LHCb 2004-023
COMP-Offline
4 March 2004

LoKi

Smart & Friendly C++ Physics Analysis Toolkit

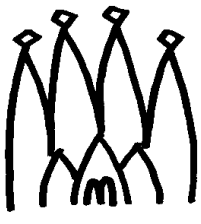
USER GIUDE AND REFERENCE MANUAL

last update: July 20, 2005

VERSION V3R13

`CVS tag $Name: v3r14 $`

Vanya Belyaev[1]
LAPP/Annecy & ITEP/Moscow



[1] E-mail: Ivan.Belyaev@lapp.in2p3.fr

Abstract

LOKI[2] is a package for the simple and user-friendly data analysis. LOKI is based on GAUDI architechture. The current functionality of LOKI includes the selection of particles, manipulaton with predefined kinematical expressions, loops over combinations of selected particles, creation of composite particles from various combinations of particles, flexible manipulation with various kinematical constrains, simplified population of histograms, N-Tuples, Event Tag Collections and access to Monte Carlo truth information.

[2] ***Loki*** is a god of wit and mischief in Norse mythology, could also be interpreted as *'**Lo**ops & **Ki**nematics'*

Table of Contents

Part I

User Guide

Chapter 1: Introduction

All off-line OO software for reconstruction [2], simulation [3], visualisation [4] and analysis [5], for LHCb [1] collaboration is based on GAUDI [6] framework. All software is written on C++, which is currently the best suited language for large scale OO software projects[1]. 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 enormouse code with complicated and probably non-obvious content:

```
1   ParticleVector::const_iterator im;
2   for ( im = vDsK.begin(); im != vDsK.end(); im++ ) {
3     if((*im)->particleID().pid() == m_DsPlusID ||
4     (*im)->particleID().pid() == m_DsMinusID) vDs.push_back(*im);
5     else if((*im)->particleID().pid() == m_KPlusID ||
6     (*im)->particleID().pid() == m_KMinusID) vK.push_back(*im);
7     else{
8       log <<MSG::ERROR<< " some message here "<<endreq;
9       return StatusCode::FAILURE;
10      }
11    }
```

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 by Hartwig Albrecht. KAL is an interpreter language written on Fortran[2]. 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 [8] and GCOMBINER [7] 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 [9] classes, like `HepChooser` and `HepCombiner`
- LOKI [10] 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,

[1] 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.

[2] It is cool, isn't it?

non-trivial and non-obvious recipes for efficient solving of major common tasks and problems.

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

```
HYPOTH  E+  MU+  PI+ 5  K+  PROTON

IDENT PI+      PI+
IDENT K+       K+
IDENT PROTON   PROTON
IDENT E+       E+
IDENT MU+      MU+

SELECT K- PI+
 IF P > 2 THEN
  SAVEFITM D0 DMASS 0.045 CHI2 16
 ENDIF
ENDSEL

SELECT D0 PI+
 PLOT MASS L 2.0 H 2.100 NB 100 TEXT ' Mass of D0 pi+ '
ENDSEL

GO 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 $D^0 \to K^-\pi^0$ decay is clear and unambigously visible between 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 [8] of $K^0_S \to \pi^+\pi^-$ reconstruction:

```
TrackPattern   piMinus = pi_minus.with ( pt > 0.1 & p > 1 );
TrackPattern   piPlus  = pi_plus.with  ( pt > 0.1 & p > 1 );
TwoProngDecay kShort   = K0S.decaysTo  ( PiPlus & PiMinus );
kShort.with( vz > 0   );
kShort.with( pt > 0.1 );
```

This code fragment is not so transparent as specialised 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:

```
select ( "Pi+" , ID =="pi+" && CL > 0.01  && P > 5 * GeV );
select ( "K-"  , ID =="K-"  && CL > 0.01  && P > 5 * GeV );
for ( Loop D0 = loop("K- pi+","D0") ; D0 ; ++D0 )
{
  if ( P( D0 ) > 10 * GeV ){ D0->save("D0"); }
}
for ( Loop Dstar = loop( "D0 Pi+" , "D*+" ) ; Dstar ; ++Dstar )
{
  plot ( M( Dstar ) / GeV , " Mass of D0 pi+ " , 2.0 , 2.1 , 100 );
}
```

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 obviouse that an essential emulation of KAL semantics is performed[3].

LOKI follows general GAUDI [6] architecture and indeed it is just a thin layer atop of tools, classes, methods and utilities from DAVINCI [5], DAVINCITOOLS, DAVINCIMCTOOLS and DAVINCIASSOCIATORS packages.

Since LOKI is just a thin layer, all DAVINCI tools are 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.

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.

[3] 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.

Chapter 2: LOKI ingredients

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

- selection/filtering with the criteria based on particle(s) kinematical, 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 kinematical 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 kinematical 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 5 major actions.

Chapter 3: Selection

3.1 Selection of Particles

LOKI allows to select/filter a subset of reconstructed particles which fulfils the chosen criteria, based on their kinematical, identification and topological properties and to refer later to this selected subset with defined tag:

```
1
2       select( "AllKaons" , abs( ID ) == 321 && CL > 0.05  && PT > 100 * MeV );
```

Here from all particles, loaded by DAVINCI, the subset of particles identified as charged kaons (**`abs(ID)==321`**) with confidence level in excess of 5% (**`CL>0.05`**) and 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`**, **`CL`** and **`PT`**, are predefined LOKI *variables* or *functions*[1] which allow to extract the identifier, confidence level and the transverse momentum for given particle. *Cuts* or *predicates* or selection criteria are constructed with comparison operations ('**`<`**', '**`<=`**', '**`==`**', '**`!=`**', '**`>=`**', '**`>`**') from *variables*. The arbitrary combinations of *cuts* with boolean operations (**`&&`** or **`||`**) are allowed to be used as selection criteria.

LOKI defines many frequently used *variables* and set of mathematical operation on them ('**`+`**', '**`-`**', '**`*`**', '**`/`**' and all elementary functions, like '**`sin`**', '**`cos`**', '**`log`**' etc), which could be used for construction of *variables* of arbitrary complexity. *Cuts* and *variables* are discussed in detail in chapters 13 and 14.

Indeed the function **`select`** has a return value of type `Range`, which is essentially the light-weight container of selected particles. This return 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:

```
1
2       Range kaons =
3        select( "AllKaons" , abs( ID ) == 321 && CL > 0.05  && PT > 100 * MeV );
4       select( "kaon+" , kaons , Q >  0.5 ) ;
5       select( "kaon-" , kaons , Q < -0.5 ) ;
```

Here all positive kaons are selected into the subset named **`"kaon+"`** and all negative kaons 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/GAUDI containers of particles **`ParticleVector`** and **`Particles`**

