Table of Contents

LoKi User Guide

For completeness see also LoKi Reference Manual

Abstract

LoKi is a package for the simple and user-friendly data analysis. LoKi is based on Gaudi architecture. The current functionality of LoKi includes the selection of particles, manipulations with predefined kinematic expressions, loops over combinations of selected particles, creation of composite particles from various combinations of particles, flexible manipulation with various kinematic constrains and access to Monte Carlo truth information.

LoKi User Guide

Introduction

All off-line OO software for simulation, digitization, recontruction, vizualization and analysis, for LHCb collaboration is based on Gaudi framework. All software is written on C++, which is currently the best suited language for large scale OO software projects. It is worth to mention here that for the experts coding in C++ is like a real fun. The language itself and its embedded abilities to provide *ready-to-use'* nontrivial, brilliant and exiting solution for almost all ordinary, tedious and quite boring problems seems to be very attractive features for persons who has some knowledge and experience with OO programming. Unfortunately C++ requires significant amount of efforts from beginners to obtain some first results of acceptable quality. An essential static nature of the language itself requires the knowledge of compilation and linkage details. In addition quite often in the *"typical"* code fragments for physical analysis the explicit C++ semantics and syntax hide the *"physical"* meaning of the line and thus obscure the physical analysis algorithm. Often a simple operation corresponding to one *"physics"* statement results in an enormous code with complicated and probably non-obvious content:

```
1000ParticleVector::const_iterator im;
1010for ( im = vDsK.begin(); im != vDsK.end(); im++ ) {
1020      if((*im)->particleID().pid() == m_DsPlusID||
1030      (*im)->particleID().pid() == m_DsMinusID) vDs.push_back(*im);
1040      else if((*im)->particleID().pid() == m_KPlusID||
1050      (*im)->particleID().pid() == m_KMinusID) vK.push_back(*im);
1060      else{
1070        log <<MSG::ERROR<< " some message here "<<endreq;
1080        return StatusCode::FAILURE;
1090        }
1100      }
```

The usage of comments becomes mandatory for understanding makes the code even longer and again results in additional complexity.

The idea of LoKi package is to provide the users with possibility to write the code, which does not obscure the actual physics content by technical C++ semantic. The idea of user-friendly components for physics analysis were essentially induced by the spirit of following packages:

- **KAL** language (*Kinematical Analysis Language*) by genius Hartwig Albrecht. **KAL** is an interpreter language written on **FORTRAN** (*It is cool, isn't it?*). The user writes script-like ASCII file, which is interpreted and executed by standard **KAL** executable. The package was very successfully used for physics analysis by ARGUS collaboration.
- **Pattern** and **GCombiner** packages by Thorsten Glebe. These nice, powerful and friendly C++ components are used now for the physics analysis by HERA-B collaboration
- Some obsolete **CLHEP** classes, like **HepChooser** and **HepCombiner**
- Loki library by Andrei Alexandrescu. The library from one side is a state-of-art for so called *generic meta-programming and compile time programming*, and simultaneously from another side it is the

excellent cook-book, which contains very interesting, non-trivial and non-obvious recipes for efficient solving of major common tasks and problems.

The attractiveness of *specialized*, physics-oriented code for physics analysis could be demonstrated e.g. with *"typical"* code fragment in **KAL**:

```
1000HYPOTH  E+  MU+  PI+ 5  K+  PROTON
1010
1020IDENT PI+     PI+
1030IDENT K+      K
1040
1050IDENT PROTON  PROTON
1060IDENT E+      E+
1070IDENT MU+     MU+
1080
1090SELECT K- PI+
1100  IF P > 2 THEN
1110    SAVEFITM D0 DMASS 0.045 CHI2 16
1120  ENDIF
1130ENDSEL
1140
1150SELECT D0 PI+
1160  PLOT MASS L 2.0 H 2.100 NB 100 TEXT ' Mass of D0 pi+ '
1170ENDSEL
1180
1190GO 1000000
```

This **KAL** pseudo-code gives an example of self-explanatory code. The physical content of selection of $D^{*+} \to D^0\pi^+$, followed by $D^0 \to K^-\pi^+$ decay is clear and unambigously visible between these lines. Indeed no comments are needed for understanding the analysis within 2 minutes.

