Table of Contents

LHCb Bender Tutorial v12r1: Getting started with Bender

This hands-on tutorial is an introduction to Bender - Python-based user friendly environment for physics analysis. The purpose of these exercises is to allow you to write a complete though simple **analysis** algorithms for "typical" decay: $\mathrm{B}_\mathrm{s}^0 \to \mathrm{J}/\psi\phi$.

The exercises cover the following topics:

This tutorial has last been tested with Bender v12r1, the (partily obsolete) slides are available in pptx and pdf formats.

If the content of this pseudo-"hands-on" tutorial is a bit boring for you or you know well all the described topics please refer to these or even these pages in between the exercises.

Prerequisites

It is assumed that you are more or less familiar with the basic tutorial, also some level of familiarity with the DaVinci tutorial is assumed. Some familiarity with basic GaudiPython and python-based histograms & N-tuples is welcomed. It is also recommended to follow LoKi tutorial and the basic GaudiPython tutorial. You are also invited to the lhcb-bender mailing list.

Setup the environment for Bender

For this particular tutorial we'll concentrate on the interactive jobs and let the batch system and GRID, Ganga and DIRAC tool some rest. Batch and GRID-aware actions for Bender-based analysis are not covered by this tutorial.

The package **`Tutorial/BenderTutor`** is used as the placeholder for the tutorial exercises:

```
1000> SetupProject Bender vr12r1
1001> getpack Tutorial/BenderTutor v12r1
1002> cd Tutorial/BenderTutor/cmt
1003> cmt show uses | grep cmtuser
1004> make
1005> source ./setup.[c]sh
```

If everything is done in a correct way one can make an interactive inspection of the basic Bender objects:

```
1000> python
1001Python 2.4.2 (#1, Mar 24 2006, 16:38:17) [GCC 3.4.5 20051201 (Red Hat 3.4.5-2)] on linux2
1002Type "help", "copyright", "credits" or "license" for more information.
1003>>> dir()
1004['__builtins__', '__doc__', '__name__']
1005>>> from Bender.Main import *
1006Warning in <TEnvRec::ChangeValue>: duplicate entry <Library.ROOT@@Math@@CylindricalEta3D<dou
1007Warning in <TEnvRec::ChangeValue>: duplicate entry <Library.ROOT@@Math@@Cylindrical3D<double
1008Warning in <TEnvRec::ChangeValue>: duplicate entry <Library.ROOT@@Math@@Polar3D<double>=libM
1009Warning in <TEnvRec::ChangeValue>: duplicate entry <Library.ROOT@@Math@@Cartesian3D<double>=
1010ApplicationMgr    SUCCESS
1011=====================================================================================
1012                                                  Welcome to ApplicationMgr $Revision: 1.9
1013                                            running on pclbitep01 on Sat Sep 29 16:32:32 2007
1014=====================================================================================
1015ApplicationMgr       INFO Successfully loaded modules :
1016ApplicationMgr       INFO Application Manager Configured successfully
```

```
1017
1018Welcome Welcome Welcome Welcome Welcome Welcome Welcome Welcome Welcome Welcome Welcome Welc
1019LoKi
1020LoKi
1021LoKi                                          Welcome to LoKi!
1022LoKi
1023LoKi                                        (LOops & KInematics)
1024LoKi
1025LoKi                           Smart & Friendly C++ Physics Analysis Tool Kit
1026LoKi
1027LoKi
1028LoKi                           Author:  Vanya BELYAEV ibelyaev@physics.syr.edu
1029LoKi                       With the kind help of Galina Pakhlova & Sergey Barsuk
1030LoKi
1031LoKi                                          Have fun and enjoy!
1032LoKi
1033Welcome Welcome Welcome Welcome Welcome Welcome Welcome Welcome Welcome Welcome Welcome Welc
1034Welcome Welcome Welcome Welcome Welcome Welcome Welcome Welcome Welcome Welcome Welcome Welc
1035
1036Welcome Welcome Welcome Welcome Welcome Welcome Welcome Welcome Welcome Welcome Welcome Welc
1037Welcome Welcome Welcome Welcome Welcome Welcome Welcome Welcome Welcome Welcome Welcome Welc
1038Bender
1039Bender
1040Bender                                          Welcome to Bender!
1041Bender
1042Bender                              Python-based Physics Analysis Application
1043Bender
1044Bender                           Author:  Vanya BELYAEV ibelyaev@physics.syr.edu
1045Bender                       With the kind help of Pere Mato & Andrey Tsaregorodtsev
1046Bender
1047Bender                                       Have fun and enjoy! Good Luck!
1048Bender
1049Welcome Welcome Welcome Welcome Welcome Welcome Welcome Welcome Welcome Welcome Welcome Welc
1050Welcome Welcome Welcome Welcome Welcome Welcome Welcome Welcome Welcome Welcome Welcome Welc
1051>>> dir()
```

