Table of Contents

LHCb LoKi Tutorial: Getting started with LoKi

This hands-on tutorial is an introduction to LoKi - C++ toolkit for easy and friendly 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 DaVinci v24r0, the slides available as pdf or pptx, also the (slightly obsolete) slides are available through:

- Tutorial agenda page
- LoKi-pages

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 supposed. It is also recommended to look through histograms & N-tuples pages. You are also invited to the lhcb-loki mailing list.

Setup the environment for DaVinci

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 LoKi-bases analysis are identical to the actions needed for native DaVinci and thus are not covered by this tutorial. For more details on GRID&batch see DaVinci tutorial.

The package **`Tutorial/LoKiTutor`** is used as the placeholder for the tutorial exercises. Please take care that you have installed and working version of DaVinci with local version of **`Tutorial/LoKiTutor`** package. E.g. for the clean environment one can perform the following actions:

```
1000> SetupProject DaVinci v24r0 --build-env
1001> getpack Tutorial/LoKiTutor v9r0
```

To build the tutorial one needs to use the standard procedure, described in much detail for the basic tutorial:

```
1000> cd Tutorial/LoKiTutor/cmt
1001> make
1002> SetupProject DaVinci v24r0
```

If everything is done in a correct way one can now run simple DaVinci job:

```
1000> gaudirun.py $DAVINCIROOT/options/DaVinci.py
```

In a similar way one can run the most trivial (empty, "Hello, World!") LoKi-algorithm:

```
1000> cd $LOKITUTORROOT/cmt
1001> cp ../solution/HelloWorld/HelloWorld.cpp ../src
1002> make
1003> gaudirun.py $LOKITUTORROOT/solutions/HelloWorld/HelloWorld.py
```

The most trivial (empty) "Hello,World!" LoKi algorithm

The most trivial LoKi empty algorithm (**"Hello,World!"**) demonstrates in a very clear way the main principles of any LoKi-based algorithms:

- It resides in a single ***.cpp** file, no header (***.h**) is required
- To get access to LoKi classes one needs to include the file **LoKi/LoKi.h** which imports around 90-95% of functionality, available in LoKi
- The body of the algorithm is defined using **LOKI_ALGORITHM** or **LOKI_MCALGORITHM** macros
  - The macro **LOKI_ALGORITHM** should be used for Monte Carlo-independent studies

The algorithm

The simplest algorithm looks like:

```
1000// Include files
1010// ============================================================================
1020#include "LoKi/LoKi.h"                       // from the package Phys/LoKi
1030// ============================================================================
1040LOKI_ALGORITHM(HelloWorld)       // algorithm C++ type as the parameter of macro
1050{
1060
1070  always() << " Hello, World! " << endreq ;
1080
1090  return StatusCode::SUCCESS ;                                    // RETURN
1100
1110}
```

Please, do not forget to add *some* Doxygen comments. It is a good style even for your tutorial algorithm:

```
1000/** @file
1010  *  it is a simple tutorial algorithm for ...
1020  *  @author  ...
1030  *  @date ...
1040  */
1050
1060// and/or:
1070
1080/** @class AAA
1090  *  it is a simple tutorial algorithm for ...
1100  *  @author  ...
1110  *  @date ...
1120  */
```

Put your algorithm into **src** ditrectory of **Tutorial/LoKiTutor** package and build the package.

