stdout :

        try :
            value = eval ( line )
        except :
            continue

```

```
214             if not isinstance ( value    , tuple ) : continue
215             if not isinstance ( value[0] , str   ) : continue
216             if not isinstance ( value[1] , str   ) : continue
217             if not isinstance ( value[2] , str   ) : continue
218             if not isinstance ( value[3] , int   ) : continue
219             if not isinstance ( value[4] , int   ) : continue
220             if not isinstance ( value[5] , int   ) : continue
221
222
223             path       = value [0]
224             production = value [5]
225
226             ## adjust buggy path for GAUSSHIST
227             ## if path.endswith('/GAUSSHIST') :
228             ##    nl = path.find('\n')
229             ##    if 0 <= nl : path = path[nl:]
230
231             ## skip GAUSSHIST
232             if path.endswith('/GAUSSHIST') : continue
233
234             ## create the entry:
235             entry = CIDict()
236
237             entry [ 'path'         ] = value [0]
238             entry [ 'DDDBtag'      ] = value [1]
239             entry [ 'CondDBtag'    ] = value [2]
240             entry [ 'NumFiles'     ] = value [3]
241             entry [ 'NumEvents'    ] = value [4]
242
243             entry [ 'EventType'    ] = evttype
244             entry [ 'ProductionID' ] = production
245             entry [ 'Production'   ] = production
246
247             if   'MagDown' in path : entry [ 'Magnet' ] = 'MagDown'
248             elif 'MagUp'   in path : entry [ 'Magnet' ] = 'MagUp'
249
250             spath =  path.split('/')
251
252             for s in spath[3:] :
253                 su = s.upper()
254                 if   su.startswith ( 'SIM'      ) : entry [ 'Simulation' ] = s
255                 elif su.startswith ( 'TRIG0X'   ) : entry [ 'TriggerTCK' ] = s[4:]
256                 elif su.startswith ( 'STRIPPING') : entry [ 'Stripping'  ] = s
257                 elif su.startswith ( 'RECO'     ) : entry [ 'Reco'       ] = s
258                 elif su.startswith ( 'TURBO'    ) : entry [ 'Turbo'      ] = s
259
```

```
260         entry ['FileType'  ] = spath[-1]
261
262         upath = path.upper()
263         if   upath.startswith ( '/MC/2011'    ) : entry [ 'DataType' ] = '2011'
264         elif upath.startswith ( '/MC/2012'    ) : entry [ 'DataType' ] = '2012'
265         elif upath.startswith ( '/MC/2015'    ) : entry [ 'DataType' ] = '2015'
266         elif upath.startswith ( '/MC/2016'    ) : entry [ 'DataType' ] = '2016'
267         elif upath.startswith ( '/MC/2017'    ) : entry [ 'DataType' ] = '2017'
268         elif upath.startswith ( '/MC/UPGRADE' ) : entry [ 'DataType' ] = 'Upgrade
269
270         if  entry.has_key('DataType') :
271             datatype = entry['DataType']
272             try :
273                 if str ( int ( datatype ) ) ==  datatype : entry['Year'] = datatype
274             except :
275                 pass
276
277         if   upath.endswith ( '.DST'  ) : entry ['InputType'] = 'DST'
278         elif upath.endswith ( '.LDST' ) : entry ['InputType'] = 'LDST'
279         elif upath.endswith ( '.XDST' ) : entry ['InputType'] = 'XDST'
280         elif upath.endswith ( '.MDST' ) : entry ['InputType'] = 'MDST'
281
282         prod_path = spath[-1]
283         if entry.has_key('Magnet'    ) : prod_path = '%s/%s'  % ( entry['Magnet'   ] , prod_p
284         if entry.has_key('DataType'  ) : prod_path = '%s/%s'  % ( entry['DataType
285         if entry.has_key('EventType' ) : prod_path = '%s/%s'  % ( entry['EventType
286         entry ['productionpath'] = '/%s/%s' % ( production , prod_path )
287
288         ##  prepare configuration for DaVinci
289         dv      = {}
290         dv_keys = ( 'DDDBtag' , 'CondDBtag' , 'InputType' , 'DataType' )
291         for k in dv_keys :
292             if entry.has_key( k ) : dv[k] = entry[k]
293
294         dv [ 'Simulation' ] = True
295         dv [ 'Lumi'       ] = False
296         if upath.endswith ( 'ALLSTREAMS.MDST' ) : dv [ 'RootInTES' ] = '/Event/AllStreams
297
298         entry  ['DaVinciConf' ] = dv
299
300         entry  ['DaVinciExtraOpts' ]  = '''
301         \nfrom Configurables import DaVinci
302         \nconfig = %s
303         \ndv     = DaVinci ( **config )
304         '''  % dv
305
```