One could argue that it is not possible to get the similar transparency of the physical content of code with native C++. The best answer to this argument could be just an example from T. Glebe's **Pattern** of $K^0_S \to \pi^+\pi^-$ reconstruction:

```
1000TrackPattern    piMinus = pi_minus.with ( pt > 0.1 & p > 1 ) ;
1010
1020TrackPattern    piPlus  = pi_plus.with   ( pt > 0.1 & p > 1 ) ;
1030
1040TwoProngDecay kShort  = K0S.decaysTo  ( PiPlus & PiMinus ) ;
1050
1060kShort.with ( vz > 0   ) ;
1070
1080kShort.with ( pt > 0.1 ) ;
```

This code fragment is not so transparent as specialized **KAL** pseudo-code but it is easy-to-read, the physical content is clear, and it is just a *native* C++! I personally tend to consider the above code as an *experimental prove* of possibility to develop easy-to-use C++ package for physics analysis. Indeed the work has been started soon after I've seen these 5 lines.

Here it is a good moment to jump to the end of the whole story and present some LoKi fragment for illustration:

```
1000select ( "Pi+" , ID =="pi+" && P > 5 * GeV ) ;
1010
```

```
1020select ( "K-"  , ID =="K-"  && P > 5 * GeV ) ;
1030
1040for ( Loop D0 = loop ("K- pi+" , "D0" ) ; D0 ; ++D0 )
1050   {
1060     if ( P ( D0 ) > 10 * GeV ){ D0 -> save ( "D0" ) ; }
1070   }
1080
1090for ( Loop Dstar = loop ( "D0 Pi+" , "D*+" ) ; Dstar ; ++Dstar )
1100   {
1110     plot ( M ( Dstar ) / GeV , " Mass of D0 pi+ " , 2.0 , 2.1 ) ;
1120   }
```

The physical content of these lines is quite transparent. Again I suppose that it is not obscured with C++ semantics. From these LoKi lines it is obvious that an essential emulation of **KAL** semantics is performed. Indeed I think that **KAL** was just state-of-art for physics pseudo-code and is practically *impossible* to make something better. But of course it is the aspect where I am highly biased.

LoKi follows general Gaudi architecture and indeed it is just a thin layer atop of tools, classes, methods and utilities from developed within DaVinci project.

Since LoKi is just a thin layer, all DaVinci toolsare available in LoKi and could be directly invoked and manipulated. However there is no need in it, since LoKi provides the physicist with *significantly* simpler, better and more friendly interface.

Acknowledgments

As a last line of this chapter I'd like to thank Galina Pakhlova, Andrey Tsaregorodtsev and Sergey Barsuk for fruitfull discussions and active help in overall desing of LoKi. It is a pleasure to thank Andrey Golutvin as *the first active user* of LoKi for constructive feedback. Many people have contributed in a very constructive way into available LoKi functionality and development directions. Within them I would like to thank Victor Coco, Hans Dijkstra, Markus Frank, Jose Angel Hernando Morata, Jeroen van Hunen, Sander Klaus, Patrick Koppenburg, Pere Mato, Juan Palacios, Boleslav Pietrzyk, Gerhard Raven, Thomas Ruf, Hugo Ruiz Perez, Jeroen van Tilburg, and Benoit Viaud. It is the real pleasure to thank the leaders of ITEP/Moscow (Andrey Golutvin), CERN-LBD (Hans-Jurgen Hilke and Hans Dijkstra), LAPP/Annecy (Marie-Noelle Minard and Boleslav Pietrzyk) and Syracuse University (Sheldon Stone and Tomasz Skwarnicki) teams for the kind support.

Who is LoKi?

- ***Loki*** is a god of wit and mischief in Norse mythology
- ***LoKi*** could be interpreted as *Loops&Kinematics*

LoKi ingredients

Typical analysis algorithm consists of quite complex combination of the following elementary actions:

- *selection/filtering* with the criteria based on particle(s) kinematics, identification and topological properties, e.g. particle momenta, transverse momenta, confidence levels, impact parameters etc.
- *looping* over the various combinations of selected particles and applying other criteria based on kinematic properties of the whole combination or any sub-combinations or some topology information (e.g. vertexing), including mass and/or vertex constrain fits.
- *saving* of interesting combinations as *"particles"* which acquire all kinematic properties and could be further treated in the standard way.
- for evaluation of efficiencies and resolutions *the access for Monte Carlo truth* information is needed.
- also required is *the filling of histograms and N-tuples*.