The configuration

Of course the configuration of this simple ("do-nothing") algorithm does not require a lot of efforts:

```
1000#!/usr/bin/env gaudirun.py
1010# ============================================================================
1020## @file
1030#  The configuration fiel to run the simplest "Hello World!" exersize for
1040#  LoKi-tutorial
1050#  @author Vanya  BELYAEV Ivan.Belyaev@nikhef.nl
1060#  @date 2008-10-07
1070# ============================================================================
1080"""
1090The simple configuration file to run the simplest 'Hello World!'
```

```
1100exersize for LoKi-tutorial
1110"""
1120__author__   = " Vanya  BELYAEV Ivan.Belyaev@nikhef.nl "
1130__version__  = " CVS Tag $Name:  $, version $Revision: 1.14 $  "
1140# ============================================================================
1150## The general configuration stuff
1160# ============================================================================
1170from  Gaudi.Configuration import *
1180# ============================================================================
1190## The general configuration stuff form DaVinci
1200from Configurables import DaVinci
1210# ============================================================================
1220## pick-up the 'configurable' for our specific algorithm
1230# ============================================================================
1240from  Configurables import LoKi__MyHelloWorld as HelloWorld
1250
1260## create the configurable specific algorithm:
1270hello = HelloWorld ('Hello')
1280
1290DaVinci  (
1300    DataType       = 'DC06'    ,  ## data type
1310    UserAlgorithms = [ hello ] ,  ## the list of algorithms
1320    EvtMax         = 100          ## number of events
1330    )
1340
1350## input data:
1360from LoKiExample.Bs2Jpsiphi_mm_data import Inputs
1370EventSelector (
1380    Input     = Inputs , ## input data
1390    PrintFreq = 1
1400    )
1410
1420# ============================================================================
1430# The END
1440# ============================================================================
1450
```

Clearly only few lines are important here:

```
1000from  Gaudi.Configuration import *
1010from Configurables import DaVinci
1020from  Configurables import LoKi__MyHelloWorld as HelloWorld
1030
1040hello = HelloWorld ('Hello')
1050
1060DaVinci  (
1070    DataType       = 'DC06'    ,  ## data type
1080    UserAlgorithms = [ hello ] ,  ## the list of algorithms
1090    EvtMax         = 100          ## number of events
1100    )
1110
1120## input data:
1130from LoKiExample.Bs2Jpsiphi_mm_data import Inputs
1140EventSelector (
1150    Input     = Inputs , ## input data
1160    PrintFreq = 1
1170    )
```

The rest is just some kind of documentation, decorations and comments.

As the input data for this particular exercise one can use **any** arbitrary input data, e.g.

```
1000# use another data:
1010
1020EventSelector (
```

```
    Input     = [ 'PFN:file1.dst' , 'PFN:file2.dst'] , ## input data
    PrintFreq = 1
    )
```

Also one can follow the instructions from DaVinci tutorial and find some appropriate data using LHCb bookkeeping data base.

The solution

The full **solution** to this exercise (`*.cpp` & `*.py` files) is available in the directory `solutions/HelloWorld/` in the `Tutorial/LoKiTutor` package and could be inspected/copied in case of problems.

Simple selection of particles

The second exercise demonstrates the simple selections of particles using LoKi *functors*.

The basic working horse for the simple selections is the member function `select`. It allows to select/filter from all input particles only those satisfying the certain criteria. For more information see LoKi User Guide/TWiki.

```
// get all positive muons:
Range muplus = select ( "mymu+" , "mu+" == ID ) ; /// "select" is a member function of class

// get all negative muons:
Range muminus = select ( "mumu-" , "mu-" == ID ) ;

// get all (positive&negative) muons:
Range muall  = select ("mymuons" , "mu+" == ABSID ) ;

// get all FAST muons:
Range mufast = select ( "fastmuons" , "mu+" == ABSID && P > 10 * Gaudi::Units::GeV  && 500 <

// get positive muons from all fast muons (subset)
Range mu1 = select ( "fastmuons+" , mufast , Q > 0 ) ;
```

In this example `ID`, `P`, `PT` and `Q` are the simplest LoKi *functors*, which evaluate the particle id, momentum, transverse momentum and the charge correspondingly. Almost exhaustive list of currently available *particle functors* is available here.

The return value of function `select` represents a lightweight (named) container which contains the selected particles and can be directly inspected:

```
Range particles = ... ;

// print the size of the conatiner
always() << " Number of selected particles: " << particles.size() << endreq ;

// make the explicit loop over the container:
for ( Range::iterator ip = particles.begin() ; particles.end() != ip ; ++ip )
{
    const LHCb::Particle* p = *ip ;
    // fill the histogram:
    plot ( PT ( p ) / Gaudi::Units::GeV , "pt of selected particles [in GeV]" , 0 , 5 ) ;
}

// fill the historgam through implicit loop:
plot ( PT  / Gaudi::Units::GeV ,  particles.begin() , particles.end()    , " pt of selected
```

If you are not familar with easy and friendly histogramming in Gaudi, see here.