Try to compare the output of the first **`dir()`** command (the line 01003) and the output of the second **`dir()`** command. Using the command **`help()`** try to inspect in more detail some symbols:

```
1000>>> dir(M)
1001>>> help(M)
1002>>> dir(child)
1003>>> help(child)
1004>>> dir(children)
1005>>> help(children)
1006>>> help(Algo)
```

Learn how LoKi-functors work for Bender:

```
1000>>> p=LHCb.Particle()        ## create the "empty" particle
1001>>> P(p),M(p),E(p),PT(p),PX(p),PY(p),PZ(p)
1002>>> fun1=P+5000
1003>>> fun1
1004>>> fun1(p)
1005>>> fun1 += M*E
1006>>> fun1
1007>>> fun1 ( p )
1008>>> from LoKiCore.math import sin
1009>>> fun1 += sin(PZ)*M/E
1010>>> fun1
1011>>> fun1( p)
```

Read the configuration from some external **`*.opts`** file:

```
1000>>> gaudi.config( files = ['$DAVINCIROOT/options/DaVinci.opts'] )
```

Run some events:

```
1000>>> gaudi.run(10)
```

Leave Bender: **cntrl-D** for Linux, **cntrl-Z** for Windows.

Using this knowledge you already know how to run **any** DaVinci application as Bender script.

It is highly recommended (and hopefully will be enforced by high-level tools) that any of your Bender-based analysis job has the following structure:

```
1000from Bender.Main import *
1001
1002# put here the local ("importable") classes&functions
1003# it is a useful way to share your code with your friends&colleagues.
1004# in this way they can reuse (import) your lines..
1005
1006...
1007
1008# the specific job configuration:
1009def configure() :
1010    """ Job configuration """
1011    ...
1012    return SUCCESS
1013
1014# job steering:
1015if '__main__' == '__name__' :
1016
1017    configure()           ## configure the job
1018
1019    gaudi.run( 1000 )        ## run Gaudi
1020
```

The solutions:

As an example see the most trivial scripts **Minimalistic*.py** in the directory **$BENDERTUTOR/solutions/**.

The most trivial (empty) "Hello, World!" Bender-based algorithm

```
1000class HelloWorld( Algo ) :
1010    def analyse ( self ) :
1020        print "Hello, World"
1030        return SUCCESS
```

Also one can use Gaudi-printout:

```
1000class HelloWorld( Algo ) :
1010    """ This is a documentation string for the algorithm """
1020    def analyse ( self ) :
1030        """ This is a documentation string for the method """
1040        print "Hello, World"
1050        self.Print('Hello, World! (using Gaudi streams)')
1060        return SUCCESS
```

The algorithm

Essentially the algorithm has been presented few lines above. Please do not forget to decorate it a bit and add some documentation. Learn how the documentation appears in **`help()`**.

Please do not forget that one needs to "import" the base class **`Algo`**, the best (and recommended) way is:

```
1000form Bender.Main import *
```

Please also do not forget to add the documentation lines for the module itself (the first string in the module) and to fill properly the attribute **`__author__`**.