LoKi has been designed to attack all these five major actions.

Selection

Selection of *Particles*

LoKi allows to select/filter a subset of reconstructed *particles* (of C++ type **LHCb::Particle**) which fulfills the chosen criteria, based on their kinematic, identification and topological properties and to refer later to this selected subset with the defined tag:

```
1000select ( "AllKaons" , abs ( ID ) == 321 && PT > 100 * MeV ) ;
```

Here from *all particles*, loaded by DaVinci, the subset of particles identified as charged kaons (`abs(ID)==321`) with transverse momentum greater than 100 *MeV/c* (`PT>100*MeV`) is selected. These particles are copied into internal local LoKi storage and could be accessed later using the symbolic name `"AllKaons"`.

In this example **ID** and **PT** are predefined LoKi *variables* or *functions* (indeed they are *function objects*, or *functors* in C++ terminology) which allow to extract the identifier and the transverse momentum for the given particle. *Cuts* or *predicates* or *selection criteria* are constructed with the comparison operations '<', '<=', '==', '!=', '>=', '>' from *variables*. The arbitrary combinations of *cuts* with the boolean operations '&&' or '||' are allowed to be used as selection criteria.

LoKi defines many frequently used *variables* and set of the regular mathematical operation on them '+', '-', '*', '/' and all elementary functions, like **sin**, **cos**, **log**, **atan**, **atan2**, **pow** etc, which could be used for construction of *variables* of the arbitrary complexity. ***Cuts*** and ***variables*** are discussed in detail in subsequent chapters

Indeed the function `select` has a return value of type `Range`, which is essentially the light-weight container of selected *particles*. The returned value could be used for immediate access to the selected particles and in turn could be used for further sub-selections. The following example illustrates the idea: the selected sample of kaons is

subdivided into samples of positive and negative kaons:

```
1000Range kaons =  select ( "AllKaons" , abs ( ID ) == 321 && PT > 100 * MeV );
1010
1020select ( "kaon+" , kaons , Q >   0.5 ) ;
1030
1040select ( "kaon-" , kaons , Q < -0.5 ) ;
```

Here all *positive kaons* (Q>0.5) are selected into the subset named "kaon+" and all *negative kaons* (Q<-0.5) go to the subset named "kaon-". These subsets again could be subject to further selection/filtering in a similar way.

LoKi allows to perform selection of *particles* from *standard* DaVinci containers of *particles* LHCb::Particle::Vector, LHCb::Particle::ConstVector and LHCb::Particle::Container (also known as LHCb::Particles).

```
1000const LHCb::Particle::ConstVector& particles = ... ;
1010
1020Range kaons_1 =  select ( "Kaons_1" , particles , abs( ID ) == 321 ) ;
1030
1040const LHCb::Particle::Container* event = get<LHCb::Particle::Container>( "..." ) ;
1050
1060Range kaons_2 =  select ( "Kaons_2" , event     , 321 == abs( ID ) ) ;
```

Also any arbitrary sequnce of objects, implicitely convertible to the C++ type const LHCb::Particle* can be used as input for selection of particles:

```
1000/// SEQUENCE is an arbitrary sequence of objects,
1010/// implicitely convertible to type const LHCb::Particle*
1020/// e.q. std::vector<LHCb::Particle*>,
1030/// LHCb::Particle::ConstVector, LHCb::Particles, std::set<LHCb::Particle*> etc.
1040SEQUENCE particles = ... ;
1050
1060Range kaons =  select
1070   ( "AllKaons"          ,   // 'tag'
1080     particles.begin () ,   // begin of sequence
1090     particles.end   () ,   // end of sequence
1100     abs ( ID ) == 321   ) ; // cut
1110
```

The output of selection (object of type Range) could be directly inspected through the explicit loop over the content of the selected container:

```
1000Range kaons =  select( "AllKaons" , abs( ID ) == 321 && PT > 100 * MeV ) ;
1010
1020// regular C++ loop:
1030for ( Range::iterator kaon = kaons.begin() ; kaons.end() != kaon ; ++kaon )
1040   {
1050     const LHCb::Particle* k = *kaon ;
1060     /* do something with this raw C++ pointer */
1070   }
```