Let's write the simple algorithm which selects, counts and fills some histos for the particles, which satisfy certain criteria. E.g. good or fast or well identified muos or kaons.

Start new algorithm in **`src`** directory of **`Tutorial/LoKiTutor`** package:

```
1000#include "LoKi/LoKi.h"
1010
1020LOKI_ALGORITHM(GetData)
1030{
1040    // avoid very long names:
1050    using namespace LoKi ;
1060    using namespace LoKi::Types ;
1070    using namespace LoKi::Cuts   ;
1080
1090    ... put your lines here ...
1100
1110    return StatusCode::SUCCESS ;                                  // RETURN
1120}
```

Put your selection lines after the line 01080. E.g. try to select fast&well-identified muons. Consult these pages for the list of functions. Probably the following functions could be useful:

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

Useful quantity for separation of particles from decays of beautyparticles and the particles, originating in the primary vertices, are the minimal χ^2 of impact parameter, calculated over all reconstructed primary vertices. This quantity in LoKi is evaluated using the function **`MIPCHI2`** :

```
1000// get all reconstructed primary vertices:
1010VRange pvs = vselect ('pv' , ISPRIMARY ) ;
1020
1030// create the function from the list of primary vertices and some helper object (essentially
1040Fun fun = MIPCHI2 ( pvs , geo() ) ;   /// "geo" is a member function of class LoKi::Algo
1050
1060// use it explicitly:
1070const LHCb::Particle* p = ... ;
1080const double minChi2 = fun ( p ) ;
1090
1100 // use it for selection:
1110Range pionsNotFromPV = select ("pions" , "pi+" == ABSID && 9 < fun ) ;
```

Try to select, count particles and make the simple plots..

The configuration

Of course the configuration of this algorithm require a bit more typing, in particular one needs to specify the input locations for the algorithm:

```
#!/usr/bin/env gaudirun.py
# ==============================================================================
## @file
#  The configuration file for 'GetData'-solution for LoKi-tutorial
#  @author Vanya BELYAEV Ivan.Belyaev@nikhef.nl
#  @date 2008-10-07
# ==============================================================================
"""
The configuration file for 'GetData'-solution for LoKi-tutorial
"""
# ==============================================================================
__author__  = " Vanya BELYAEV Ivan.Belyaev@nikhef.nl "
__version__ = " CVS Tag $Name:  $, version $Revision: 1.14 $ "
# ==============================================================================
## The general configuration stuff
# ==============================================================================
from  Gaudi.Configuration import *
# ==============================================================================
## The general configuration stuff from DaVinci
from Configurables import DaVinci
# ==============================================================================
## pick-up the 'configurable' for our specific algorithm
# ==============================================================================
from  Configurables import LoKi__GetData as GetData

## create the configurable specific algorithm:
data = GetData (
    'GetData' ,             ## Algorithm instamce name
    ## Input particles
    InputLocations = [ 'StdLoosePions' , 'StdTightKaons' , 'StdLooseMuons' ]
    )

DaVinci  (
    DataType       = 'DC06'    ,   ## data type
    UserAlgorithms = [ data ]  ,   ## the list of algorithms
    EvtMax         = 100           ## number of events
    )

## input data:
from LoKiExample.Bs2Jpsiphi_mm_data import Inputs
EventSelector (
    Input     = Inputs , ## input data
    PrintFreq = 10
    )

# ==============================================================================
# The END
# ==============================================================================

```

The solution

The full **solution** to this exercise (**`*.cpp`** & **`*.py`** files) is available in the directory **`solutions/GetData/`** in the **`Tutorial/LoKiTutor`** package and could be inspected/copied in case of problems.

The loops over multiparticle combinations

The next example illustrates the looping over the multiparticle combinations.