As the template one can use the following example:

```
1000#!/usr/bin/env python2.4
1010# ==============================================================================
1020"""
1030This a template file for the Bender-based script/module
1040"""
1050# ==============================================================================
1060## @file
1070#  This a template file for the Bender-based script/module
1080#  @author ...
1090#  @date   ...
1100# ==============================================================================
1110__author__ = " Do not forget your name here "
1120# ==============================================================================
1130
1140# ==============================================================================
1150## import all necessary stuff from Bender
1160from Bender.MainMC import *
1170# ==============================================================================
1180
1190# ==============================================================================
1200## @class Template
1210class Template(AlgoMC) :
1220    """
1230    This is the template algorithm
1240    """
1250
1260    ## standard constructor
1270    def __init__ ( self , name = 'Template' , **args ) :
1280        """
1290        Standard constructor
1300        """
1310        AlgoMC.__init__ ( self , name )
1320        for k in args : setattr ( self , k , args[k] )
1330
1340   ## standard method for analysis
1350    def analyse( self ) :
1360        """
1370        Standard method for analysis
1380        """
1390
1400        return SUCCESS
1410
1420# ==============================================================================
1430## job configuration:
1440def configure ( **args ) :
1450    """
1460    Configure the job
1470    """
1480
1490    ## configure DaVinci:
```

```
1500    from Configurables import DaVinci
1510    DaVinci (
1520        DataType   = 'DC06'     , # default
1530        Simulation = True       ,
1540        HltType    = '' )
1550
1560
1570    ## instantiate application Manager
1580
1590    gaudi = appMgr()
1600
1610
1620    ## create the algorithm
1630    alg = Template()
1640
1650    .... other actions ...
1660
1670    ## get the input data
1680    import data_Bs2Jpsiphi_mm as input
1690
1700
1710
1720    return SUCCESS
1730
1740# ================================================================================
1750## job steering
1760if __name__ == '__main__' :
1770
1780    ## make printout of the own documentations
1790    print __doc__
1800
1810    ## configure the job:
1820    configure()
1830
1840    ## run the job
1850    gaudi.run(1000)
1860
1870# ================================================================================
1880# The END
1890# ================================================================================
1900
```

The configuration:

For configuration we need to instantiate the algorithm and to define the list of top-level algorithms:

```
1000# instantiate the algorithm:
1010alg = MyAlg( "AlgorithmName" )
1020
1030#configure it, if needed
1040alg.OutputLevel = 3
1050
1060# set it as the only algorithm for Gaudi:
1070gaudi.setAlgorithms( [alg] )
```

Run it:

One can always run it using:

```
1000> python MyScript.py
```

Please note that if the first line of your script has the following form:

```
#!/usr/bin/env python
```

and the script is set to be "executable":

```
> chmod +x MyScript.py
```

In this case one can execute the script directly without explicit call for python executable:

```
> ./MyScript.py
```

There exists useful mode with **-i** key, in which after the execution of the script the system stays in "interactive" mode and provides use with the command prompt:

```
> python -i MyScript.py
....
>>> dir()
```

All interactive commands from the prompt will be saved into the file **.BenderHistory** in the current working directory.

The solution

The full solution to this exercise is available as **solutions/HelloWorld.py** in the **Tutorial/BenderTutor** package and could be inspected/copied in case of problems.

More about configuration

It is recommended to put all configuration stuff into function configure. The function configure contains two different part. Everything before the line **gaudi=apppMgr()** belongs to static configuration ("Configurables) and everything after belongs to dynamic configuration ("GaudiPython"). One should not mix these two parts..

Static Configuration using Configurables

For the most typical case static configurtaion is done using DaVinci Configurable:

```
## configure the job
def configure() :
    """
    Configure the job
    """

    from Configurables import DaVinci

    DaVinci (
        DataType   = 'DC06'     , # default
        Simulation = True       ,
        HltType    = '' )

     ....

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

Dynamic Configuration using GaudiPython

```
1000## configure the job
1010def configure() :
1020    """
1030    Configure the job
1040    """
1050
1060    ....
1070
1080    ## get/create application manager
1090    gaudi = appMgr()
1100
1110    myAlg = Template ( .... )
1120
1130
1140    gaudi.addAlgorithm ( myAlg )
1150
1160     ....
```

How to define the input data?

The input data can be specified in two different ways:

Using Configurables:

```
1000## configure the job
1010def configure() :
1020    """
1030    Configure the job
1040    """
1050
1060    from Configurables import DaVinci
1070
1080    DaVinci (
1090        DataType    = 'DC06'      , # default
1100        Simulation = True        ,
1110        HltType     = '' )
1120
1130     from Configurables import EventSelector
1140     EventSelector (
1150      Input = [
1160     ##
1170    "DATAFILE='PFN:castor:/castor/cern.ch/user/d/dijkstra/Stripped-L0xHlt1-DC06/Mbias-01.dst
1180    "DATAFILE='PFN:castor:/castor/cern.ch/user/d/dijkstra/Stripped-L0xHlt1-DC06/Mbias-02.dst
1190    "DATAFILE='PFN:castor:/castor/cern.ch/user/d/dijkstra/Stripped-L0xHlt1-DC06/Mbias-03.dst
1200    "DATAFILE='PFN:castor:/castor/cern.ch/user/d/dijkstra/Stripped-L0xHlt1-DC06/Mbias-04.dst
1210    "DATAFILE='PFN:castor:/castor/cern.ch/user/d/dijkstra/Stripped-L0xHlt1-DC06/Mbias-05.dst
1220     ])
1230
```

Using GaudiPython