```
306             result.append ( entry )
307
308         ## sorting: year, magnet, production
309         def  _kcmp ( o ) :
310             try :
311                 k = o['productionpath'][1:].split('/')
312                 return k[2],k[3],k[0]
313             except :
314                 return o['production']
315
316         ## sorting: year, magnet, production
317         result.sort ( key =  _kcmp )
318
319         return tuple(result)
320
321     # ============================================================================
322     ## Prepare the temporary file for DaVinci with useful configuration to process MC.
323     #  Typical usage in  Ganga script
324     #  @code
325     #  config   = { 'DDDBtag' : ... , 'CondDBTag' : ... }
326     #  the_file = daVinciMCconf ( **config )
327     #  job.application.options += [ the_file ]
328     #  @endcode
329     #  It is especially useful together with  <code>getBKInfo2</code>
330     #  @code
331     #  tjob        = JobTemplate ( ....     ) ## create the job template: application, outputfile
332     #  productions = getBKInfo2  ( 28144041 ) ## the event type
333     #  for entry in productions :
334     #     prod_path               = entry['productionpath'] ## productino path in bookkeeping da
335     #     job                     = Job  ( tjob )           ##  create the actual job from  the
336     #     job.inputdata           = BKQuery ( prod_path , Type = 'Production' ).getDataset() ##
337     #     davinci_conf            = entry['DaVinciConf']
338     #     conf_file               = daVinciMCConf ( **davinci_conf )
339     #     job.application.options += [ conf_file ]
340     #     job.comment = 'evttype %s, datatype %s, magnet %s ' % ( entry['EventType'] , entry['Dat
341     #     job.submit()  ## submit the job!
342     #  @endcode
343     def daVinciMCConf ( **config ) :
344         """Prepare the temporary file for DaVinci with useful configuration to process MC
345         Typical usage in Ganga script:
346         >>> config   = { 'DDDBtag' : ... , 'CondDBTag' : ... }
347         >>> the_file = daVinciMCconf ( **config )
348         >>> job.application.options += [ the_file ]
349
350         It is especially useful together with getBKInfo2:
351
```

```
352       tjob        = JobTemplate ( ....     ) ## create the job template: application, outputfil
353       productions = getBKInfo2  ( 28144041 ) ## the event type
354       for entry in productions :
355       ... prod_path               = entry['productionpath'] ## productino path in bookkeeping
356       ... job                     = Job  ( tjob )           ##  create the actual job from  th
357       ... job.inputdata           = BKQuery ( prod_path , Type = 'Production' ).getDataset() #
358       ... davinci_conf            = entry['DaVinciConf']
359       ... conf_file               = daVinciMCconf ( **davinci_conf )
360       ... job.application.options += [ conf_file ]
361       ... job.comment = 'evttype: {eventtype}, datatype: {datatype}, magnet: {magnet}, producti
362       ... job.submit()  ## submit the job!
363
364       """
365       from tempfile import NamedTemporaryFile
366       conf_file = NamedTemporaryFile ( delete = False , prefix = 'DaVinciConf_', suffix
367       options   = '''
368       \nfrom Configurables import DaVinci
369       \nconfig = %s
370       \ndv     = DaVinci ( **config )
371       '''  % config
372       conf_file.write( options )
373       conf_file.close()
374       return conf_file.name
375
376    # ============================================================================
377    # The END
378    # ============================================================================
```

getBKInfo2.py hosted with ❤ by **GitHub** **view raw**

Easy, safe and robust alternative :-)

In practice, none of the step described above are really needed, since one can just instruct Bender to obtain the tags directly from the input files. In this *recommended* scenario, no `DDDBtag/CondDBtags` to be specified for `DaVinci` -configurable, but one needs to activate `useDBtags=True` flag for `setData` -function:

```
dv = DaVinci ( Simulation      = True                    ,
               ...
               ## DDDBtag         = 'dddb-20130929-1'         , ## NOT NEEDED
               ## CondDBtag       = 'sim-20130522-1-vc-mu100' , ## NOT NEEDED
               ...
               TupleFile       = 'MCtruth.root' )
...
setData  ( inputdata , catalogs , castor , useDBtags = True ) ## <--- HERE!
```

This is, probably, the most robust, safe and simultaneously the most convinient way to treat `DDDB/SIMCOND` -tags for your application :-)

The price to pay: since internally it relies on the functionality provided by `get-dbtags` -script, for processing it could take addtional O(1-2)

minutes to open the first input file and to read `DDDB/SIMCOND` -tags from it.

Access MC-truth information in Bender

As it was mentioned above, the processing of simulation data in Bender is rather simple, one just needs to inherit the algorithm from base class `AlgoMC` , this class can be imported from the `Bender.MainMC` module.

```
from Bender.MainMC import *  # it imports also the whole content of Bender.Main module
class MyAlg(AlgoMC) : ...
```

And the corresponding wrapper for `Selection` -framework is `BenderMCSelection`

Get MC-truth data

To access to MC-truth data one uses the method `mcselect` , that is very similar to the method `select` discussed above

```
mcB = self.mcselect ( 'mcB' , '[Beauty ==>  J/psi(1S)  K+  K-]CC' )
mcK = self.mcselect ( 'mcK' , '[Beauty ==>  J/psi(1S) ^K+ ^K-]CC' ) ## note marked components
mcb = self.mcselect ( 'mcb' ,  BEAUTY ) ##  any beauty hadron
mc0 = self.mcselect ( 'mc0' , 'B0' ==  MCABSID ) ##  B0 or B~0
```

Again, as selection criteria one can use the *decay descriptors* or *LoKi MC-functors*.

```
dv   = DaVinci ( Simulation = True , ...  )
bsel = BenderMCSelection ( 'MCtruth' , [] )  ## <--- HERE!
daVinci.UserAlgorithms.append ( bsel )
```

Challenge

Try to code some MC-truth algorithm, that get some true MC-decays, and fill simple n-tuple/tree with some simple kinematical information on some of decay products. Try to select as an example the *ordinary* `ALLSTREAMS.DST` (not Turbo version!). Processing of `ALLSTREAMS.DST/Turbo` and various kinds of `ALLSTREAMS.MDST` and `MC-Turbo` requires a bit different configurtaion steos, that we'll discuss later. For a time being y ou can use e.g. the MC-file `'/lhcb/MC/2012/ALLSTREAMS.DST/00033494/0000/00033494_00000013_1.allstreams.dst'` , that contains the true MC-decays *Bs -> J/psi K+ K- pi+ pi-* with many intermediate resonances.

Solution

The complete module, that processes the events of `[B_s0 ==> J/psi(1S) K+ K- pi+ pi-]CC` with very rich structure of intermediate resonances is available here

Match *reco* with MC-truth

Matching of recontructed candidates with MC-truth is performed using `MCTRUTH` functors, that allows to answer on the *basic* question: *does this MC-particle makes the contribution to this reconstructed particle?*. Note that this question is different from e.g. *what is MC-truth for this reconstructed particle?*. These are *different* questions, and therefore one should not misinterpret the answers. For more details see the chapter 15 in LHCb-2004-023.

There are helper methods `mcTruth` , that are needed to create the functor `MCTRUTH`