Let's assume we want to loop over all $\mathrm{K}^+\mathrm{K}^-$ combinations and plot the invariant mass of the dikaons. It could be done easily by the explicit double loop in the spirit of native DaVinci:

```
Range kplus = select ( "k+" , ... ) ; // you already know how to select "good" positive kaon
Range kminus = select ( "k-" , ... ) ; // you already know how to select "good" negative kac

// make explicit double loop
for ( Range::iterator ik1 = kplus.begin() ; kplus.end() != ik1 ; ++ik1 )
{
  const LHCb::Particle* k1 = *ik1 ;
  for ( Range::iterator ik2 = kminus.begin() ; kminus.end() != ik2 ; ++ik2  )
   {
     const LHCb::Particle* k2 = *ik2 ;

     // evaluate the invariant mass:
     const double mass = ( k1->momentum() + k2->momentum() ).M() ;

     // fill the histo:
     plot ( mass / Gaudi::Units::GeV , "dikaon invariant mass in GeV " , 1. , 1.1 , 200 ) ;
   }
}
```

LoKi offers the alternative way of doing the same stuff using **`loop`** function and **`Loop`** object:

```
Range kplus = select ( "k+" , ... ) ; // you already know how to select "good" positive kaon
Range kminus = select ( "k-" , ... ) ; // you already know how to select "good" negative kac

// make a loop over all dikaon combinations:
for ( Loop dikaons = loop ( "k+ k-" ) ; dikaons ; ++dikaons )
{
     // fast evaluation of the invariant mass:
     const double mass = dikaon->mass(1,2) ;

     // fill the histo:
     plot ( mass / Gaudi::Units::GeV , "dikaon invariant mass in GeV " , 1. , 1.1 , 200 ) ;
}
```

For such trivial case both approaches (the native DaVinci) and LoKi (see above) are very similar. The clear difference appears immediately as soon as one goes to a bit more complicated tasks. Usually we have no interest in seeing the invariant mass of **all** combinations. If one tries to reconstruct $\phi \to \mathrm{K}^+\mathrm{K}^-$, one also performs the vertex fit and creates *the compound* particle for ϕ. You already know well how to do it in DaVinci - one needs to use explicitly some *vertex fitter tool*. In LoKi *the compound particle* is the effective "mother" particle of the loop automatically and implicitly on demand. Please, note that in order to use this functionality one needs to specify the identifier of the effective compound particle:

```
// make a loop over all dikaon combinations:
for ( Loop dikaons = loop ( "k+ k-" , "phi(1020)" ) ; dikaons ; ++dikaons )
{
     // get the effective particle of the loop:
     const LHCb::Particle* phi = dikaons->particle() ;

     // fill the histo:
     plot (  M ( phi ) / Gaudi::Units::GeV , "dikaon invariant mass in GeV " , 1. , 1.1 , 20
}
```

One also can get the access to *the effective vertex* of the dikaon combiation:

```
1000// make a loop over all dikaon combinations:
1010for ( Loop dikaons = loop ( "k+ k-" , "phi(1020)" ) ; dikaons ; ++dikaons )
1020{
1030     // get the effective vertex of the loop:
1040     const LHCb::Vertex* v = dikaons->vertex() ;
1050
1060     // fill the histo:
1070     plot (  VCHI2 ( v ) , "chi2 of the vertex fit " , 0 , 100 ) ; // function VCHI2 gets th
1080}
```

Please note that explicit cast to **`LHCb::Particle`** or **`LHCb::Vertex`** is usually not needed, one can use the looping object of type **`Loop`** directly as the argument for many functions:

```
1000// make a loop over all dikaon combinations:
1010for ( Loop dikaons = loop ( "k+ k-" , "phi(1020)" ) ; dikaons ; ++dikaons )
1020{
1030     // fill the histos:
1040      plot (  M ( phi ) / Gaudi::Units::GeV , "dikaon invariant mass in GeV " , 1. , 1.1 , 2
1050      plot (  VCHI2 ( dikaon ) , "chi2 of the vertex fit " , 0 , 100 ) ; // function VCHI2 g
1060}
```