```
1
2       ParticleVector particles = ... ;
3       Range kaons_1 =  select( "Kaons_1" , particles , abs( ID ) ) ;
4
5       Particles* event = get<Particles>( "Phys/Prod/Partciles" ) ;
6       Range kaons_2 =  select( "Kaons_2" , even t    , abs( ID ) ) ;
```

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

[1] Indeed they are *function objects*, or *functors* in C++ terminology

```
1
2       /// SEQUENCE is an arbitrary sequence of objects,
3       /// implicitely convertible to
4       /// type Particle*. e.q. std::vector<Particle*>,
5       /// ParticleVector, Particles, std::set<Particle*> etc.
6       SEQUENCE particles = ... ;
7       Range kaons =
8             select( "AllKaons"          ,    // 'tag'
9                     particles.begin () ,    // begin of sequence
10                    particles.end   () ,    // end of sequence
11                    abs( ID ) == 321   ) ; // cut
```

The output of selection could be directly inspected through explicit loop over the content of the selected container:

```
1
2       Range kaons =
3       select( "AllKaons" , abs( ID ) == 321 && CL > 0.05  && PT > 100 * MeV );
4       for( Range::iterator kaon = kaons.begin() ; kaons.end() != kaon ; ++kaon )
5         {
6           const Particle* k = *kaon ;
7           /* do something with this raw C++ pointer */
8         }
```

3.2 Selection of Vertices

Similar approach is used for selection/filtering of vertices:

```
1
2       VRange vs  = vselect( "GoodPVs" ,
3       VTYPE == Vertex::Primary && 5 < VTRACKS && VCHI2 / VDOF < 10 );
```

Here from all vertices loaded by DAVINCI one selects only vertices tagged as "Primary Vertex" (**`VTYPE==Vertex::Primary`**) and constructed from more that 5 tracks (**`5<VTRACKS`**) and with χ^2/nDoF less than 10 (**`VCHI2/VDOF<10`**). This set of selected vertices is named as **`"GoodPVs"`**.

Again **`VTYPE`**, **`VTRACKS`**, **`VCHI2`**, and **`VDOF`** are predefined LOKI *functions*/*variables*. It is internal convention of LOKI that all predefined *functions*, types and methods for vertices starts their names from capital letter "**`V`**". As an example one see type **`VRange`** for a container of vertices, 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 (type **`VRange`**), *standard* DAVINCI/GAUDI containers **`VertexVector`** and **`Vertices`** and from arbitary sequence.

```
1
2
3       VRange vertices_1 = ... ;
4       VRange vs1 =
5         vselect(  "GoodPVs1"  ,                    // 'tag'
6                   vertices_1  ,                    // input vertices
7                 VTYPE == Vertex::Primary && 5 < VTRACKS  ); // cut
8
9
10      VertexVector vertices_2 = ... ;
11      VRange vs2 =
12        vselect(  "GoodPVs2"  ,                    // 'tag'
13                  vertices_2  ,                    // input vertices
14                VTYPE == Vertex::Primary && 5 < VTRACKS  ); // cut
```

```
15
16      Vertices* vvv = get( eventSvc() , "Phys/Prod/Vertices" , vvv );
17      Vrange v3 =
18        vseelct(  "GoodPVs3"  ,                  // tag
19                  vvv       ,                   // input vertices
20                 VTYPE == Vertex::Primary && 5 < VTRACKS  ); // cut
21
22       /// arbitrary sequence of ojbects, implicitely convertible to
23       /// type Vertex*, e.g. VertexVector, std::vector<Vertex*>,
24       /// std::set<Vertex*> etc ....
25       SEQUENCE vertices_4 = ... ;
26       VRange vs4 =
27         vselect(  "GoodPVs4"          ,       // 'tag'
28                   vertices_3.begin () ,       // begin of input sequence
29                   vertices_2.end   () ,       // end of input sequence
30                  VTYPE == Vertex::Primary && 5 < VTRACKS  ); // cut
```

3.3 Selection of only one candidate

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

```
1
2       VRange vrtxs           = vselect( "GoodPVs" , VTYPE == Vertex::Primary );
3       const  Vertex* vertex  = select_max( vrtxs , VTRACKS ) ;
```

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

```
1
2       VRange vrtxs           = vselect( "GoodPVs" , VTYPE == Vertex::Primary );
3
4       // get B-candidate
5       const Particle* B = ...
6
7       // find the primatry vertex with minimal B-impact parameter
8       const Vertex* vertex  = select_min( vrtxs , VIP( B , geo() ) ) ;
```

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 method **`select_max`** and its partner **`select_min`** are type-blind and they could be applied to the containers of particles:

```
1
2       Range kaons  = select( .... );
3       const Particle* kaon  = select_min( kaons  , abs( PY ) + sin( PX )  ) ;
```

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

Chapter 4: Loops

4.1 Looping over reconstructed particles

4.1.1 Simple loops

Already shown above is how to perform simple looping over the selected range of particles

```
1
2       Range kaons =
3       select( "AllKaons" , abs( ID ) == 321 && CL > 0.05  && PT > 100 * MeV );
4       for( Range::iterator kaon = kaons.begin() ; kaons.end() != kaon ; ++kaon )
5       {
6         const Particle* k = *kaon ;
7         /* do something with this raw C++ pointer */
8       }
```

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

```
1
2       Range kaons =
3       select( "AllKaons" , abs( ID ) == 321 && CL > 0.05  && PT > 100 * MeV );
4       for( unsigned int index = 0 ; index < kaons.size() ; ++index )
5       {
6         const Particle* k1 = kaons     ( index ) ;
7         const Particle* k2 = kaons     [ index ] ;
8         const Particle* k3 = kaons .at ( index ) ;
9       }
```

The result of operators are not defined for invalid index, and `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.

4.1.2 Loops over 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`**-loop:

```
1
2       for( Loop phi = loop( "kaon- kaon+" ) ; phi ; ++phi )
3         {
4           /* do something with the combination */
5         }
```

The **`while`**-form of the loop is also supported:

```
1
2       Loop phi = loop( "kaon- kaon+" ) ;
3       while( phi )
4         {
5           /* do something with the combination */
6
7           ++phi ; // go to the next valid combination
8         }
```

The parameter of **`loop`** function is a 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.