Selection of *Monte Carlo Particles*

In a similar way one can select *Monte Carlo particles* (of C++ type LHCb::MCParticle), which satisfy the certain criteria:

```
1000MCRange kaons = mcselect ( "AllMCKaons" , abs( MCID ) == 321 && MCPT > 100 * MeV  ) ;
1010
```

```
1020MCRange k1 = mcselect ( "mcK+" , kaons , MC3Q >=  1 ) ;
1030
1040MCRange k2 = mcselect ( "mcK-"  , kaons , MC3Q <= -1 ) ;
1050
1060// regular C++ loop:
1070for ( MCRange::iterator ik1 = k1.begin() ; k1.end() != ik1 ; ++ik1 )
1080       {
1090          const LHCb::MCParticle* mc = *k1 ;
1100          /* do something with this raw C++ pointer */
1110       }
```

The differences with respect the previous case are

- one needs to use the function `mcselect` instead of `select`
- the return value of this function has C++ type `MCRange` and behaves like the container of `const LHCb::MCParticle*`
- for selection one needs to use the special *Monte Carlo variables & cuts*, e.g. in this example `MCID`, `MCPT` and `MC3Q`.

Selection of *Generator Particles*

Similar to the selection of reconstructed particles and the selection of Monte Carlo particles one can perform the selection of *Generator particles* (of C++ type `HepMC::GenParticle`):

```
1000GRange kaons = gselect ( "AllGenKaons" , abs( GID ) == 321 && GPT > 100 * MeV  ) ;
1010
1020GRange k1 = gselect ( "genK+" , kaons , G3Q >=  1 ) ;
1030GRange k2 = gselect ( "genK-"  , kaons , G3Q <= -1 ) ;
1040
1050// regular C++ loop:
1060for ( GRange::iterator ik1 = k1.begin() ; k1.end() != ik1 ; ++ik1 )
1070    {
1080          const HepMC::GenParticle* gen = *k1 ;
1090          /* do something with this raw C++ pointer */
1100    }
```

One sees that the C++ code essentially the same with the minor difference:

- one needs to use the function `gselect` instead of `select` and `mcselect`
- the return value of this function has C++ type `GRange` and behaves like the container of `const HepMC::GenParticle*`
- for selection one needs to use the special *Generator variables & cuts*, e.g. in this example `GID`, `GPT` and `G3Q`.

Selection of *Vertices*

The similar approach is used for *selection/filtering of vertices* (of C++ type `LHCb::VertexBase`):

```
1000VRange vs  = vselect( "GoodPVs" ,
1010                  PRIMARY && 5 < VTRACKS && VCHI2 / VDOF < 10 );
1020
```

Here from *all vertices* loaded by DaVinci one selects only the vertices tagged as *"Primary Vertex"* (PRIMARY) and constructed from more that 5 tracks (5<VTRACKS) and with $\chi^2/{\mathrm{nDoF}}$ less than 10 (VCHI2/VDOF<10). This set of selected vertices is named as "GoodPVs".

Again PRIMARY, VTRACKS, VCHI2 and VDOF are predefined LoKi *Vertex functions & cuts*. It is internal convention of LoKi that all predefined *functions*, types and methods for *vertices* start their names from capital letter V. As an example one see type VRange for a light pseudo-container of const LHCb::VertexBase*, function vselect and all *variables* for *vertices*.

Also there exist the variants of vselect methods, which allow the subselection of vertices from already selected *ranges of vertices* (C++ type VRange), standard DaVinci containers (C++ types LHCb::VertexBase::Vector, LHCb::VertexBase::ConstVector, LHCb::VertexBase::Container, LHCb::Vertex::Vector, LHCb::Vertex::ConstVector, LHCb::Vertex::Container, LHCb::RecVertex::Vector, LHCb::RecVertex::ConstVector, LHCb::RecVertex::Container and from arbitary sequence of objects, convertible to const LHCb::VertexBase*:

```
1000VRange vertices_1 = ... ;
1010VRange vs1 =
1020       vselect(  "GoodPVs1"  ,              // 'tag'
1030                 vertices_1  ,              // input vertices
1040                 PRIMARY && 5 < VTRACKS  ); // cut
1050
1060LHCb::VertexBase::ConstVector vertices_2 = ... ;
1070VRange vs2 =
1080       vselect(  "GoodPVs2"  ,              // 'tag'
1090                 vertices_2  ,              // input vertices
1100                 PRIMARY && 5 < VTRACKS  ); // cut
1110
1120/// arbitrary sequence of ojbects, implicitely convertible to
1130/// type LHCb::VertexBase*, e.g. std::vector<LHCb::VertexBase*>
1140SEQUENCE vertices_3 = ... ;
1150VRange vs3 =
1160      vselect(  "GoodPVs4"           ,      // 'tag'
1170                vertices_3.begin () ,      // begin of input sequence
1180                vertices_3.end   () ,      // end of input sequence
1190                PRIMARY && 5 < VTRACKS  ); // cut
1200
```