Obviously the creation of the compound particle and vertex fit are the CPU-consuming procedures, therefore it is desirable to cut "non-interesting" combinations as soon as possible, e.g.:

```
1000// make a loop over all dikaon combinations:
1010for ( Loop dikaons = loop ( "k+ k-" , "phi(1020)" ) ; dikaons ; ++dikaons )
1020{
1030     // evaluate the mass through sum of 4-momenta (fast)
1040      const double mass  = dikaons->mass(1,2) ;
1050      // skip high-mass combiations
1060      if ( mass > 1100 * Gaudi::Units::MeV ) { continue ; }          // CONTINUE
1070      // fill the histos:
1080      plot (  M ( phi ) / Gaudi::Units::GeV , "dikaon invariant mass in GeV " , 1. , 1.1 , 2
1090      plot (  VCHI2 ( dikaon ) , "chi2 of the vertex fit " , 0 , 100 ) ; // function VCHI2 g
1100}
```

Often one needs to get the access to the daughter particle (e.g. to the first kaon, or to the second kaon). It could be done in an easy way:

```
1000// make a loop over all dikaon combinations:
1010for ( Loop dikaons = loop ( "k+ k-" , "phi(1020)" ) ; dikaons ; ++dikaons )
1020{
1030      // get the first kaon:
1040      const LHCb::Particle* k1 = dikaons(1) ;  // NB!   Indices start from 1 !
1050      // get the second kaon:
1060      const LHCb::Particle* k2 = dikaons(2) ;  // NB!   Indices start from 1 !
1070}
```

Finally one can apply some cuts for the compound particle and *save* "good" candidates:

```
1000// make a loop over all dikaon combinations:
1010for ( Loop dikaons = loop ( "k+ k-" , "phi(1020)" ) ; dikaons ; ++dikaons )
1020{
1030      if ( ... ) { continue ; }
1040      if ( ... ) { continue ; }
1050
1060      // save combination if good enough:
1070      phi -> save ( "myGoodPhi" ) ;
1080}
1090
```

The "saved" compound particles could be inspected through the function **`selected`**:

```
// make a loop over all dikaon combinations:
for ( Loop dikaons = loop ( "k+ k-" , "phi(1020)" ) ; dikaons ; ++dikaons )
{
      ....
      // save combination, if good enough:
      phi -> save ( "myGoodPhi" ) ;                // MIND THE NAME
}

// get the previously saved combinations:
Range phis = selected ( "myGoodPhi" ) ;     // NOTE THE NAME

always() << " Number of phi-candidates: " << phis.size() << endreq ;
// use the counter:
counter ("#phis") += phis.size() ;
```

Now you know all ingredients to code the simple algorithm which makes the loop over multi-particle combinations, applies some cuts, plots the distributions for the compound particle and saves interesting combinations for subsequent analysis. Let's try to write such algorithm for $\phi \to \mathrm{K}^+\mathrm{K}^-$ selection using your experience from the previous exercise:

1. select the good positive kaons
2. select the good negative kaons
3. make a loop over dikaons
4. apply some cuts for the compound particle
5. plot some distributions for the compound and/or daughter particles
    - if you already familar with N-tuples (see here), fill N-tuple with all these variables
6. save "interesting" combinations and count them

The algorithm:

Start new algorithm in **`src`** directory of **`Tutorial/LoKiTutor`** package:

```
#include "LoKi/LoKi.h"

LOKI_ALGORITHM(LoKiLoop)
{
    // avoid very long names:
    using namespace LoKi ;
    using namespace LoKi::Types ;
    using namespace LoKi::Cuts  ;

    ... put your lines here ...

    return StatusCode::SUCCESS ;                          // RETURN
}
```

Note some functions which could be useful (in addition to these functions):

`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 invariant mass of the particle and the nominal mass of ϕ.

The solution

The full **solution** to this exercise (***.cpp** & ***.py** files) is available in the directory **solutions/LoKiLoop/** in the **Tutorial/LoKiTutor** 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 LoKi.

LoKi offers 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 LoKi.