```
1000## configure the job
1010def configure() :
1020    """
1030    Configure the job
1040    """
1050
1060     ...
1070     gaudi=appMgr()
1080
1090     evtSel = gaudi.evtSel()
1100
```

```
1110      evtSel.open ( 'MyFile.dst' )
1120
1130      ## or:
1140
1150      evtSel.open ( [ 'MyFile_1.dst' , 'MyFile_2.dst' , 'MyFile_3.dst' ] )
```

The access to the data from Bender-based algorithm

The access to the data in Bender is done through the member function of the ==Algo==-class:

```
1000mcparticles = self.get('MC/Particles')       ## get the container of Monte Carlo particles f
1010
```

All containers, registered in Gaudi Transient Stores are *iterable* objects in python and it is easy to make a loop over the content of the container

```
1000for mcp in mcparticles :
1010   print mcp, type(mcp)
1020   print 'Monte Carlo particle name = %s '%LoKi.Paricles.nameFromPID( mcp.particleID() )
1030   print 'dict:px=', mcp.momentum().px()
1040   print  'LoKi:px=' , MCPX( mcp )
```

Note that any python *object* (including the *type*) always could be inspected for all available functionality:

```
1000>>> o =
1010>>> dir(o)
1020>>> help(o)
1030>>> type(o)
1040>>> dir(type(o))
1050>>> help(type(o))
```

Also one always can inspect the corresponding DoxyGen documentation for the interesting C++ class or instance (the idea has been stolen from Thomas Ruf):

```
1000>>> import LoKiCore.doxygenurl as doxy
1010>>> o = ...
1020>>> doxy.browse ( o )      ## ask DoxyGen for the objects
1030>>> doxy.browse ( "LHCb::MCVertex")      ## ask doxygen for the class by name
1040
```

Important note: Bender "on-the-fly" adds some functionality to some event-model classes ("decoration") therefore for major interesting analysis classes one has more methods in python that in C++. In particular all types of *particles* (**`LHCb::Particle`**, **`LHCb::MCParticle`** & **`HepMC::GenParticle`**) gets the coherent interface for traversal of the decay tree:

```
1000>>> p = ...      ## get some  "particle": reconstructed or MC or HepMC
1010>>> c1 = p.child(1)    ## get the first daughter
1020>>> c2 = p.child(2)    ## get the second daughter
1030>>> c = p.children() ## get all daughters (as iterable vector)
1040>>> d = p.descendants() ## get all descendants (as iterable vector)
1050>>> a = p.ascendents() ## get all ascendants (as iterable vector)
1060>>> m=p.mother()    ## get a mother (if applicable)
1070>>> ps = p.parents() ## get parents (when applicable)
1080
```

All these functionality exists also in a function-like way:

```
1000>>> p = ...      ## get some  "particle": reconstructed or MC or HepMC
1010>>> for c  in children ( p ) : print c     ## loop over all children
```

```
>>> for ps in parents ( p ) : print ps  ## loop over parents
>>> for d in descendants ( p ) : print d  ## loop over descendants
```

Let's try to get some data from Gaudi Transient Event Store, e.g, container of Monte Carlo particles or vertices, make the loop over the content of the container and print some information about some particle or vertices (on your choice).

The algorithm

Note that for dealing with Monte Carlo truth one needs to import the functionality from the module **`Bender.MainMC`** (or, the same **`Bender.theMain`**, or **`Bender.All`** or **`Bender.Works`** or **`Bender.Awesome`**)

```
from Bender.MainMC import *

class MyAlg( AlgoMC) :

```

Please, do not forget the proper documentation strings! Essentially all needed fragments have been cited above.

The solution

The full solution to this exercise is available as **`solutions/HandsOn1.py`** in the **`Tutorial/BenderTutor`** package and could be inspected/copied in case of problems.

The simple selection of (Monte Carlo) particles

The second exercise demonstrates the simple selections of Monte Carlo particles using LoKi functors. The basic working horse for the simple selections is the member function **`(mc) select`**. It allows to select/filter from all input particles only the (MC)particles, which satisfy the certain criteria. For more information see LoKi User Guide/TWiki.