In summary, for selection of *vertices*:

- one needs to use the function vselect
- the return value of this function has C++ type VRange and behaves like the container of const LHCb::VertexBase*
- for selection one needs to use the special *Vertex variables & cuts*, e.g. in this example PRIMARY, VTRACKS, VCHI2 and VDOF.

Selection of *Monte Carlo Vertices* and *Generator Vertices*

The selection of *Monte Carlo Vertices* and *Generator Vertices* is performed similar to the the selection of reconstructed Vertices with following difference:

- one needs to use the functions `mcvselect` and `gvselect` for *Monte Carlo Vertices* and *Generator Vertices* respectively
- the return value of this function has C++ type `MCVRange` (`GVRange`) and behaves like the container of `const LHCb::MCVertex*` (`const HepMC::GenVertex*`) for *Monte Carlo Vertices* and *Generator Vertices* respectively
- for selection one needs to use the special *Monte Carlo Vertex variables & cuts* or *Generator Vertex variables & cuts*

Selection of minimal/maximal candidate

It is not unusual to select from some sequence or container of objects the object which maximizes or minimizes some *function*. The selection of *the primary vertex* with the maximal multiplicity could be considered as typical example:

```
1000// get all primary vertices
1010VRange vrtxs     = vselect( "GoodPVs" , PRIMARY ) ;
1020
1030const  LHCb::VertexBase* vertex  = select_max( vrtxs , VTRACKS ) ;
1040
```

Here from the preselected sample of primary vertices `vrtxs` one selects only one vertex which maximizes *Vertex function* `VTRACKS` with value equal to the number of tracks participating in this primary vertex.

```
1000// get all primary vertices:
1010VRange vrtxs          = vselect( "GoodPVs" , PRIMARY );
1020
1030// get B-candidate
1040const LHCb::Particle* B = ...
1050
1060// find the primatry vertex with minimal B-impact parameter
1070const LHCb::VertexBase* vertex  = select_min( vrtxs , VIP( B , geo() ) ) ;
1080
```

Here from the preselected sample of primary vertices `vrtxs` one selects the only vertex which minimize *function* `VIP`, with value equal to the impact parameter of the given particle (e.g. B-candidate) with respect to the vertex.

The templated methods `select_max` and `select_min` are *type-blind* and they could be applied e.g. to the containers of *particles*:

```
1000Range kaons  = select( .... );
1010
1020const LHCb::Particle* kaon  = select_min( kaons  , abs( PY ) + sin( PX )   ) ;
1030
```

Here from the container of preselected kaons the particle, which gives the minimal value of funny combination $|\mathrm{p_y}| + \sin \mathrm{p_x}$ is selected.

The methods `select_min_` and `=select_max` also allow *the conditional selection of minimal/maximal candidate*:

```
1000MCRange kaons = mcselect( "kplus" , MCID == "K+" ) ;
1010
```

```
MCRange::iterator igood = select_max
                        ( kaons.begin() ,    // begin of the sequence
                          kaons.end  () ,    // end of the sequence
                          MCPT          ,    // function to be maximized
                          0 < MCPZ      ) ;  // condition/cut

if ( kaons.end() != igood  )
     {
         const LHCb::MCParticle* good = *igood ;
        /* do something with this raw C++ pointer */
      }
```

Here the maximum is searched only within the particles whcih satisfy the *cut* (`0<MCPZ`) - the z-component of particle momenta must be positive..