Selection of Monte Carlo particles

Following the major principle of the equality of all animals, LoKi offers the functionality of selection of Monte Carlo particles, which is very similar to the functionlity, described for the second exercise:

```
1000// get the true Monte Carlo positive kaons
1010MCRange mckaon = mcselect ( "mck+" , "K+" == MCID ) ;
1020
1030// get all beauty particles
1040MCRange beauty  = mcselect ( "mck+" , BEAUTY ) ;
1050
1060// get all true fast Monte Carlo muons from decay/interaction of selected beauty particles:
1070MCRange mcmu = mcselect ( "mcmu" , "mu+" == MCABSID && FROMMCTREE ( beauty ) && MCPT > 1 * G
```

The objects of type **MCRange** have the stantard interface of the container and could be inspected through the explicit loop:

```
1000MCRange mc =
1010
1020always() << " Number of MC-particles " << mc.size() << endreq ;
1030
1040// explicit loop:
1050for ( MCRange::iterator imc = mc.begin() ; mc.end() != imc ; ++imc )
1060{
1070    const LHCb::MCParticle* p = *imc ;
1080    plot ( MCP ( p ) /Gaudi::Units::GeV , "pt of Monte Carlo particle in GeV " , 0 , 10 )  ;
1090}
```

In these examples **MCID**, **MCPT**, **MCP**, **FROMMCTREE** and **BEAUTY** are LoKi *Monte Carlo particle functions*.

Selection of Monte Carlo decays

For the frequent case of the special selection of particles, which satisfy the certain decay patterns LoKi offers the special utility **MCFinder**, which is essentially just a thin wrapper over the brilliant tool **MCDecayFinder**, the masterpiece written by Olivier Dormond from Lausanne Univertity:

```
1000// get the wrapper for MCDecayFinder tool
1010MCFinder finder = mcFinder() ;
```

```
1020
1030// get all Monte Carlo psis, which decay into mu+ mu-:
1040MCRange mcPsi  = finder -> findDecays ( "J/psi(1S) ->  mu+  mu-") ;
1050
1060// get all Monte Carlo muons from the  decay  psi -> mu+ mu-:
1070MCRange mcmu  = finder -> findDecays ( "J/psi(1S) ->  ^mu+  ^mu-") ;
```

The selected Monte Carlo particles could be subjected to the further selection:

```
1000// from the selected true muons from psi->mu+mu- decay select only the fast muons:
1010MCRange fastMuons = mcselect ( "fastMu" , mcmu , MCPT > 0.5 * Gaudi::Units::GeV && 10 < MCP
1020
```

Match to Monte Carlo truth

There are variety of the methods in LoKi 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 particles) used for construction of the function. e.g. :

```
1000// retrieve Monte Carlo matching object:
1010MCMatch mc = mcTruth() ;                                  // the member function of class LoKi:
1020
1030// get some MC-particles, (e.g.mc-muons)
1040MCRange mcmu  = .... ;
1050
1060// create the function (predicate) using the list of true muons
1070Cut fromMC = MCTRUTH ( mc , mcmu ) ;  /// this function evaluated to "true" for particles, m
1080
1090// select recontructed muons, matched with true MC-muons:
1100Range mu = select ( "goodmu" , "mu+" == ABSID && fromMC ) ;  /// use the function/predicate
1110
```

The constructed function/predicate **`fromMC`** (the line 01070) could be used also directly:

```
1000const LHCb::Particle* p = ... ;
1010
1020if ( fromMC ( p ) )
1030 {
1040    ... it is a particle matched with true MC-muons ...
1050 }
```