```
# select true Monte Carlo muons (positive and negative)
mcmu = self.mcselect ( "mcmuons" , "mu+" == MCABSID )     ## use the member function  mcsele

# select fast negative Monte Calro muons:
mcfast = self.mcselect ( "fastmu" , ("mu-" == MCID ) & ( MCPT > 10000 ) ) ## use the member

# select all charm particles
charm = self.mcselect ( "charm" , CHARM )

# select all beauty particles
beauty = self.mcselect ( "beauty" , BEAUTY )

# select all true muons form charm decay
mufromc = self.mcselect ( "mufromC" , ( "mu+" == MCABSID ) & FROMMCTREE ( charm ) )

# select all true muons from beauty but not from  charm
mufromb = self.mcselect ( "mufromB" , ( "mu+" == MCABSID ) & FROMMCTREE ( beauty ) & !FROMMC

```

The selected (MC)particles could be subjected by further sub-selections:

```
# sub-select fast negative muons:
mugood = self.mcselect ( "mugood" , mufromb ,  ( MC3Q < 0 ) & ( 1000 < MCPT ) )
```

Important Note: Due to technical restrictions the *"bit-wise" boolean* operators are overloaded for Bender instead of the *logical boolean* operators. It is a reason for some "extra" brackets.

The simple selection of Monte Carlo decay

Of course the masterpiece from Olivier Dormond - the tool **`MCDecayFinder`** is embedded into Bender (through LoKi class **`LoKi::MCFinder`**). It allows to select the particle from the certain Monte Carlo decays:

```
1000finder = self.mcFinder()     ## use the member-function "mcFinder"
1010
1020# get all kaons from the tree :
1030kaons  = finder.find( ' [B_s0 -> J/psi(1S)  ( phi(1020) -> ^K+ ^K- ) ]cc' )
```

Let's now write a simple algorithm which

- selects some Monte Carlo particles (either using **`mcselect`** or **`mcFinder`**) approach
  - ♦ or the combination of both!
- either makes a loop over the selected particles
- or fills the histogram
- or fills N-tuple

The solutions

The full solution to this exercise is available as **`solutions/HandsOn2.py`** and **`solutions/HandsOn3.py`** in the **`Tutorial/BenderTutor`** package and could be inspected/copied in case of problems.

Loops

The functionality of simple selection of Monte Carlo particles, studied for the previous exercise is well applicable also for the selection of reconstructed particles (of course with some modifications). The main modification is that one needs to use the function **`select`** instead of **`mcselect`** and use the special *functors*. E.g. the selection of good well-identified kaons:

```
1000# get all kaons with Delta log-Likelihood (K-pi) > -2 using functors ABSID and PIDK
1010allk  = self.select ( "allkaons" , ( "K+" == ABSID ) & ( PIDK > -2 ) )   ## use the member-f
1020
1030# sub-select positive kaons using the functor Q
1040kplus  = self.select ( "k+" , allk , Q > 0 )          ## use the member-function "select"
1050
1060# sub-select negative kaons  using the functor Q
1070kminus = self.select ( "k-" , allk , Q <0 )     ## use the member-function "select"
1080
```

The looping over the dikaon combinations is performed using the member-function **`loop`**:

```
1000# create the looping object:
1010phis = self.loop ( "k+ k-" , "phi(1020)" )
1020# make the loop over all dikaons:
1030for phi in phis :
1040        mass = phi.mass(1,2)  / 1000           # fast evaluation of the invariant mass (4-mome
1050        if mass > 1.100    : continue     # skip large masses
1060        # plot the mass
1070        self.plot ( mass  , "dikaon mass  (all) " , 1.0 , 1.1 )
1080        # ask for chi2 of the vertex fit:
1090        chi2 = VCHI2 ( phi )      ## the compound particle (and its vertex!)  is created "on-d
1100        if 0 > chi2 or chi2 > 100 : continue    # skip fit failures and large chi2
1110        self.plot ( mass   , "dikaon mass  (with good vertex) " , 1.0 , 1.1 )
```

```
1120        self.plot ( M(phi) / 1000 , "dikaon mass  (with good vertex) using functor M" , 1.0 ,
1130
```

As soon as one gets the composed particle, one can apply all LoKi functors, e.g.