Access to the selected objects

All selection functions, described above returns the light pseudocontainer of selected objects. Alternatively the selection result could be accessed using the unique tag (used as the first string argument of the functions select) and the function selected:

```
// get all previously selected particles, tagged as "MyGoodKaons":
Range goodK = selected ( "MyGoodKaons" ) ;

// get all previosly selected Monte Carlo particles, tagged as "TrueMCkaons":
MCRange mcK = mcselected ( "TrueMCkaons" ) ;

// get all previosly selected Generator particles, tagged as "My good b-quarks":
GRange bquarks = mcselected ("My good b-quarks") ;
```

Loops

Looping over the reconstructed particles

Simple one-particle loops

Above it has been already shown how to perform simple looping over *the selected range of particles*:

```
Range kaons =  select( "AllKaons" , abs( ID ) == 321 && PT > 100 * MeV );

for ( Range::iterator kaon = kaons.begin() ; kaons.end() != kaon ; ++kaon )
    {
      const LHCb::Particle* k = *kaon ;
      /* do something with this raw C++ pointer */
    }
```

Equivalently one can use methods `Range::operator()`, `Range::operator[]` or `Range::at()`:

```
Range kaons =  select ( "AllKaons" , abs( ID ) == 321 && PT > 100 * MeV );

for ( unsigned int index = 0 ; index < kaons.size() ; ++index )
```

```
1030      {
1040         const LHCb::Particle* k1 = kaons     ( index ) ;  // use Range::operator()
1050         const LHCb::Particle* k2 = kaons     [ index ] ;  // use Range::operator[]
1060         const LHCb::Particle* k3 = kaons .at ( index ) ; // use Range::at
1070      }
1080
```

The result of operators are not defined for invalid index, and `Range::at` method throws an exception for invalid index.

In principle one could combine these one-particle loops to get the effective loops over multi-particle combinations. But this gives no essential gain.

Loops over *the multi-particle combinations*

Looping over multi-particle combinations is performed using the special object `Loop`. All native C++ semantics for looping is supported by this object, e.g. for native C++ `for` -loop:

```
1000// loop over all "kaon- kaon+ "combinations
1010for  ( Loop phi = loop ( "kaon- kaon+" ) ; phi ; ++phi )
1020    {
1030       /* do something with the combination */
1040    }
```

The `while` -form of the native C++ loop is also supported:

```
1000Loop phi = loop( "kaon- kaon+" ) ;
1010while ( phi )
1020      {
1030         /* do something with the combination */
1040
1050         ++phi ; // go to the next valid combination
1060      }
```

The parameter of `loop` function is *the selection formula* (blank or comma separated list of particle tags). All items in the selection formula must be known for LoKi, e.g. previously selected using `select` functions.

LoKi takes care about the multiple counting within the loop over multiparticle combinations, e.g. for the following loop the given pair of two photons will appear only once:

```
1000Loop pi0 = loop( "gamma gamma" ) ;
1010while ( pi0 )
1020    {
1030       /* do something with the combination */
1040
1050       ++pi0 ; // go to the next valid combination
1060    }
1070
```

Internally LoKi eliminates such double counting through the discrimination of non-ordered combinations of the particles of the same type.

Access to the information inside *the multi-particle loops*

Inside the loop there are several ways to access the information about the properties of the combination.

Access to the daughter particles

For access to *the daughter particles* (*the selection components*) one could use following constructions:

```
1000for ( Loop D0 = loop( "K- pi+ pi+ pi-" ) ; D0 ; ++D0 )
1010    {
1020      const LHCb::Particle* kaon = D0(1) ; // the first daughter particle
1030      const LHCb::Particle* piP1 = D0(2) ; // the first positively charged pion
1040      const LHCb::Particle* piP2 = D0(3) ; // the second positively charged pion
1050      const LHCb::Particle* pim  = D0(4) ; // the fourth daughter particle
1060    }
1070
```

Please pay attention that ***the indices for daughter particles starts from 1***, because this is more consistent with actual notions *"the first daughter particle"*, *"the second daughter particle"*, etc. The index 0 is reserved for the whole combination. Alternatively one could use other functions with a bit more verbose semantics:

```
1000for ( Loop D0 = loop( "K- pi+ pi+ pi-" ) ; D0 ; ++D0 )
1010    {
1020      const LHCb::Particle* kaon = D0->daughter(1) ; // the first daughter
1030      const LHCb::Particle* piP1 = D0->daughter(2) ; // the second daughter
1040      const LHCb::Particle* piP2 = D0->child(3)    ; // the third daughter
1050      const LHCb::Particle* pim  = D0->particle(4) ; // the fourth daughter
1060    }
1070
```

Since the results of all these operations are raw C++ pointers to `LHCb::Particle=` objects, one could effectively reuse the *functions & cuts* for extraction the useful information:

```
1000for ( Loop D0 = loop( "K- pi+ pi+ pi-" ) ; D0 ; ++D0 )
1010    {
1020      const double PKaon = P ( D0(1) ) /GeV ; // Kaon momentum in GeV/c
1030      const double PTpm  = PT( D0(4) )      ; // Momentum of  "pi-"
1040    }
1050
```