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

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

```
1000// get all Monte Carlo psis, which decay into mu+ mu-:
1010MCRange mcPsi  = ... ;                  // you already know how to get them!
1020
1030// get all Monte Carlo muons from the  decay  psi -> mu+ mu-:
1040MCRange mcmu  = ... ;                  // you know well  how to get them!!!
1050
1060// create the function/predicate  for "true" psi
1070Cut truePsi = MCTRUTH ( mc , mcPsi ) ;  // check the matching with Monte Carlo true psi
1080
1090// create the function/predicate  for "true" muon from psi
1100Cut trueMu = MCTRUTH ( mc , mcmu ) ;  // check the matching with Monte Carlo true muon from
1110
1120// get the reocnstructed muons:
1130Range muplus = select ( "mu+" , .... )  ;                    //  you know well how to get t
```

```
Range muminus = select ( "mu-" , .... )  ;                    //  you know well how to get
// make the loop
for ( Loop psi = loop ( "mu+ mu-" , "J/psi(1S)" ) ; psi ; ++psi )
{
    // fast evaluation of mass
    const double m12 = psi->mass(1,2) ;
    if ( m12 < 2.0 * Gaudi::UnitsGeV || m12 > 4 * Gaudi::Units::GeV ) { continue ; } // skip

     plot ( m12 , "mass of all dimuons " , 2 , 4 ) ;
     const LHCb::Particle* mu1 = psi ( 1 ) ;   // access to the first daughter particle of t
     const LHCb::Particle* mu2 = psi ( 2 ) ;   // get the second daughter particle of the lo

    if ( trueMu ( mu1 ) || trueMu ( mu2 ) )    // use the matching predicates
    {
     plot ( m12 , "mass of all dimuons, at least one is true  " , 2 , 4 ) ; // at least one
    }
    if ( trueMu ( mu1 ) && trueMu ( mu2 ) )    // use the matching predicates
    {
      plot ( m12 , "mass of all dimuons, both muons are true  " , 2 , 4 ) ;  // both muons a
    }
    if ( truePsi ( psi ) )     // use the matching predicates
    {
      plot ( m12 , "mass of all dimuons, true J/psi " , 2 , 4 ) ; // the dimuon combination
    }
}
```

Now you know all the major ingredients useful for simple Monte Carlo match. Let's try to write the algorithm for $J/\psi \to \mu^+\mu^-$ selection using your experience from the previous exercise:

1. find true Monte Carlo decays $J/\psi \to \mu^+\mu^-$
2. create the helper matching predicates/functions for Monte match of J/ψ and $\mu^\pm$
3. select the good positive muons
4. select the good negative muos
5. make a loop over dimuons
6. apply some cuts to the compound particle
7. plot some distributions for the compound and/or daughter particles **with and without matching to Monte Carlo truth**
    - ♦ if you are already familiar with N-tuples (see here), fill N-tuple with all these variables
8. save "interesting" candidates and count them

The algorithm

Start new algorithm in **`src`** directory of **`Tutorial/LoKiTutor`** package:

```
#include "LoKi/LoKi.h"

LOKI_MCALGORITHM(PsiMC)
{
    // avoid very long names:
    using namespace LoKi ;
    using namespace LoKi::Types ;
    using namespace LoKi::Cuts ;

    ... put your lines here ...

    return StatusCode::SUCCESS ;                              // RETURN
}
```

Note: for access to Monte Carlo truth one needs to use the macro **`LOKI_MCALGORITHM`** instead of **`LOKI_ALGORITHM`**.

The configuration

The configuration of the job is quite standard and does not require the additional efforts with respect to the previous exercise. Please note that for this algorithm one needs only muons as input particles, therefore for the reasons of CPU efficiency other input locations could be suppressed. 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:

```
## use Monte Carlo truth only for charged tracks (speed-up the execution):
alg = MyALG (
    .... ,
  PP2MCs = [ "Relations/Rec/ProtoP/Charged' ]
)
```

The solution

The full **solution** to this exercise (**`*.cpp`** & **`*.py`** files) is available in the directory **`solutions/PsiMC/`** in the **`Tutorial/LoKiTutor`** package and could be inspected/copied in case of problems.

Combine everything altogether and get the nice $\mathrm{B}_\mathrm{s}^0 \to \mathrm{J}/\psi\phi$ peak.

The purpose of the next exercise is to combine all ingredient together and write the "realistic" but simple analysis algorithm for $\mathrm{B}_\mathrm{s}^0 \to \mathrm{J}/\psi\phi$ decay. Essentialy one now has all ingredients ready and one just need to combine them in a right way within one algorithm.

- Please note that it is a bit different from DaVinci approach where users encouraged to split the one algorithm into smaller algorithmm which run in the sequence. Here we'll study the possibility to recontruct the whole chain in one algorithm.

