## Table of Contents

**LHCb LoKi Tutorial: Getting started with LoKi**

- Prerequisites.........................................................................................................................1
- Setup the environment for Da Vinci..........................................................................................1
- The most trivial (empty) "Hello, World!" LoKi algorithm..........................................................2
  - The algorithm.......................................................................................................................2
  - The configuration..................................................................................................................2
  - The solution..........................................................................................................................4
- Simple selection of particles.......................................................................................................4
  - The configuration..................................................................................................................6
  - The solution..........................................................................................................................6
- The loops over multiparticle combinations..................................................................................7
  - The algorithm.......................................................................................................................9
  - The solution..........................................................................................................................10
- Easy matching to Monte Carlo truth..........................................................................................10
  - Selection of Monte Carlo particles.........................................................................................10
  - Selection of Monte Carlo decays............................................................................................10
  - Match to Monte Carlo truth..................................................................................................11
  - The algorithm.......................................................................................................................12
  - The configuration..................................................................................................................13
  - The solution..........................................................................................................................13
- Combine everything altogether and get the nice peak...............................................................13
  - The algorithm.......................................................................................................................14
  - The solution..........................................................................................................................15
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: $D_s^0 \rightarrow J/\psi \phi$.

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:

```cpp
// Include files
#include "LoKi/LoKi.h" // from the package Phys/LoKi

LOKI_ALGORITHM(HelloWorld) // algorithm C++ type as the parameter of macro
{
  always() << " Hello, World! " << endreq;
  return StatusCode::SUCCESS; // RETURN
}
```

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

```cpp
/**
 * it is a simple tutorial algorithm for ...
 * @author ...
 * @date ...
 */

// and/or:
/**
 * it is a simple tutorial algorithm for ...
 * @author ...
 * @date ...
 */
```

Put your algorithm into src directory 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:

```bash
#!/usr/bin/env gaudirun.py
```

```bash
#!/usr/bin/env gaudirun.py
```

```
@file
It is a simple tutorial algorithm for ...  
@author ...  
@date ...
```

```
@class AAA
It is a simple tutorial algorithm for ...
@author ...
@date ...
```

The simple configuration file to run the simplest 'Hello World!'
Clearly only few lines are important here:

definition of the data type:

definition of the list of algorithms:

definition of the number of events:

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.
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.

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

// 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 < PT / Gaudi::Units::MeV ) ;

// 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:

```cpp
// get all positive muons:
Range particles = ... ;

// print the size of the container
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 ) ;
}
```

The solution
If you are not familiar 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 muons or kaons.

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

```cpp
#include "LoKi/LoKi.h"
LOKI_ALGORITHM(GetData)
{
    // avoid very long names:
    using namespace LoKi;
    using namespace LoKi::Types;
    using namespace LoKi::Cuts;
    ...
    ... put your lines here ...
    return StatusCode::SUCCESS ;
}
```

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 momentum
- \( PIDK, PIDmu \) \( \Delta L_L \) (K-\( \pi \))
- \( PIDmu \) \( \Delta L_L \) (\( \mu - \pi \))

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

```cpp
// get all reconstructed primary vertices:
Range pvs = vselect ("pv", ISPRIMARY ) ;
// create the function from the list of primary vertices and some helper object (essentially it is a pointer to IDistanceCalculator tool)
Fun fun = MIPCHI2 ( pvs , geo() ) ;  // "geo" is a member function of class LoKi::Algo
// use it explicitly:
const LHCb::Particle* p = ... ;
const double minChi2 = fun ( p ) ;
// use it for selection:
Range 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:

```python
#!/usr/bin/env gaudirun.py

# The configuration file for 'GetData'-solution for LoKi-tutorial
@file
@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',
                InputLocations = [ 'StdLoosePions', 'StdTightKaons', 'StdLooseMuons' ]
)

DaVinci (DataType = 'DC06',
         UserAlgorithms = [ data ],
         EvtMax = 100)

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

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 $K^+K^-$ 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:

```cpp
1000 Range kplus = select ( "k+", ... ) ; // you already know how to select "good" positive kaons!
1010 Range kminus = select ( "k-", ... ) ; // you already know how to select "good" negative kaons!
1020 // make explicit double loop
1040 for ( Range::iterator ik1 = kplus.begin() ; kplus.end() != ik1 ; ++ik1 )
1050 { const LHCb::Particle* k1 = *ik1 ;
1060 for ( Range::iterator ik2 = kminus.begin() ; kminus.end() != ik2 ; ++ik2 )
1080 {
1090 const LHCb::Particle* k2 = *ik2 ;
1100 // evaluate the invariant mass:
1120 const double mass = ( k1->momentum() + k2->momentum() ).M() ;
1130 // fill the histo:
1150 plot ( mass / Gaudi::Units::GeV , "dikaon invariant mass in GeV " , 1. , 1.1 , 200 ) ;
1160 }
1170 }
```