Plenty of methods exist for evaluation of different kinematic quantities of different combinations of daughter particles:

```
1000for ( Loop D0 = loop( "K- pi+ pi+ pi-" ) ; D0 ; ++D0 )
1010    {
1020      const LoKi::LorentzVector v   = D0->p()  ; // 4 vector of the whole combination
1030      const LoKi::LorentzVector v0  = D0->p(0) ; // 4 vector of the whole combination
1040      const LoKi::LorentzVector v1  = D0->p(1)    ; // 4-vector of K-
1050      const LoKi::LorentzVector v14 = D0->p(1,4)  ; // 4-vector of K- and pi-
1060      const LoKi::LorentzVector v123 = D0->momentum(1,2,3)  ; // 4-vector of K- and pi+ and
1070      double m12  = D0->p(1,2).m()       ; // mass of K- and the first pi+
1080      double m234 = D0->p(2,3,4).m()     ; // mass of 3 pion sub-combination
1090      doule  m24  = D0->mass(2,4)        ; // mass of 2nd and 4th particles
1100      doule  m13  = D0->m(1,3)            ; // mass of 1st and 3rd particles
```

```
1110    }
1120
```

Alternatively to the convenient short-cut methods `Loop::p`, `Loop::m` one could use the equivalent methods `Loop::momentum` and `Loop::mass` respectively.

Access to *the mother particle* and its properties

Access to information on *the effective mother particle* of the combination requires the call of `loop` method to be supplied with the type of the *particle*. The information on the particle type can be introduced into the loop in the following different ways:

```
1000// particle name
1010for ( Loop D0 = loop ( "K- pi+ pi+ pi-", "D0" ) ; D0 ; ++D0 ) { ... }
1020
1030// particle ID
1040for ( Loop D0 = loop ( "K- pi+ pi+ pi-", 241  ) ; D0 ; ++D0 ) { ... }
1050
1060// through ParticleProperty object:
1070const ParticleProperty* pp = ... ;
1080for ( Loop D0 = loop ( "K- pi+ pi+ pi-", pp  ) ; D0 ; ++D0 ) { ... }
1090
1100// explicit set/reset
1110Loop D0 = loop ( "K- pi+ pi+ pi-" ) ;
1120
1130D0 -> setPID ( 241 )      ;  // set/reset the partcle ID
1140
1150// the same:
1160D0 -> setPID ( "D0" )     ;  // set/reset the partcle ID
1170
1180// the same:
1190D0 -> setPID ( pp )       ;  // set/reset the partcle ID
1200
1210// perform a loop:
1220for ( ; D0 ; ++D0 )  { ... }
1230
```

For properly instrumented looping construction one has an access to the information about *the effective mother particle of the combination*:

```
1000for ( Loop D0 = loop ( "K- pi+ pi+ pi-", "D0" ) ; D0 ; ++D0 )
1010    {
1020      const LHCb::Particle*   d0_1 = D0   ;
1030      const LHCb::Particle*   d0_2 = D0( 0 )  ;
1040      const LHCb::Particle*   d0_3 = D0->particle()  ;
1050      const LHCb::Particle*   d0_4 = D0->particle( 0 )  ;
1060      const LHCb::Vertex*     v_1  = D0   ;
1070      const LHCb::Vertex*     v_2  = D0->vertex() ;
1080    }
1090
```

The example above shows several alternative ways for accessing information on *"the effective particle"* and *"the effective vertex"* of the combination.

The existence of the implicit conversion of the looping construction to the types `const LHCb::Particle*` and `const LHCb::Vertex*` allows to apply all machinery of *Particle and Vertex functions and cuts* to the looping construction:

```
for ( Loop D0 = loop( "K- pi+ pi+ pi-", "D0" ) ; D0 ; ++D0 ) {
  double mass  = M( D0 ) / GeV ; // mass in GeV
  double chi2v = VCHI2( D0 )   ; // chi2 of vertex fit
  double pt    = PT ( D0 ) ;  // transverse momentum
  }
```

Saving of *the interesting combinations*

Every interesting combination of particles could be saved for future reuse in LoKi and/or DaVinci:

```
for ( Loop phi = loop( "kaon- kaon+" , "phi(1020)" ) ; phi ; ++phi )
  {
    if( M( phi )  < 1.050 * Gev ) { phi->save( "phi" ) ; }
  }
```

When particle is saved in internal LoKi storage, it is simultaneously saved into DaVinci's *Desktop Tool*. For each saved category of particles a new LoKi tag is assigned. In the above example the tag "phi" is assigned to all selected and saved combinations. One could reuse already existing tags to add the newly saved particles to existing LoKi containers/selections.