```
1000# get the absolute value of the mass difference with respect to the nominal mass
1010adm = ADMASS( "phi(1020" ) < 20
1020...
1030# make the loop over all dikaons:
1040for phi in phis :
1050    ...
1060    if PT ( phi ) < 500 : continue    ## use the functor
1070    if not adm ( phi ) : continue        ## use the functor/predicate!
1080    ...
```

If the composed particle is selected as the interesting one, one can "save" is for subsequent analysis:

```
1000# make the loop over all dikaons:
1010for phi in phis :
1020    ...
1030    if  ... : continue    ## use the functor
1040    if  ... : continue     ## use the functor/predicate!
1050
1060    phi.save ( "myGoodPhi" )       ## MIND THE NAME
1070
```

All saved combination could be extracted (by name!) and analysed:

```
1000# get all previosuly saved phis:
1010phis = selef.selected ( "muGoodPhi" )
1020
1030if phis.empty() :
1040    self.Warning("No phi-candidates are sleelcted")
1050
1060## count them
1070nPhi = self.counter ("#phi")
1080nPhi += phis.size()
```

If one have saved previously $\mathrm{J}/\psi \to \mu^+\mu^-$ candidates in analogous way under the name **"myGoodPsi"**, one can perform the next selection step and try to recontruct the $\mathrm{B}^0_\mathrm{s} \to \mathrm{J}/\psi\phi$ candidates:

```
1000# construct the looping objects
1010bs = self.loop ( "myGoodPsi myGoodPhi" , "B_s0")    ## MIDN THE NAMES
1020# amke the loop over all combinations:
1030for Bs in bs:
1040    ... apply cuts
1050    self.plot ( M( Bs ) / 1000 , "mass of Bs-candidate " , 4.0 , 6.0 )    ## make plots
1060    Bs.save ("Bs")    ## save good Bs-candidates
1070
1080
1090bs = self.selected ("Bs")
1100
1110# make an algorithm selection decision:
1120self.setFilterPassed ( not bs.empty() )
1130
```

The algorithm

Let's try to write the simple algorithm which:

1. selects $\phi \to \mathrm{K}^+\mathrm{K}^-$ candidates
2. selects $\mathrm{J}/\psi \to \mu^+\mu^-$ candidates

3. combines the selected ϕ and J/ψ candidated to search $\mathrm{B}^0_\mathrm{s} \to \mathrm{J}/\psi\phi$ candidates
4. makes certain plots
    - if you are familar enough with N-tuples, fill N-tuple with simple variables

Your algorithm will include essentially five parts:

1. selection of good kaons for recontruction of ϕ candidates
2. the loop over dikaons and the selection of ϕ candidates
3. selection of good muons for recontruction of J/ψ candidates
4. the loop over dimuons and the selection of J/ψ candidates
5. the loop over $\phi\mathrm{J}/\psi$ and selection of B^0_s candidates

The functions, which could be of the interest for you:

`P`
: momentum of the particle

`ID`
: numerical ID of the particle

`KEY`
: the key of the particle

`PT`
: transverse momentum

`PX` , **`PY`** , **`PZ`**
: x-,y-,z-components of the particle mometum

`PIDK`
: $\Delta_{LL}\left(\mathrm{K}-\pi\right)$

`PIDmu`
: $\Delta_{LL}\left(\mu-\pi\right)$

`M`
: The invariant mass of the particle, **`LHCb::Particle::momentum().M()`**, $\sqrt{E^2-\vec{p}^2}$

`M12`
: The invariant mass of the first and second daughter particles

`CHILD`
: Meta-function, which delegates the evaluation of another function to daughter particle, e.g. **`CHILD( P , 1 )`** evaluates the momentum of the first daughter particle

`DMASS`
: The function is able to evaluate the invariant mass difference with respect to some reference mass: e.g. **`DMASS("phi(1020")`** evalutes the difference between the invarinat mass of the particle and the nominal mass of ϕ.

`ADMASS`
: The function evaluates the absolute value of the invariant mass diffrence with respect to some reference mass: e.g. **`ADMASS("phi(1020")`** evalutes the absolute value of the difference between the invarinat mass of the particle and the nominal mass of ϕ.

Also note that the name for J/ψ particle is **`"J/psi(1S)"`** and the name for B^0_s is **`"B_s0"`**.

The solutions

The full solution to this exercise is available as **`solutions/RCSelect.py`** in the **`Tutorial/BenderTutor`** package and could be inspected/copied in case of problems.

Easy matching to Monte Carlo truth

The next exercise illustrates the usage of Monte Carlo information in Bender.

Bender (through LoKi) offer nice possibility to perform easy "on-the-fly" access to Monte Calro truth information. Not all possible cases are covered on the equal basis but the most frequent idioms are reflected and well-covered in Bender and LoKi

Match to Monte Carlo truth

There are variety of the methods in Bender for Monte Carlo truth matching. Here we describe the most trivial (which covers well the most frequent case) one, the function **MCTRUTH**. This function being constructed with the "list" of Monte Carlo particles, is evaluated to **true** for reconstructed particles, which are matched with one of the Monte Carlo particle (or one of its Monte Carlo daughter particle) used for construction of the function. e.g. :