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

```cpp
1000 Range kplus = select ( "k+", ... ) ; // you already know how to select "good" positive kaons!
1010 Range kminus = select ( "k-", ... ) ; // you already know how to select "good" negative kaons!
1020 // make a loop over all dikaon combinations:
1040 for ( Loop dikaons = loop ( "k+ k-" ) ; dikaons ; ++dikaons )
1050 {
1060 // fast evaluation of the invariant mass:
1070 const double mass = dikaons->mass(1,2) ;
1080 // fill the histo:
1100 plot ( mass / Gaudi::Units::GeV , "dikaon invariant mass in GeV " , 1. , 1.1 , 200 ) ;
1110 }
```

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 $\phi \rightarrow K^-K^-$, one also performs the vertex fit and creates the compound particle for $\phi$. 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:

```cpp
1000 // make a loop over all dikaon combinations:
1010 for ( Loop dikaons = loop ( "k+ k-" , "phi(1020)" ) ; dikaons ; ++dikaons )
1020 {
1030 // get the effective particle of the loop:
1040 const LHCb::Particle* phi = dikaons->particle() ;
1050 // fill the histo:
1070 plot ( M ( phi ) / Gaudi::Units::GeV , "dikaon invariant mass in GeV " , 1. , 1.1 , 200 ) ;
1080 }
```

One also can get the access to the effective vertex of the dikaon combination:
// make a loop over all dikaon combinations:
for ( Loop dikaons = loop ( "k+ k-" , "phi(1020)" ) ; dikaons ; ++dikaons )

// get the effective vertex of the loop:
const LHCb::Vertex* v = dikaons->vertex();

// fill the histo:
plot (  VCHI2 ( v ) , "chi2 of the vertex fit " , 0 , 100 ) ; // function VCHI2 gets the vertex as the argument

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:

for ( Loop dikaons = loop ( "k+ k-" , "phi(1020)" ) ; dikaons ; ++dikaons )

// fill the histos:
plot (  M ( phi ) / Gaudi::Units::GeV , "dikaon invariant mass in GeV " , 1. , 1.1 , 200 ) ;
plot (  VCHI2 ( dikaon ) , "chi2 of the vertex fit " , 0 , 100 ) ; // function VCHI2 gets the vertex as the argument

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.:

for ( Loop dikaons = loop ( "k+ k-" , "phi(1020)" ) ; dikaons ; ++dikaons )

const double mass  = dikaons->mass(1,2) ;
if ( mass > 1100 * Gaudi::Units::MeV ) { continue ; } // CONTINUE

// fill the histos:
plot (  M ( phi ) / Gaudi::Units::GeV , "dikaon invariant mass in GeV " , 1. , 1.1 , 200 ) ;
plot (  VCHI2 ( dikaon ) , "chi2 of the vertex fit " , 0 , 100 ) ; // function VCHI2 gets the vertex as the argument

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:

for ( Loop dikaons = loop ( "k+ k-" , "phi(1020)" ) ; dikaons ; ++dikaons )

const LHCb::Particle* k1 = dikaons(1) ; // NB!  Indices start from 1 !
const LHCb::Particle* k2 = dikaons(2) ; // NB!  Indices start from 1 !

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

for ( Loop dikaons = loop ( "k+ k-" , "phi(1020)" ) ; dikaons ; ++dikaons )

if ( ... ) { continue ; }
if ( ... ) { continue ; }

// save combination if good enough:
phi -> save ( "myGoodPhi" ) ;

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 \( \phi \rightarrow K^+K^- \) 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 familiar 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:

```c++
#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()`, \( \sqrt{E^2 - p^2} \)

### 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)")` evaluates the difference between the invariant mass of the particle and the nominal mass of \( \phi \).
**ADMASS**

The function evaluates the absolute value of the invariant mass difference with respect to some reference mass: e.g. **ADMASS(“phi(1020”)** evaluates the absolute value of the difference between the invariant mass of the particle and the nominal mass of $\phi$.

**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 Carlo 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 functionality, described for the second exercise:

```cpp
// get the true Monte Carlo positive kaons
MCRange mckaon = mcselect ( "mck+", "K+" == MCID ) ;

// get all beauty particles
MCRange beauty = mcselect ( "mck+", BEAUTY ) ;

// get all true fast Monte Carlo muons from decay/interaction of selected beauty particles:
MCRange mcmu = mcselect ( "mcmu", "mu+" == MCABSID && FROMMCTREE ( beauty ) && MCPT > 1 * Gaudi::Units::GeV ) ;
```

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

```cpp
MCRange mc =

// always() << " Number of MC-particles " << mc.size() << endreq ;