```
mcK = self.mcselect ( 'mcK' , '[Beauty ==>  J/psi(1S) ^K+ ^K- pi+ pi-]CC' ) ## get true MC-kaons
trueK = MCTRUTH ( mcK , mc.mcTruth() ) ## <--- HERE: create MCTRUTH    functor
```

The created object `trueK` is *LoKi fuctor*, that evaluates to `True` for reconstructed particles, that get the contribution from true MC kaons, selected by the `mcselect` method, otherwise it evalautes to `False':

```
reco_kaons = self.select ( ... )
for k in reco_kaons :
    print ' True(MC-truth matched) kaon?  %s' % trueK ( k )
```

Since it is an ordinary *LoKi functor* is could be combined with all other functors, e.g. one can select from the input only MC-truth matcehd kaons:

```
truth_matched_kaons = self.select ( 'K' ,  ( 'K+' == ABSID ) & trueK )
```

Could it be inverted?

the short answer is *yes*, *there is inverse for* `MCTRUTH` *functor*. The inverse functor is `RCTRUTH` , it evalauted for `True` for any MC particle, that makes contribution to the selected recontructed candidate, and `False` otherwise.

```
B = self.select ( 'B' ,  'Beauty -> J/psi(1S)  K+  K- pi+ pi-' )
## get kaons from our reconstrcuted B-candidates
K = self.select ( 'K' , '[Beauty -> J/psi(1S) ^K+ ^K- pi+ pi-' )

recoK = RCTRUTH ( K , self.mcTruth() )

##  get all MC-particles that makes contribution to recontructed Kaon
mc = self.mcselect ( 'mc' , recoK )
print mc
```

Note that the list of found MC-particles here could be rather long (do you remember *the basic question*?)

Challenge

Try to code some MC-truth match algorithm, that get some MC-decays, some reconstructed decays, and perform MC-truth macth between them. Try to select as an example the *ordinary* `ALLSTREAMS.DST` (not Turbo version!). Processing of `ALLSTREAMS.DST/Turbo` and various kinds of `ALLSTREAMS.MDST` and `MC-Turbo` requires a bit different configurtaion steos, that we'll discuss later. For a time being y ou can use e.g. the MC-file `'/lhcb/MC/2012/ALLSTREAMS.DST/00033494/0000/00033494_00000013_1.allstreams.dst'` , that contains the true MC-decays *Bs -> J/psi*

K+ K- pi+ pi- with many intermediate resonances.

Solution

The complete module, that processes the events of `[B_s0 ==> J/psi(1S) K+ K- pi+ pi-]CC` with very rich structure of intermediate resonances is available here

How to process `HepMC` information

To access `HepMC`-information one can uses the method `gselect` that is very similar to the methods `select` and `mcselect` discussed above.