```
# retrieve Monte Carlo matching object:
mc = self.mcTruth()                              ## the member function of class AlgoMC

# get some MC-particles, (e.g.mc-muons)
mcmu  = ....

# create the function (predicate) using the list of true muons
fromMC = MCTRUTH ( mc , mcmu ) ## this function is evaluated to "true" for particles, matche

# select recontructed muons, matched with true MC-muons:
mu = self.select ( "goodmu" , ( "mu+" == ABSID ) & fromMC ) ##  use the function/predicate
--++++ Using Configurables:

%SYNTAX{ syntax="python" numbered="1000" numstep="10"}%
## configure the job
def configure() :
    """
    Configure the job
    """

    from Configurables import DaVinci

    DaVinci (
        DataType    = 'DC06'     , # default
        Simulation = True       ,
        HltType     = '' )

     from Configurables import EventSelector
     EventSelector (
      Input = [
     ##
    "DATAFILE='PFN:castor:/castor/cern.ch/user/d/dijkstra/Stripped-L0xHlt1-DC06/Mbias-01.dst
    "DATAFILE='PFN:castor:/castor/cern.ch/user/d/dijkstra/Stripped-L0xHlt1-DC06/Mbias-02.dst
    "DATAFILE='PFN:castor:/castor/cern.ch/user/d/dijkstra/Stripped-L0xHlt1-DC06/Mbias-03.dst
    "DATAFILE='PFN:castor:/castor/cern.ch/user/d/dijkstra/Stripped-L0xHlt1-DC06/Mbias-04.dst
    "DATAFILE='PFN:castor:/castor/cern.ch/user/d/dijkstra/Stripped-L0xHlt1-DC06/Mbias-05.dst
     ])
```

particle = ...

if fromMC (particle) : ... it is a particle matched with true MC-muons ...

%ENDSYNTAX% **Important note**: the function **MCTRUTH** evaluates to **true** also for recontructed particles, which are matched to Monte Carlo particles form decay trees of the original Monte Carlo particles.

The function **MCTRUTH** described above is very useful for selection of "True"-decays and combinations:

```
## get all Monte Carlo psis, which decay into mu+ mu-:
mcPsi  = ...                    ## you already know how to get them, see the previous exercis
```