// explicit loop:
for ( MCRange::iterator imc = mc.begin() ; mc.end() != imc ; ++imc )
{
    const LHCb::MCParticle* p = *imc ;
    plot ( MCP ( p ) /Gaudi::Units::GeV , "pt of Monte Carlo particle in GeV " , 0 , 10 ) ;
}
```

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 University:

```cpp
// get the wrapper for MCDecayFinder tool
MCFinder finder = mcFinder() ;
```
// get all Monte Carlo psis, which decay into mu+ mu-:
MCRange mcPsi = finder -> findDecays ("J/psi(1S) -> mu+ mu-" ) ;

// get all Monte Carlo muons from the decay psi -> mu+ mu-:
MCRange mcmu = finder -> findDecays ("J/psi(1S) -> ^mu+ ^mu-" ) ;

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

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

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.:

// retrieve Monte Carlo matching object:
MCMatch mc = mcTruth() ;

// get some MC-particles, (e.g.mc-muons)
MCRange mcmu = .... ;

// create the function (predicate) using the list of true muons
Cut fromMC = MCTRUTH ( mc , mcmu ) ;

// select reconstructed muons, matched with true MC-muons:
Range mu = select ("goodmu","mu+"==ABSID && fromMC ) ;

The constructed function/predicate fromMC (the line 1070) could be used also directly:

const LHCb::Particle* p = ... ;
if ( fromMC ( p ) )
{
... it is a particle matched with true MC-muons ...
}

Important note: the function MCTRUTH evaluates to true also for reconstructed 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:

// get all Monte Carlo psis, which decay into mu+ mu-:
MCRange mcPsi = ... ;

// get all Monte Carlo muons from the decay psi -> mu+ mu-:
MCRange mcmu = ... ;

// create the function/predicate for "true" psi
Cut truePsi = MCTRUTH ( mc , mcPsi ) ;

// check the matching with Monte Carlo true psi
Cut trueMu = MCTRUTH ( mc , mcmu ) ;

// check the matching with Monte Carlo true muon from psi
Range muplus = select ("mu+", .... ) ;
// you know well how to get the reconstructed muons:
LoKiTutorial < LHCb < TWiki

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::Units::GeV || m12 > 4 * Gaudi::Units::GeV ) { continue ; } // skip
    const LHCb::Particle* mu1 = psi ( 1 ) ; // access to the first daughter particle of the loop
    const LHCb::Particle* mu2 = psi ( 2 ) ; // get the second daughter particle of the loop
    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 are true
    }
    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 $\text{J/}\psi \rightarrow \mu^+ \mu^-$ selection using your experience from the previous exercise:

1. find true Monte Carlo decays $\text{J/}\psi \rightarrow \mu^+ \mu^-$
2. create the helper matching predicates/functions for Monte match of $\text{J/}\psi$ and $\mu^\pm$
3. select the good positive muons
4. select the good negative muons
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:

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

LOKI_MCALGORITHM(PsiMC)
{
    using namespace LoKi ;
    using namespace LoKi::Types ;
    using namespace LoKi::Cuts ;
    // 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`.

Match to Monte Carlo truth
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 performance 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 (speed-up the execution):
1010alg = MyALG (  
1020....,  
1030PP2MCs = [ "Relations/Rec/ProtoP/Charged" ]  
1040)
1050
1060

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 $B_s^0 \rightarrow J/\psi \phi$ peak.

The purpose of the next exercise is to combine all ingredient together and write the "realistic" but simple analysis algorithm for $B_s^0 \rightarrow J/\psi \phi$ decay. Essentially 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 algorithmmm which run in the sequence. Here we'll study the possibility to reconstrcut the whole chain in one algorithm.

The analysis of $B_s^0 \rightarrow J/\psi \phi$ decay chain could be naturally split into three similar phases

1. selection of $J/\psi \rightarrow \mu^+ \mu^-$ through the loop over dimuons
2. selection of $\phi \rightarrow K^- K^+$ through the loop over dikaons
3. selection of $B_s^0 \rightarrow J/\psi \phi$ through the loop over selected $J/\psi$ and $\phi$ 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:

1000#include "LoKi/LoKi.h"
1010LOKI_ALGORITHM(PsiPhi)
1020{
1030....
1050Range kplus= select ( "k+" , ... ) ;
1060Range kminus= select ( "k-" , ... ) ;
1070for ( Loop dikaon = loop ( "k+ k-" ,.... ) ... )
1080{
1090....
1100dikaon -> save ( "phi" ) ; // MIND THE NAME
1110}
1120

The configuration
LoKiTutorial < LHCb < TWiki

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 $B^0_s \rightarrow J/\psi \phi$?

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

Please mind the names of "component" used for $B^0_s$-candidate selection: loop("psi phi",...).

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

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

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

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

The algorithm

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

1000 #include "LoKi/LoKi.h"
1010 1020 LOKI_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.
... put your lines here, essentially the compilation from the previous examples.

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

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