Patterns

Usage of kinematic constraints in LoKi

Access to Monte Carlo truth information

Monte Carlo truth matching

LoKi offers fast, easy, flexible and configurable access to Monte Carlo truth information. The helper utility `MCMatch` could be used to check if given reconstructed particle has the match with given Monte Carlo particle:

```
MCMatch mcmatch = mcTruth ("My MC-truth matcher") ;

const LHCb::MCParticle* MCD0 = ... ;

for ( Loop D0 = loop( "K- pi+", "D0" ) ; D0 ; ++D0 )
  {

    if ( mcmatch ( D0 , MCD0 ) )
      { plot ( M(D0) / GeV , "Mass of true D0 1 " , 1.5 , 2.0 ) ;}

    // the same as previous
    if ( mcmatch -> match ( D0 , MCD0 ) )
      { plot ( M(D0) / GeV, "Mass of true D0 2 " , 1.5 , 2.0 ) ; }

  }
```

The actual Monte Carlo matching procedure is described in detail here.

MCMatch object could be used for repetitive matching with sequences of arbitrary type of Monte Carlo and reconstructed particles:

```
1000// some 'sequence' or 'range' type
1010typedef std::vector<const LHCb::MCParticle*> MCSEQ   ;
1020
1030// some 'sequence' or 'range' type
1040typedef std::vector<const   LHCb::Particle*> RECOSEQ ;
1050
1060MCSEQ                  mcps = ... ;
1070RECOSEQ                  ps = ... ;
1080const LHCb::MCParticle*  mcp = ... ;
1090const   LHCb::Particle*    p = ... ;
1100
1110MCMatch mcmatch = mcTruth() ;
1120
1130/// return the iterator to the first matched RECO particle
1140RECOSEQ::const_iterator ip =
1150       mcmatch->match( ps.begin () ,   // begin of sequence of Particles
1160                       ps.end   () ,   // end   of sequence of Particles
1170                       mcp         ) ; // Monte Carlo particle
1180
1190/// return the iterator to the first  matched MC particle
1200MCSEQ::const_iterator imcp =
1210       mcmatch->match( p              , // reconstructed particle
1220                       mcps.begin ()  , // begin of MC sequence
1230                       mcps.end   ()  );// end   of MC sequence
1240
1250/// return true if *ALL* RECO particles are matched to MC
1260bool all = mcmatch->
1270        match(   ps.begin   () ,    // begin of sequence of reco particles
1280                 ps.end     () ,    // end of sequence o=f reco particles
1290                 mcps.begin () ,    // begin of MC sequence
1300                 mcps.end   () ) ;  // end   of MC sequence
1310
```

The methods described above are template, and therefore they could be applied to *any* type of sequence of pointers to LHCb::MCParticle and LHCb::Particle objects.

Of course in the spirit of LoKi is to provide the same functionality in a more useful and elegant way as ordinary *predicate* or *cut*:

```
1000const LHCb::MCParticle* MCD0 = ... ;
1010
1020// create the predicate:
1030Cut mc = MCTRUTH ( mcTruth () , MCD0 ) ;
1040
1050for ( Loop D0 = loop ( "K- pi+" ,  "D0" ) ; D0 ; ++D0 )
1060    {
1070
1080      // use it!
1090      if ( mc( D0 ) )   { plot ( "mass of true D0" ,  M ( D0 ) / GeV , 1.5 , 2.0 )  ; }
1100
1110    }
```

The latter way is especially convenient for analysis.

LoKi Reference Manual

See here

Commonly used LoKi::Hybrid::Filters

The list of commonly used `LoKi::Hybrid::Filters` could be inspected here.

-- Vanya Belyaev - 12 Jul 2007

-- Vanya BELYAEV - 17 May 2008

This topic: LHCb > LoKiLUG
Topic revision: r9 - 2008-05-17 - VanyaBelyaev