```
gBs = self.gselect ( 'bs' ,  "[ Beauty => ( D_s+ ==> K- K+ pi+ ) K-]CC ")
```

Again, as selection criteria one can use the *decay descriptors* or *LoKi HepMC-functors*.

`BenderScript`

`BenderScript` is a kind of *steroid-enhanced* and *doped* `GaudiPython` session. `BenderScript` shares all pros and contras with `GaudiPython` . It definitely provides very simple and efficient way for exploring the data, in particular very simple way for investigation of the content for the input data and TES. One can easily loop over events, data containers, make simple calcualtions, apply large part of LoKi functors, use some part of DaVinci machinery, etc. Howver there is large part of tasks that are very difficult in `BenderScrip` :

- large part of `DaVinci` *tools* is not working. It includes all *tools* and functionality related to e.g. actions with the associated best primary vertex. These manipulations are very complicated or some of them (e.g. PV-refit) is practically impossible to perform in a correct way without proper DaVinci context.
- Many *tools* do not work without the significant efforts. The results of *tools* often are very difficult to save to be used/reused e.g. for subsequent processing (e.g. save manually created particles and vertices to TES and to output file)
- Some very important and popular LoKi functors do not operate (mainly due to the issues from the previous item), e.g `BPV*` or `*DV` or `DTF_*` - those are not easy to use from the plain command line. There are some alternatives , but not for all functors.

See also a little bit more detailed TWiki-based tutorial on `BenderScript`

`bender` script

`BenderScript` session is started using the command `bender` . There are many command line options and keys for this script.

Challenge

Invoke this command using `-h` option. Is the displayed help clear enough?

Many of the options/keys can be deduced from the input data itself, in particular from the names of centrally produced files.

Input data

It is mandatory to supply `bender` session with input data. There are two ways to do it

1. supply data files as command like parameters

   ```
   lb-run Bender/prod bender /lhcb/LHCb/Collision12/PSIX.MDST/00035290/0000/00035290_00000221_1.psix.mdst
   ```

2. *import* the python file with data in a form of Gaudi Configurables, e.g. with `IOHelper().Input = [ ...]`

   ```
   lb-run Bender/prod bender -i data.py
   ```

   where `data.py` is

   ```
   from GaudiConf import IOHelper
   IOHelper().inputFiles([
   'root://eoslhcb.cern.ch//eos/lhcb/grid/prod/lhcb/LHCb/Collision12/PSIX.MDST/00035290/0000/00035290_00000221_1.psix.mdst',
   ], clear=True)
   ```

   Xml-catalogs can be supplied either via import from `data.py` or explicitely via `-x` -key

   ```
   lb-run Bender/prod bender -i data.py -x xml_catalog.xml
   ```

   Many of the options/keys (`DataType` , `Simulation` , `Turbo` , 'RootInTES', `InputType` , ...) can be deduced from the input data itself, in particular from the names of centrally produced files.

The basic action at command prompt

The incomplete list of the basic commands at the command prompts is :

- `ls`
- `get`
- `run`
- `skip`
- `rewind`

Do not forget that `bender` session is just a python session, therefore one always can invoke the commands `dir` and `help` .

Challenge

Start `bender` session and check the description for these method. Is it clear enough?

More advanced commands are:

- `seekForData`
- `seekStripDecision`
- `seekForODIN`
- `seekForEvtRun`
- `seekForAlgDecision`
- `seekForVoidDecision`

Challenge

Start `bender` session and check the description for these method. Is it clear enough?

The functions that help coding the actual scripts:

- `irun`
- `execute`

Histograms

Of course one can book & `ROOT` histograms at the command line

```
h1 = ROOT.TH1D('h1','histo',10,0,1000)
for i in irun ( ... ) :
     data = get ( .... )
     h1.Fill ( len (  data )
```

Of course at the end one needs to save this histogram into some ROOT-file

As viable and practical alternative one can rely here on `Gaudi` -histograms:

```
h1 = book ( 'some_path_here' , 'title' , 100 , 0 , 100 )
...
h1.fill( x )
```

In this case all created histograms will be automaticlaly saved into the outout file with historgams, that is defined via the command line

option `--histofile`, e.g. `bender ... --histofile DVhistos.root`

N-tuples/Trees

Again, oen can rely on bare `ROOT` (e.g. `PyROOT`) here to book/fill/save the tuples, but one can also use the functionality from `Gaudi`:

```
t = nTuple('QQQ','MyTuple')
data = ...
for b in data :
   t.column_float ( 'pt' , PT ( b ) )
   t.column_int   ( 'id' , int ( ID ( b ) ) )
   t.column_float ( 'mass' , M ( b ) )
   t.write()
```

In this case all created histograms will be automaticlaly saved into the outout file with historgams, that is defined via the command line option `--tuplefile`, e.g. `bender ... --tuplefile DVtuples.root`

Standard Bender scripts

There are several preconfigured `bender` scripts that helps to solve frequent problems. The list includes:

- `dst-dump` : the script that allows to *dump the content sumamry* for the certain (x,u,...)DST-file.
- `get-dbtags` : the script that helps to get `DDDB/CONDB` -tags from certain data file. It coudl be useful for processing of simulated samples
- `get-metainfo` : the script that help to a lot of available meta-information (including `DDDB/CONDB` -tags) from certain data file and corresponisng entry in `RunDB` . It could be useful for processing of simulated samples of other purposes
- `trg-check` : the script that helps to get the list of the relevant `L0TIS` , `L0TOS` , `Hlt1TIS` , `Hlt1TOS` , `Hlt2TIS` and `Hlt2TOS` trigger lines fo the given selection
- `no-mc-decays` : helper script that often helps to understand why not all MC-truth decays are selected using certain [decay descriptor]](https://twiki.cern.ch/twiki/bin/view/LHCb/FAQ/LoKiNewDecayFinders_. It allows to tune the decay descriptor properly to accept the missing entries.

All there scripts are properly documented and one can invoke them using `-h` option.

Bender & Ganga/GRID

`BenderModule` in `Ganga`

To submit the GRID job in `Ganga` with your module, there is an application `BenderModule` in Ganga, writted by Vladimir Romanovsky. The usage of this application is rather simple:

```
job = Job  ( ... )
job.application = BenderModule (
   module     =     'the_path/my_module.py' , ## <--- HERE
   directory  = ...   , ## the  directory where the existing project lives
   platform   = 'x86_64-slc6-gcc62-opt'
)
```

There us helper function `prepareBender` that allows to prepare the application

```
job = Job  ( ... )
job.application = prepareBender (
   version    = 'v31r0' ,
   platform   = 'x86_64-slc6-gcc62-opt' ,
   ## path       = '$HOME/cmtuser' ## use this  directory to prepare the project
   use_tmp    = True              ## use  some temporary directory
   params     = ...  ## optionally feed it with params arguments for configure method
   )
```

For more details consult `help(BenderModule)` and `help(prepareBender)` in `Ganga`

`BenderRun` in `Ganga`

`BenderRun` is a dedicated application in `Ganga` to run *bender script*. The usage is fairly trivial

```
j.application = BenderRun    ( scripts   = [ 'the_path/the_module.py'     ] ,
                               imports   = [ 'some_miport_file.py'        ] ,
                               commands  = [ 'ls()' , 'run(10)'  , 'ls()' ] ,
                               arguments = [ ... ] ,
                               directory =  ...  )
```

Again, there is helper function `prepareBenderRun` j.applictaion = prepareBenderRun (version = 'v30r1' , scripts = ['the_path/the_script.py' , 'another_script.py'] , commands = [...] , arguments = [...] , use_tmp = True , ...) `` `For more details consult` help(BenderRun) `and` help(prepareBenderRun) `in` Ganga`

Contributing

bender-tutorials is an open source project, and we welcome contributions of all kinds:

- New lessons;
- Fixes to existing material;
- Bug reports; and
- Reviews of proposed changes.

By contributing, you are agreeing that we may redistribute your work under these licenses. You also agree to abide by our contributor code of conduct.

Getting Started

1. We use the fork and pull model to manage changes. More information about forking a repository and making a Pull Request.
2. To build the lessons please install the dependencies.
3. For our lessons, you should branch from and submit pull requests against the `master` branch.
4. When editing lesson pages, you need only commit changes to the Markdown source files.
5. If you're looking for things to work on, please see the list of issues for this repository. Comments on issues and reviews of pull requests are equally welcome.

Dependencies

To build the lessons locally, install the following:

1. Gitbook

Install the Gitbook plugins:

```
$ gitbook install
```

Then (from the bender-tutorials directory) build the pages and start a web server to host them:

```
$ gitbook serve
```

You can see your local version by using a web-browser to navigate to `http://localhost:4000` or wherever it says it's serving the book.