The analysis of $\mathrm{B}_\mathrm{s}^0 \to \mathrm{J}/\psi\phi$ decay chain could be naturally split into three similar phases

1. selection of $\mathrm{J}/\psi \to \mu^+\mu^-$ through the loop over dimuons
2. selection of $\phi \to \mathrm{K}^+\mathrm{K}^-$ through the loop over dikaons
3. selection of $\mathrm{B}_\mathrm{s}^0 \to \mathrm{J}/\psi\phi$ through the loop over selected J/ψ and ϕ candidates

From the Exercise 3 you already know well how to reach the first goal, and from the Exercise 4 you know how to reach the second goal. Therefore here one just needs to concentrate on the third item. The overall design of the algorithm could be sketched as:

```
#include "LoKi/LoKi.h"
LOKI_ALGORITHM(PsiPhi)
{

   ....
  Range kplus= select ( "k+" , ... ) ;
  Range kminus= select ( "k-" , ... ) ;
  for ( Loop dikaon = loop ( "k+ k-" , .... ) ...  )
  {
      ...
     dikaon -> save ( "phi" ) ;                // MIND THE NAME
  }

```

```
1130   ....
1140   Range muplus= select ( "m+" , ... ) ;
1150   Range muminus= select ( "m-" , ... ) ;
1160   for ( Loop dimuon =  loop (  "m+ m-" , ...) ...  )
1170   {
1180       ...
1190      dimuon -> save ( "psi" ) ;            // MIND THE NAME
1200   }
1210
1220   ... and here one needs to add the selection of Bs ...
1230
1240    return StatusCode::SUCCESS ;                              // RETURN
1250}
```

How to make the selection of $\mathrm{B}_\mathrm{s}^0 \to \mathrm{J}/\psi\phi$?

```
1000for (  Loop Bs = loop ( "psi phi" , "B_s0" ) ; Bs ; ++Bs )          /// MIND THE NAME
1010{
1020  ...
1030}
```

Please mind the names of "component" used for B_s^0-candidate selection: **`loop("psi phi",...)`**.

To make your lines more CPU-efficient, try to skip all unnesessary actions, e.g.:

```
1000Range muplus = select ( ... ) ;
1010if ( muplus.empty() ) { return StatusCode::SUCCEES ; }   // no need to continue the executio
1020...
1030for (  Loop dimuon = ... )
1040{
1050   ...
1060  dimuon->save ( "psi" ) ;
1070}
1080Range psis = selected ( "psi" ) ;
1090if ( psis.empty() ) { return StatusCode::SUCCESS ; } // no need to continue the execution .
1100
```

Plase also note that if one needs to code **selection** algorithm, one must take care about **`setFilterPassed`** method:

```
1000for (  Loop Bs = ... )
1010{
1020   ...
1030  Bs->save ( "Bs" ) ;
1040}
1050Range bs = selected ( "Bs" ) ;
1060
1070// filter decision depends on the saved Bs-candidates:
1080setFilterPassed ( !bs.empty() ) ;   // "setFilterPassed" is a member function of class Algor
1090
```

The algorithm

Start new algorithm in **`src`** directory of **`Tutorial/LoKiTutor`** package:

```
1000#include "LoKi/LoKi.h"
1010
1020LOKI_ALGORITHM(PsiPhi)
1030{
1040    // avoid very long names:
1050    using namespace LoKi ;
1060    using namespace LoKi::Types ;
1070    using namespace LoKi::Cuts  ;
```

Combine everything altogether and get the nice peak.

```
1080
1090    ... put your lines here, essentially the compilation from the previous examples.
1100
1110    return StatusCode::SUCCESS ;                                 // RETURN
1120}
```

Please note that if one wants to have access to Monte Carlo truth information, one must to use the macro **`LOKI_MCALGORITHM`**.

The solution

The full **solution** to this exercise (**`*.cpp`** & **`*.py`** files) is available in the directory **`solutions/PsiPhi/`** in the **`Tutorial/LoKiTutor`** package and could be inspected/copied in case of problems.

-- Vanya Belyaev - 30 July 2k+9

This topic: LHCb > LoKiTutorial
Topic revision: r14 - 2009-07-30 - VanyaBelyaev