```
1020
1030## get all Monte Carlo muons from the  decay  psi -> mu+ mu-:
1040mcmu  = ...                    ## you know well  how to get them, see the previous exercises
1050
1060## create the function/predicate  for "true" psi
1070truePsi = MCTRUTH ( mc , mcPsi ) ## check the matching with Monte Carlo true psi
1080
1090## create the function/predicate  for "true" muon from psi
1100trueMu = MCTRUTH ( mc , mcmu ) ## check the matching with Monte Carlo true muon from psi
1110
1120## get the reocnstructed muons:
1130muplus = self.select ( "mu+" , .... )           ## you know well how to get them!
1140muminus = self.select ( "mu-" , .... )         ## you know well how to get them!
1150
1160## make the loop
1170Psi = self.loop ( "mu+ mu-" , "J/psi(1S)" )
1180for psi in Psi :
1190   # fast evalaution of mass
1200   m12 = psi.mass(1,2)
1210   if m12 < 2000 || m12 > 4000 : continue               ## skip bad combinations
1220
1230   self.plot ( m12 , "mass of all dimuons " , 2 , 4 )
1240
1250   mu1 = psi ( 1 ) ## access to the first daughter particle of the loop
1260   mu2 = psi ( 2 ) ## get the second daughter particle of the loop
1270
1280   if trueMu ( mu1 ) or trueMu ( mu2 ) :       ## use the matching predicates
1290      self.plot ( m12 , "mass of all dimuons, at least one is true  " , 2 , 4 )  ##  at leas
1300
1310   if trueMu ( mu1 ) and trueMu ( mu2 ) :   ## use the matching predicates
1320      self.plot ( m12 , "mass of all dimuons, both muons are true  " , 2 , 4 ) ##  both muon
1330
1340   if truePsi ( psi ):    ## use the matching predicates
1350      self.plot ( m12 , "mass of all dimuons, true J/psi " , 2 , 4 ) ; // the dimuon combina
```

Now you know all the major ingredients useful for simple Monte Carlo match. Let's try to write the algorithm for $\phi \to \mathrm{K}^+\mathrm{K}^-$ selection using your experience from the previous exercise:

1. find true Monte Carlo decays $\phi \to \mathrm{K}^+\mathrm{K}^-$
2. find true Monte Carlo kaons from the decay $\phi \to \mathrm{K}^+\mathrm{K}^-$
3. create the helper matching predicates/functions for Monte match of ϕ and $\mathrm{K}^\pm$
4. select the good positive kaons
5. select the good negative kaons
6. make a loop over dikaons
7. apply some cuts for the compound particle
8. plot some distributions for the compound and/or daughter particles **with and without matching to Monte Carlo truth**
    - ♦ if you are already familar with N-tuples fill N-tuple with all these variables
9. save "interesting" candidates and count them

Note: for access to Monte Carlo truth one needs to inherit the algorithm from **`AlgoMC`** istead of **`Algo`**.

Also since we are working only with charged particles, one can gain some CPU performace by disabling the Monte Carlo truth for calorimeter objects and neutral protoparticles, which are enabled in the default configuration. It could be done using the following section for the property **`PP2MCs`** in your algorithm:

```
1000# use Monte Carlo truth only for charged tracks:
1010alg =
1020alg .PP2MCs = [ "Relations/Rec/ProtoP/Charged" ]
```

The solution

The full **solution** to this exercise is available as **`solutions/HandsOn4.py`** in the **`Tutorial/BenderTutor`** package and could be inspected/copied in case of problems.

Gluing everything together

Finally we have all ingredients for analysis algorithms:

- one knows how to select/filter the particles
- one knows how to combine/save the composed particles
- one knows hot to access to Monte Carlo truth

If in addition one is familar with histograms and N-tuples, one is ready to start the physical analysis in LHCb using Python/Bender. As a final example, try to merge two previous exercises and to write the analsysis/selection algorithm, similar to the exercise 4 but equipped with Monte Carlo truth flags in the spirit of exercise 5.

The solution

The full **solution** to this exercise is available as **`solutions/RCMCSelect.py`** in the **`Tutorial/BenderTutor`** package and could be inspected/copied in case of problems.

How to use (Py)ROOT with Bender for the same session ?

To enable ROOT visualizaton, somewhen close to the first line of script of need to import it excplicitly:

For configuration we need to instantiate the algorithm and to define the list of top-level algorithms:

```
1000import ROOT
```

After, all ROOT classes are available for manipulations, e.g.:

```
1000b = ROOT.TBrowser()
1010
1020f = ROOT.TFile('my_root_file.root' )
```

E.g. one can make following (interactive) session:

```
1000ln -s ../solution/RCSelect.py  RCSelect.py
1010
1020python
```

And inside the python session:

```
1000>>> import ROOT
1010>>> c = ROOT.TCanvas( .... )
1020
1030>>> import RCSelect
1040>>> RCSelect.configure ()          ## configure
1050>>> RCSelect.run(100)              ## run 100 events
1060
1070>>> gaudi=RCSelect.appMgr()    ## get applictaion manager
1080>>> hsvc = gaudi.histSvc()         ## get histogram service
1090>>> hsvc.dump ()                        ## get the list of all booked histograms
```

```
1100
1110>>> h1 = hsvc['/stat/RCSelect/dimuon invariant mass' ]      ## get the histogram by name
1120>>> h1.plot()                     ##  visualize it!
1130
1140>>> RCSelect.run(500)      ## run more events
1150>>> h1.plot()
```

-- Vanya Belyaev - 29 Sep 2007

-- VanyaBelyaev - 31-Aug-2010

This topic: LHCb > BenderTutorialV12r1
Topic revision: r1 - 2010-08-31 - VanyaBelyaev