```
1
2       Loop pi0 = loop( "gamma gamma" ) ;
3       while( pi0 )
4        {
5           /* do something with the combination */
6
7           ++pi0 ; // go to the next valid combination
8        }
```

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

4.1.3 Loops over charge conjugated combinations

It is quite often one needs to perform the looping over charge conjugates stateses as well:

```
1
2       for( Loop DP = loop( "K- pi+ pi+" , "D+") ; DP ; ++DP )
3       {
4       }
5       for( Loop DM = loop( "K+ pi- pi-" , "D-") ; DM ; ++DM )
6       {
7       }
```

These two loops can be combined into one loop using helper objects `CC` and `LoopCC`:

```
1
2       CC ccK  ( "K-"  , "K+"  ) ;
3       CC ccPi ( "pi+" , "pi+" ) ;
4       CC ccD  ( "D+"  , "D-"  ) ;
5       for( LoopCC D = loop( ccK + ccPi + ccPi , ccD ) ; D ; ++D )
6       {
7       }
```

4.2 Access to the information inside the loops

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

4.2.1 Access to daughter particles and their properties

For access to the daughter particles / selection components one could use following constructions:

```
1
2       for( Loop D0 = loop( "K- pi+ pi+ pi-" ) ; D0 ; ++D0 )
3        {
4          const Particle* kaon = D0(1) ; // the first daughter particle
5          const Particle* piP1 = D0(2) ; // the first positively charged pion
6          const Particle* piP2 = D0(3) ; // the second positively charged pion
7          const Particle* pim  = D0(4) ; // the fourth daughter particle
8        }
```

Please pay attention that 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 more verbose semantics:

```
1
2       for( Loop D0 = loop( "K- pi+ pi+ pi-" ) ; D0 ; ++D0 )
3        {
4          const Particle* kaon = D0->daughter(1) ;
5          const Particle* piP1 = D0->daughter(2) ;
6          const Particle* piP2 = D0->child(3) ;
7          const Particle* pim  = D0->particle(4) ;
8        }
```

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

```
1
2       for( Loop D0 = loop( "K- pi+ pi+ pi-" ) ; D0 ; ++D0 ) {
3         double PKaon = P ( D0(1) ) /GeV ; // Kaon momentum in GeV/c
4         double CLpp  = CL( D0(2) )      ; // confidence level of the  first "pi+"
5         double PTpm  = PT( D0(4) )      ; // Momentum of  "pi-"
6         }
```

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

```
1
2       for( Loop D0 = loop( "K- pi+ pi+ pi-" ) ; D0 ; ++D0 ) {
3         const HepLorentzVector v   = D0->p()  ; // 4 vector of the whole combination
4         const HepLorentzVector v0  = D0->p(0) ; // 4 vector of the whole combination
5         const HepLorentzVector v1  = D0->p(1)   ; // 4-vector of K-
6         const HepLorentzVector v14 = D0->p(1,4)  ; // 4-vector of K- and pi-
7         double m12  = D0->p(1,2).m()      ; // mass of K- and the first pi+
8         double m13  = D0->p(1,3).m()      ; // mass of K- and the second pi+
9         double m234 = D0->p(2,3,4).m()    ; // mass of 3 pion sub-combination
10        doule  m24  = D0->mass(2,4)       ; // mass of 2nd and 4th particles
11        }
```

Alternatively to the convenient short-cut method **`p`** one could use the equivalent method **`momentum`**.

4.2.2 Access to 'mother' particles of the loop and its properties

Access to information on the efefctive *"mother particle"* of the combination requires the call of **`loop`** method to be supplied with the type of the particle[1]. The information on the particle type can be introduced into the loop in the following different ways:

```
1
2       // particle name
3       for( Loop D0 = loop( "K- pi+ pi+ pi-", "D0" ) ; D0 ; ++D0 ) {}
4       // particle ID
5       for( Loop D0 = loop( "K- pi+ pi+ pi-", 241  ) ; D0 ; ++D0 ) {}
6       //
7       Loop D0 = loop( "K- pi+ pi+ pi-" ) ;
8       D0->setPID( 241 )       ;  // set/reset the partcle ID later
9       for( ; D0 ; ++D0 ) {}
```

For creation of specific particles one needs to supply the looping construction with information on various vertex, mass and/or direction constrains to be applied to the combination of particles. This is discussed in detail later in this document.

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

[1] This sad limitation comes from underlying DAVINCI tools, which e.g. for vertex fit perform checking to the particle nominal lifetime

```
1
2       for( Loop D0 = loop( "K- pi+ pi+ pi-", "D0" ) ; D0 ; ++D0 ) {
3         const Particle* d0_1 = D0    ;
4         const Particle* d0_1 = D0( 0 )  ;
5         const Particle* d0_1 = D0->particle()  ;
6         const Particle* d0_1 = D0->particle( 0 )  ;
7         const Vertex*   v_1  = D0    ;
8         const Vertex*   v_2  = D0->vertex() ;
9         }
```

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

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

```
1
2       for( Loop D0 = loop( "K- pi+ pi+ pi-", "D0" ) ; D0 ; ++D0 ) {
3         double mass  = M( D0 ) / GeV ; // mass in GeV
4         double chi2v = VCHI2( D0 )   ; // chi2 of vertex fit
5         double pt    = PT ( D0 ) ;  // transverse momentum
6         }
```

4.3 Saving of "interesting" combination

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

```
1
2       for( Loop phi = loop( "kaon- kaon+" , "phi(1020)" ) ; phi ; ++phi )
3       {
4         if( M( phi )  < 1.050 * Gev ) { phi->save( "phi" ) ; }
5       }
```

When particle is saved in internal *LoKi* storage, it is simultaneously saved into DAVINCI *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.

4.4 Patterns

The following code fragment seems to be not uncommon:

```
1
2       // some particle cuts
3       Cut  cuts  = ... ;
4       // some vertex cuts
5       VCut vcuts = ... ;
6       for( Loop phi = loop( "kaon- kaon+" , "phi(1020)" ) ; phi ; ++phi )
7       {
8         if( cuts( phi ) && vcuts( phi ) ) { phi->save( "phi" ) ; }
9       };
10      // extract all saved phis
11      Range phis = selected(``phi'');
```

The LOKI offers convenient short-cut for such code fragment:

```
1
2       // some particle cuts
3       Cut  cuts  = ... ;
4       // some vertex cuts
5       VCut vcuts = ... ;
6       Range phis = pattern( "phi", "kaon+ kaon-" , "phi(1020)" , cuts , vcuts );
```

4.5 Kinematical constraints

When LOKI creates a composite particle, it performs a vertex fit to find a common vertex for the daughter particles. The vertex fit is performed through standard abstract interface **`IVertexFitter`** from DAVINCITOOLS package:

```
1
2       // default fit strategy - vertex constrain fit
3       for( Loop dstar = loop( "D0 pi+" , 413 , FitVertex ) ; dstar ; ++dstar )
4       {
5         if( M( dstar )  < 2020 * MeV ) { dstar->save( "D*+" ) ; }
6       }
```

This default option could be switched off by supplying LOKI with appropriate (empty) **`FitStrategy`** object:

```
1
2       /// no kinematical constraint !
3       for( Loop dstar = loop( "D0 gamma" , 413 , FitNone ) ; dstar ; ++dstar )
4       {
5         if( M( dstar )  < 2020 * MeV ) { dstar->save( "D*0" ) ; }
6       }
```

Optionally a mass constrained vertex fit and/or direction fit could be applied to a composite particle. The fits are performed through their standard abstract interfaces **`IMassVertexFitter`** and **`IDirectionFitter`** from DAVINCITOOLS package:

```
1
2       /// default fit strategy - vertex constrained fit
3       for( Loop dstar = loop( "D0 pi+" , 413 ) ; dstar ; ++dstar )
4       {
5         StatusCode sc = dstar->fit( FitMassVertex ) ;
6         if( sc.isFailure() ) { continue ;}
7         if( M( dstar )  < 2020 * MeV ) { phi->save( "D*+" ) ; }
8       }
```

The different fit policies could be easily combined:

```
1
2       /// default fit strategy - vertex constrained fit
3       for( Loop dstar = loop( "D0 pi+" , 413 ) ; dstar ; ++dstar )
4       {
5         StatusCode sc = dstar->fit( FitMassVertex + FitDirection ) ;
6         if( sc.isFailure() ) { continue ;}
7         if( M( dstar )  < 2020 * MeV ) { phi->save( "D*+" ) ; }
8       }
```

The boolean operators **&&** and **||** as well as "numerical" operators **+**, and ***** could be used for building a complex fit policies from basic policies:

- **`FitNone`** : no kinematical constrains are applied
- **`FitVertex`** : vertex constrained fit is performed

- **`FitMass`** : mass constrained fit is performed[2]
- **`FitMassVertex`** : mass constrained vertex fit is performed
- **`FitDirection`** : direction constrained fit is performed. **`Loop`** object need to be explicitely supplied with additional **`Vertex`** object.
- **`FitLifeTime`** : *lifetime fit* is performed. **`Loop`** object need to be explicitely supplied with additional **`Vertex`** object.

Indeefd *lifetime fit* do not change the pacrticle parameters, it evaluates the additional parameters:

```
1
2       /// default fit strategy - vertex constrained fit
3       for( Loop bplus = loop( "D0 pi+" , 521 ) ; bplus ; ++bplus )
4       {
5         dstar->setPV( .... ) ; // set the  primary vertex (needed foe fits)
6         StatusCode sc = bplus -> fit ( FitMassVertex + FitLifetime) ;
7         if( sc.isFailure() ) { continue ;}
8         const double lifetime      = dstar->lifeTime      () ;
9         const double lifetimeError = dstar->lifeTimeError () ;
10        const double lifetimeChi2  = dstar->lifeTimeChi2  () ;
11      };
```

[2] not yet implemented

Chapter 5: Histograms & N-Tuples

5.1 Histograms

GAUDI [6] framework provides access to the histogramming facilities through their AIDA [11] abstract interfaces. This powerful and flexible way of making the histogramming has one important disadvantage. The *"locality"* of the histogramming facilities is lost. For typical GAUDI *algorithm*, one needs to declare the local variables for pointers to AIDA histograms in the body of the *algorithm* (which is usually resides inside separate file), and afterwords to book the histograms with the help of `IHistogramSvc`. Usually this operation is performed inside the **`initialize`** method of the *algorithm*, the actual histogram filling is performed inside **`execute`** method Usually the modification of different files and different code fragments is necessary to deal with histograms. Thus standard GAUDI histogramming facilities become *"non-local"*.

LOKI provides *"local"* approach to access the histogram facilities:

```
1
2       for( Loop D0 = loop( "K- pi+ pi+ pi-", "D0" ) ; D0 ; ++D0 )
3         {
4           double mass  = M( D0 ) / GeV ; // mass in GeV
5           plot ( mass , "D0 mass in GeV" , 1.7 , 3.0 , 150 ) ;
6         }
```

Here the histogram with title **`"D0 mass in GeV"`** will be booked with low limit **`1.7`**, high limit **`3.0`** and number of bins equal to **`150`**. The histogram will be booked "on-demand" only. Thus for empty loops the histograms will not be booked. The title of the histogram must be unique within the algorithm. The histogram gets the sequential number within the given algorithm[1]. If the proposed sequential number is already in use, the sequential search will be performed.

There are also shortcuts for selection containers

```
1
2     Range  kaons =  select ( ... ) ;
3     plot ( PT/GeV , kaons.begin() , kaons.end()  , "Kaon transverse momentum" , 0 , 5 ) ;
4
5     VRange prims = vselect ( ... ) ;
6     plot ( VZ/mm , prims.begin() , prims.end()  , "VZ of primary vertex" , -10 , 10 , 200 ) ;
```

The filling of histograms from arbitrary sequences of arbitrary objects through templated implicit loop is possible:

```
1
2      std::vector<double> v = ... ;
3      // fill a histogram with sine of the vector elements
4      plot( ::sin , v.begin() , v.end() , "sin" , -1. , 1 ) ;
5      // fill a histogram with sine using the weight = abs
6      plot( ::sin , v.begin() , v.end() , "sin abs " , -1. , 1 , 100 , ::abs) ;
7
8      Range  kaons =  select ( ... ) ;
9      plot ( PT/GeV , kaons.begin() , kaons.end() , "Kaon PT" ,  0 ,  5 ,  50 ) ;
```

One can force the assignement of given histogram identifier:

[1] One could apply the overall offset in the numbering of the histograms

```
1
2       double mass = ... ;
3       plot( mass , 100 , "D0 mass in GeV" , mass , 1.7 , 3.0 , 150 ) ;
4
5      Range  kaons =  select ( ... ) ;
6      plot ( PT/GeV , kaons.begin() , kaon.end() , 103 , "Kaon transverse momentum" , 0 , 5 ) ;
```

5.2 N-Tuples

For typical GAUDI *algorithm* one needs to declare *all* N-Tuple columns inside *algorithm* body in the header file of the *algorithm*, then book the N-Tuple itself and add *all* previously declared columns inside **`initialize`** method of the *algorithm* and only then fill columns and write N-Tuple record. This is *"non-local"* and very verbose approach.

5.2.1 Simple columns

LOKI suggests few *"local"* techniques to deal with N-Tuples. Booking and filling the N-Tuple and its columns is performed locally. The most conservative traditional technique is illustrated by following example:

```
1
2       Tuple  tuple = nTuple("My N-Tuple");
3       for( Loop D0 = loop( "K- pi+", "D0" ) ; D0 ; ++D0 )
4       {
5         tuple->column("mass", M(D0)/GeV  ) ;
6         tuple->column("p"   , P(D0)/GeV  ) ;
7         tuple->column("pt"  , PT(D0)/GeV ) ;
8         tuple->write();
9       }
```

In this case the N-Tuple with title **`"My N-Tuple"`** will be booked. The title of the N-tuple must be unique within given user *algorithm*. Later, "on-demand" basis, three N-Tuple columns named **`"mass"`**, **`"p"`** and **`"pt"`** will be booked and added to the N-Tuple. One could book, commit and fill several columns to the N-Tuple simultaneously[2]:

```
1
2       Tuple  tuple = nTuple("My N-Tuple");
3       for( Loop D0 = loop( "K- pi+", "D0" ) ; D0 ; ++D0 )
4       {
5         tuple->fill( "mass,p,pt" , M(D0)/GeV , P(D0)/GeV , PT(D0)/GeV ) ;
6         tuple->write();
7       }
```

The first argument of **`fill`** method is blank or comma separated list of column names. The variable number of arguments of type `double` could not be less than number of items in this list. Both methods (**`column`** and **`fill`**) can be combined in an arbitrary way:

```
1
2       Tuple  tuple = nTuple("My N-Tuple");
3       for( Loop D0 = loop( "K- pi+", "D0" ) ; D0 ; ++D0 )
4       {
5         tuple->fill( "mass,p" , M(D0)/GeV , P(D0)/GeV ) ;
6         tuple->colume("pt"  , PT(D0)/GeV ) ;
7         tuple->write();
8       }
```

[2] In this case all of them **must** be of type **`double`**

It is sometimes desirable to put three components of space vector or four components of Lorentz vector into N-tuple. LOKI offers the simplified approach to fill N-tuple with such objects:

```
1
2       Tuple  tuple = nTuple("My N-Tuple");
3       for( Loop D0 = loop( "K- pi+", "D0" ) ; D0 ; ++D0 )
4       {
5         // put 4-vector into N-tuple
6         tuple->column("D0p"  , D0->p()                    ) ;
7         // put 3D-vector into N-tuple
8         tuple->column("D0v"  , D0->vertex()->position() ) ;
9         tuple->write();
10      }
```

The usage of tedious `write` method could be eliminated with usage of local automatic object of type `Record`:

```
1
2       Tuple  tuple = nTuple("My N-Tuple");
3       for( Loop D0 = loop( "K- pi+", "D0" ) ; D0 ; ++D0 )
4       {
5         Record rec(tuple,"mass,p,pt" , M(D0)/GeV , P(D0)/GeV , PT(D0)/GeV ) ;
6       }
```

This is the most compact way of dealing with N-Tuples[3].

The alternative tecknique to "feed" the N-Tuple with data is the utility **`Column`**

```
1
2       Tuple  tuple = nTuple("My N-Tuple");
3       const EventHeader* header =
4          get<EventHeader>( EventHeaderLocation ::Default ) ;
5       const L0DUReport*  l0     =
6          get<L0DUReport> ( L0DUReportLocation  ::Default ) ;
7       const L1Report*    l1     =
8          get<L1Report>   ( L1ReportLocation    ::Default ) ;
9       const HepLorenzVector& v1   = ... ;
10      const IOpaqueAddress*  addr = ... ;
11      const HepPoin3D&       p3   = ... ;
12
13      // 'feed' n-Tuple with columns
14      tuple << Column( ""          , header )
15            << Column( ""          , l0     )
16            << Column( ""          , l1     )
17            << Column( "v1"        , v1     )
18            << Column( "Address"   , addr   )
19            << Column( "p3"        , p3     )
20            << Column( "mc"        , false  )
21            << Column( "one"       , 1.0    ) ;
22
23     tuple->write() ;
```

5.2.2 Array-like columns

Current version of LOKI provides users with the possibility to book and fill array-like N-tuple items.

```
1
2       std::vector<double> pts    ;
3       std::vector<float>  masses ;
4       std::vector<double> moms   ;
5
```

[3] And again it is some kind of emulation of the semantics of KAL language

```
6       for( Loop D0 = loop( "K- pi+", "D0" ) ; D0 ; ++D0 )
7       {
8         pts    .push_back( PT ( D0 ) / GeV ) ;
9         moms   .push_back( P  ( D0 ) / GeV ) ;
10        masses .push_back( M  ( D0 ) / GeV ) ;
11      }
12
13      /// get the tuple (book on demand)
14      Tuple  tuple = nTuple("My N-Tuple");
15
16      /// put an array to the tuple
17      tuple -> farray( "pt"           ,   /// item name
18                       pts.begin    () ,   /// begin of data sequence
19                       pts.end      () ,   /// end of data sequence
20                       "num"          ,   /// name of index item
21                       1000           ) ; /// maximal array size
22
23      /// put an array to the tuple
24      tuple -> farray( "p"            ,   /// item name
25                       moms.begin   () ,   /// begin of data sequence
26                       moms.end     () ,   /// end of data sequence
27                       "num"          ,   /// name of index item
28                       1000           ) ; /// maximal array size
29
30      /// put an array to the tuple
31      tuple -> farray( "mass"         ,   /// item name
32                       masses.begin () ,   /// begin of data sequence
33                       masses.end   () ,   /// end of data sequence
34                       "num"          ,   /// name of index item
35                       1000           ) ; /// maximal array size
36
37      /// write tuple
38      tuple -> write() ;
```

For this example four columns will be booked for the N-tuple **`"num"`**, **`"pt"`** , **`"p"`** and **`"mass"`** , the first one will contains the number of entries in variable size columns.

5.2.3 Advanced array-like columns

The modification of `"farray"` method can be applied to arbitrary sequences of objects of arbitrary types with arbitrary functions. There exist only two slight limitations: the function must accept as an argument the type from the sequence and the result of function must be convertible to type `"float"`:

```
1
2       Range gammas   = select( "gamma" , PT > 100 * MeV && ID == 22  ) ;
3       Range piplus   = select( "pi+"   , PT > 100 * MeV && ID == 121 ) ;
4       Range piminus  = select( "pi-"   , PT > 100 * MeV && ID == 211 ) ;
5
6       /// get N-Tuple (book on demand)
7       Tuple tuple = nTuple( "My Ntuple" );
8
9       /// put the transverse momentum of all photons into N-Tuple
10      tuple->farray( "gpt"           ,   // data item name
11                     PT              ,   // function object
12                     gammas.begin () ,   // begin of data sequence
13                     gammas.end   () ,   // end of data sequence
14                    "ng"             ,   // name of "length" tuple item
15                     500             ) ; // maximal array length
16
17      /// put the momentum of positive pions into N-Tuple
```

```
18      tuple->farray( "pplus"          ,    // data item name
19                     P                ,    // function object
20                     piplus.begin () ,    // begin of data sequence
21                     piplus.end   () ,    // end of data sequence
22                    "npip"           ,    // name of "length" tuple item
23                     200              ) ; // maximal array length
24
25      /// put the the whole momentum of negative pions into N-Tuple
26      tuple->farray(  "px"             ,    // data item name
27                      PX               ,    // function object
28                     "py"              ,    // data item name
29                      PY               ,    // function object
30                     "pz"              ,    // data item name
31                      PZ               ,    // function object
32                     "pt"              ,    // data item name
33                      PT               ,    // function object
34                     piminus.begin () ,    // begin of data sequence
35                     piminus.end   () ,    // end of data sequence
36                    "npim"           ,    // name of "length" tuple item
37                     200              ) ; // maximal array length
38
39
40      /// add some sensless data to the tuple
41      std::vector<double> data = ... ;
42      tuple->farray( "sqrtfun"       ,    // data item name
43                     sqrt            ,    // function
44                    "sinfun"         ,    // data item name
45                     sin             ,    // function
46                    "cosfun"         ,    // data item name
47                     cos             ,    // function
48                     data.begin () ,    // begin of data sequence
49                      data.end   () ,    // end of data sequence
50                     "len"           ,    // data length
51                      1000           ) ; // maximal array size
52
53      /// write tuple
54      tuple -> write () ;
```

Again the usage of **`"farray"`** method can in a trivial way be combined with other methods.

5.3 Event Tag Collections

LOKI offers easy possibility to access the *Event Tag Collection* ntuples. One books the event tag collection N-Tuple in exactly the same way as ordinary N-Tuple:

```
1
2       Tuple  tuple = evtCol("My Event Collection");
3
4       /// fill tuple with some information
5
6       ....
7
8       /// add the event address to event collection tuple
9       DataObject* event = get<DataObject>( "/Event" ) ;
10
11      /// put Event's IOpaqueAddress into N-Tuple
12      tuple->column( "Address" ,
13                     event->registry()->address() ) ;
14
15      /// mark the event if event should be written
16      setFilterPassed( true );
17
```

```
18      // write N-tuple
19      tuple->write();
```

Indeed **`write`** is explicitely disabled for *Event Tag Collections*. One need explicitely run the appropriate instance of `EvtCollectionStream` algorithm from GAUDISVC package to write the event tag collection.

Chapter 6: Access to MC truth information

6.1 Match to MC truth

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

```
1
2       MCMatch mcmatch = mctruth() ;
3       const MCParticle* MCD0 = ... ;
4       for( Loop D0 = loop( "K- pi+", "D0" ) ; D0 ; ++D0 )
5       {
6         if( mcmatch( D0 , MCD0 ) )
7         { plot( M(D0)/GeV , "Mass of true D0 1 " , 1.5 , 2.0 ) ;}
8         // the same as previous
9         if( mcmatch->match( D0 , MCD0 ) )
10        { plot( M(D0)/GeV, "Mass of true D0 2 " , 1.5 , 2.0 ) ;}
11      }
```

The actual MC matching procedure is descrived in detail in section 15.

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

```
1
2       // or any other 'sequence' or 'range' type
3       typedef std::vector<const MCParticle*> MCSEQ   ;
4       // or any other 'sequence' or 'range' type
5       typedef std::vector<const   Particle*> RECOSEQ ;
6
7       MCSEQ              mcps = ... ;
8       RECOSEQ              ps = ... ;
9       const MCParticle*  mcp = ... ;
10      const   Particle*    p = ... ;
11
12      MCMatch mcmatch = mctruth() ;
13
14      /// return the iterator to the first matched RECO particle
15      RECOSEQ::const_iterator ip =
16         mcmatch->match( ps.begin () ,   // begin of sequence of Particles
17                         ps.end   () ,   // end   of sequence of Particles
18                         mcp         ) ; // Monte Carlo particle
19
20      /// return the iterator to the first  matched MC particle
21      MCSEQ::const_iterator imcp =
22         mcmatch->match( p              , // reconstructed particle
23                         mcps.begin ()  , // begin of MC sequence
24                         mcps.end   ()  ); // end   of MC sequence
25
26      /// return true if *ALL* RECO particles are matched to MC
27      bool all = mcmatch->
28          match(   ps.begin   () ,    // begin of sequence of reco particles
29                   ps.end     () ,    // end of sequence o=f reco particles
30                   mcps.begin () ,    // begin of MC sequence
31                   mcps.end   () ) ;  // end   of MC sequence
```

The methods described above are templated, and thefore they could be applied to ***any*** type of sequence of pointers to **`MCParticle`** and **`Particle`** objects, e.g. they are applicable to standard **`MCParticles`** and **`Particles`** and **`ParticleVector`** types. Also the types

`LoKi::Algo::Range` and `LoKi::Algo::MCRange`, could be used for Monte Carlo matching since they both follow the standard sequence interface.

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*:

```
1
2       const MCParticle* MCD0 = ... ;
3
4       Cut mc = MCTRUTH( mctruth() , MCD0 );
5       for( Loop D0 = loop( "K- pi+", "D0" ) ; D0 ; ++D0 )
6       {
7         if( mc( D0 ) )
8         { plot( M(D0)/GeV ,  "mass of true D0" , M(D0)/GeV, 1.5 , 2.0 ) ;}
9       }
```

The latter way is especially convenient for analysis.

6.2 Easy access to decay trees

DAVINCIMCTOOLS package provides the physicist with the tool `MCDecayFinder` by Olivier Dormond[1]. The tool implements **`IMCDecayFinder`** abstract interface and it is indeed very easy to configure and use it.

LOKI provides a tiny layer to make the usage of this tool even more easy through increased locality of the algorithm:

```
1
2       MCMatch finder = mctruth() ;
3       MCRange mcd0s  = finder->findDecays( "D0 -> K- pi+" ) ;
```

The result of the search is stored into internal LOKI storage and the return value is the range of found Monte Carlo decay trees.

The combination of both functionalities of **`MCMatch`** objects, Monte Carlo matching and finding of Monte Carlo decays, provides physisist with the easy way of matching with Monte Carlo truth information:

```
1
2       MCMatch finder = mctruth() ;
3       // find interesting MC decay trees
4       MCRange mcd0s  = finder->findDecays(``D0 -> K- pi+'');
5       // use them to create the ``MC match cut'' :
6       Cut mc = MCTRUTH( mctruth() , mcd0s  );
7       for( Loop D0 = loop( "K- pi+", "D0" ) ; D0 ; ++D0 )
8       {
9         if( mc( D0 ) )
10        { plot( "mass of true D0" , M(D0)/GeV, 1.5 , 2.0 , 100 ) ;}
11      }
```

The selection of specific particles from decay trees is done in the similar way by using `^` and `:` symbols:

```
1
2       MCMatch finder = mctruth() ;
3
4       // find kaons from D0->K- pi+ decays :
5       MCRange k1  = finder->findDecays(  "D0 -> ^K- pi+ ");
6
```

[1] I consider this tool as a real pearl in DAVINCI

```
7       // find gamma from Bs -> phi gamma
8       MCRange k2  = finder->findDecays(  "Bs0 -> phi(1020) gamma");
9
10      // find all positive pions form B -> K 3pi decay
11      // attention: secondary piond from K- interactions will
12      // be also accepted!
13      MCRange k2  = finder->findDecays(  "pi+ pi+ D0 -> K- pi+ pi+ pi-");
```

6.3 Selection of Monte Carlo particles and vertices

The selection of Monte Carlo particels is illustarted by the listing:

```
1
2       // select all MC phi
3       MCRange phis    = mcselect (  "MCphi"  , MCID == "phi(1020)"  );
4
5       // select all MC muons
6       MCRange muons    = mcselect (  "MCmuon"   , MCABSID == 13   );
7       // select all MC muons with large PT
8       MCRange muons1   = mcselect (  "MCmuonPT" , MCABSID == 13 && MCPT > 1 * GeV   );
9       // from selected muons subselect positive
10      MCRange mu1      = mcselect (  "MCmuon1" , MC3Q > 0   ); // MCQ > 0
11
12      // select all MC beautry particles
13      MCRange beauty   = mcselect (  "beauty" , BEAUTY   );
14
15      // select all MC charm particles
16      MCRange charm    = mcselect (  "charm"  , CHARM   );
```

The selected particles can be extracted using the similar semantics:

```
1
2       // get selected phis
3       MCRange phis     = mcselected (  "MCphi"  );
4
5       // get selected muons MC muons
6       MCRange muons    = mcselected (  "MCmuon" );
7
8       // get the selected MC charm particles
9       MCRange charm    = mcselected (  "charm"  );
```

Chapter 7: Other useful utilities

7.1 Association of B-candidates with the primary vertices

Several physical algorithms are used to associate selected B-candidate with the primary vertex (e.g. the first primary vertex, the primary vertex with maximal track multiplicity, primary vertex with minimal value of B-candidate impact parameter, the primary vertex with minimal value of B-candidate impact parameter χ^2, etc). Unfortunately after completion of the algorithm and saving the B-candidate into Gaudi Transient Event store the ionformation about the associated primary vertex is lost. LOKI offers a simple way for accessing this information for saved B-candidates, using the predefined relations.

```
1
2       /// B-cnadidate
3       const Particle* B  = ... ;
4
5       /// some selected primary vertex
6       const Vertex*   pv = ... ;
7
8       /// some 'weight' for the association
9       const double weight = .... ;
10
11      /// make an association
12      asctPVs( B , pv , weight ) ;
```

For the tipical cases `weight` is either the impact parameter or impact parameter χ^2. Two-argument functions `asctPVsIP` and `asctPVsIPCHI2` are provided for this case.

The extraction of the information about the associated primary vertices is trivial:

```
1
2       /// B-cnadidate
3       const Particle* B  = ... ;
4
5       /// get all associatred primary vertices
6       P2PVRange pvs = PVs( B ) ;
7
8       /// loop over all associated vertices:
9       for ( P2PVRange::iterator pv = pvs.begin() ;
10      pvs.end() != pv ; ++pv )
11      {
12        /// extract the primary vertex
13        const Vertex* primary = pv->to()     ;
14        /// extract the associated weight (optionally)
15        const double  weight  = pv->weigth() ;
16      }
```

It is worth to mention that the extracted primary vertices are sorted according to the value of *weight*, therefore the vertex with minimal value of *weight* comes the first.

The relation table are saved into TES directory `OutputLocation` under the name `"P2PVs"`.

7.2 Access to decay products

LOKI offers a set of unclassified useful utilities to provide an easy access to decay products of the particles

```
1       const Particle* B = ... ;
2
3       const Particle* child_1 = child( B , 1 ) ;
4       const Particle* child_2 = child( B , 2 ) ;
```

Function **`child`** returns the pointer to i^{th} decay product. For invalid index or invalid argument, a zero pointer is returned. The same function allows also non-constant access to decay products:

```
1
2       Particle* B = ... ;
3
4       Particle* child_1 = child( B , 1 ) ;
5       Particle* child_2 = child( B , 2 ) ;
```

Complex navigation up to 4 levels in depth is possible:

```
1
2       const Particle* B = ... ;
3
4       /// the 2nd daughter of 3rd daughter of 1st daughter of B
5       const Particle* p1 = child( B1 , 1 , 3 , 2 ) ;
6
7       /// the 3rd daughter of 1st daughter of B
8       const Particle* p2 = child( B1 , 1 , 3 ) ;
```

The function is easy to chain:

```
1
2       const Particle* B = ... ;
3
4       /// the 2nd daughter of 3rd daughter of 1st daughter of B
5       const Particle* particle =
6                     child( child( child( B1 , 1 ) , 3 ) , 2 ) ;
```

The function can be directly applied to **`Loop`** objects also:

```
1
2       for( Loop phi = loop("K+ K-" , 333 ) ; phi ; ++phi  )
3       {
4
5         /// save ``interesting phi''
6         if( .... ) { phi->save( phi ) ; }
7
8       }
9
10      for( Loop Bs = loop("phi gamma" , 531 , NoFit ) ; Bs  ; ++Bs )
11      {
12
13        /// get the first daughter of the first daughter !
14        const Particle* kplus  = child ( Bs , 1 , 1 ) ;
15
16        /// get the second daughter of the first daughter !
17        const Particle* kminus = child ( Bs , 1 , 2 ) ;
18
19      }
```

7.3 Decay tree expansion

7.3.1 Extraction of `Particles`

An easy function **`getParticles`** allow to extract recursively all decay products from a given **`Particle`** object into flat container:

```
1
2       using namespace LoKi::Extract ;
3
4       const Particle* B = ... ;
5
6       /// extract ALL particles
7       ParticleVector PV;
8       getParticles( B , std::back_inserter( PV ) );
9
10      /// alternative name
11      /*
12         particles( B , std::back_inserter( PV ) );
13       */
```

In addition this function allows selective extraction of daughter particles of a given type

```
1
2       using namespace LoKi::Extract ;
3
4       const Particle* B = ... ;
5
6       ParticleVector PV;
7       /// extract all positive kaons
8       getParticles( B , std::back_inserter( PV ) , ParticleID(  321 ) );
9
10      /// append negative kaons to the list of extracted daughters
11      getParticles( B , std::back_inserter( PV ) , ParticleID( -321 ) );
```

The function can be applied to *any* sequence of **`Particle*`** objects:

```
1
2       using namespace LoKi::Extract ;
3
4       ParticleVector Bs  = ... ;
5
6       ParticleVector Kaons;
7       /// extract all positive kaons
8       getParticles( Bs.begin   ()              ,
9                     Bs.end     ()              ,
10                    std::back_inserter( Kaons ) ,
11                    ParticleID ( 321 )          ) ;
12
13      /// alternative name
14      /*
15         particles( Bs.begin   ()              ,
16                    Bs.end     ()              ,
17                    std::back_inserter( Kaons ) ,
18                    ParticleID ( 321 )          ) ;
19       */
```

7.3.2 Extraction of `ProtoParticles`

A function **`getProtoParticles`** allows to extract all **`ProtoParticles`** which contribute to a given **`Particle`** object:

```
1
2       using namespace LoKi::Extract ;
3
4       const Particle* B = ... ;
5
6       /// extract ALL protoparticles
7       std::vector<const ProtoParticle*> PPs;
8       getProtoParticles( B , std::back_inserter( PPs ) );
9
```

```
10      /// alternative name
11      /*
12         protoParticles( B , std::back_inserter( PPs ) );
13       */
```

The function can be applied to *any* sequence of **`Particle*`** objects:

```
1
2       using namespace LoKi::Extract ;
3
4       ParticleVector Bs  = ... ;
5
6       /// extract ALL protoparticles
7       std::vector<const ProtoParticle*> PPs;
8       getProtoParticles  ( Bs.begin () ,
9                            Bs.end   () , std::back_inserter( PPs ) );
10
11      /// alternative name
12      /*
13         protoParticles ( Bs.begin () ,
14                          Bs.end   () , std::back_inserter( PPs ) ) ;
15       */
```

7.3.3 Extraction of `MCParticles`

```
1
2       using namespace LoKi::Extract ;
3
4       const MCParticle* B = ... ;
5
6       /// extract ALL decay producst
7       std::vector<const MCParticle*> MCPs;
8       getMCParticles( B , std::back_inserter( MCPs ) ) ;
9
10      /// alternative name
11      /*
12         mcParticles( B , std::back_inserter( MCPs ) ) ;
13       */
```

Again, only daughter particles of a given type can be selectively extracted:

```
1
2       using namespace LoKi::Extract ;
3
4       const MCParticle* B = ... ;
5
6       /// extract daughter muons
7       std::vector<const MCParticle*> MCPs;
8       getMCParticles( B , std::back_inserter( MCPs ) , ParticleID( 13 ) );
9
10      /// alternative name
11      /*
12        mcParticles( B , std::back_inserter( MCPs ) , ParticleID( 13 ) ) ;
13      */
```

The function can be applied to *any* sequence of **`MCParticle*`** objects:

```
1
2       using namespace LoKi::Extract ;
3
4       MCParticles* Bs = ... ;
5
6       /// extract daughter muons
7       std::vector<const MCParticle*> MCPs;
8       getMCParticles( Bs->begin   ()              ,
9                       Bs->end     ()              ,
```

```
10                      std::back_inserter( MCPs ) ,
11                      ParticleID ( 13 )          ) ;
12
13      /// alternative name
14      /*
15         mcParticles( Bs->begin   ()             ,
16                      Bs->end     ()             ,
17                      std::back_inserter( MCPs ) ,
18                      ParticleID ( 13 )          ) ;
19      */
```

Chapter 8: Realistic examples

In this chapter few simple realistic examples of LOKI *algorithm* are presented.

8.1 Histogram example

The simplest realistic example of LOKI based algorithm is available at **$LOKIEXAMPLES-ROOS/src/LoKi_Histos.cpp**

```
1
2  // LoKi itself
3  #include "LoKi/LoKi.h"
4
5  // ============================================================================
6  LOKI_ALGORITHM( LoKi_Histos )
7  {
8    // avoid long names
9    using namespace LoKi        ;
10   using namespace LoKi::Cuts ;
11
12   // select all charged kaons
13   Range kaons = select ( "kaons" , 321 == abs( ID ) );
14
15   // plot the multiplicity of kaons
16   plot( kaons.size() , " Multiplicity of charged kaons " , 0 , 100 );
17
18
19   // fill few histos through the explicit loop over all charged kaons
20   for( Range::iterator iKaon = kaons.begin() ;
21        kaons.end() != iKaon ; ++iKaon )
22     {
23       const Particle* kaon = *iKaon ;
24       if( 0 == kaon ) { continue ; }
25
26       // plot the momentum of charged kaons
27       plot( P(kaon)/GeV , "Kaon momentum" , 0 , 200 ) ;
28
29       // plot the transverse momentum of charged kaons
30       plot( PT(kaon)/GeV , "Kaon transverse momentum" , 0 , 10 ) ;
31
32       // evaluate beta factor for kaon
33       double beta  = P( kaon ) / E( kaon ) ;
34       // plot it
35       plot( beta , "Beta for charged kaons" , 0.8 , 1.01 , 202 );
36
37     };
38
39   // fill the histo through the implicit loop over all charged kaons
40   plot( CL , kaons.begin() , kaons.end()  , "Confidence Level for kaons" , 0 , 1 );
41
42
43   return StatusCode::SUCCESS ;
44 };
45 // ============================================================================
```

8.2 N-Tuple example

```
1
2  // LoKi
3  #include "LoKi/LoKi.h"
```

```
4  // Event
5  #include "Event/EventHeader.h"
6
7  // ============================================================================
8  LOKI_ALGORITHM( LoKi_Tuple )
9  {
10   // avoid long names
11   using namespace LoKi        ;
12   using namespace LoKi::Cuts ;
13
14   // select all charged kaons with Confidence level in excess of 5 percent
15   Range kaons = select( "kaons" , 321 == abs( ID ) && 0.05 < CL ) ;
16
17   // subdivide sample of kaons to subsamples of positive and negative kaons
18   Range kplus  = select( "K+"  , kaons , Q >  0.5 ) ;
19   Range kminus = select( "K-"  , kaons , Q < -0.5 ) ;
20
21   // create the cut for phi :  abs(delta mass) < 15 MeV
22   Fun dmass = DMASS( "phi(1020)" , ppSvc() ) ;
23   Cut dmcut  = abs( dmass ) < 15 * MeV  ;
24
25   // get the N-Tuple (book it if not yet done)
26   Tuple tuple1  = nTuple ( "The first N-Tuple (scalar columns)" ) ;
27   for( Loop phi = loop( "K+ K-" , "phi(1020)" ) ; phi ; ++phi )
28     {
29       // put 4-momentum of phi
30       tuple1 -> column ( "phiP"    , phi    -> momentum () ) ;
31       // put 4-momentum of kplus
32       tuple1 -> column ( "kplusP"  , phi(1) -> momentum () ) ;
33       // put 4-momentum of kminus
34       tuple1 -> column ( "kminusP" , phi(2) -> momentum () ) ;
35
36       // fill separate columns
37       tuple1 -> column( "MASS"    , M     ( phi ) ) ;
38       tuple1 -> column( "DM"      , dmass ( phi ) ) ;
39       tuple1 -> column( "P"       , P     ( phi ) ) ;
40       tuple1 -> column( "PT"      , PT    ( phi ) ) ;
41
42       // fill few columns at once
43       Record rec( tuple1                          ,
44                   "LV01,LV02,CL1,CL2"             ,
45                   LV01 ( phi    ) , LV01 ( phi    ) ,
46                   CL   ( phi(1) ) , CL   ( phi(2) ) ) ;
47
48       // save "good" phi
49       if( dmcut( phi ) ) { phi->save("phi"); }
50     }
51
52   // get the second ntuple
53   Tuple tuple2 = nTuple( "The second N-Tuple (array-like columns)") ;
54
55   // get all saved phi
56   Range phi = selected("phi") ;
57
58   // fill the tuple
59   tuple2->farray( "M"    , M     ,
60                   "DM"   , dmass ,
61                   "P"    , P     ,
62                   "PT"   , PT    ,
63                   phi.begin ()   ,
64                   phi.end   ()   ,
65                   "nPhi"         ,
66                   100            ) ;
67
```

```
68    // add the event information to the N-Tuple
69    const EventHeader* evt =
70      get<EventHeader>( EventHeaderLocation::Default );
71
72    // add event header information to N-Tuple
73    tuple2 << Column( "" , evt );
74
75    // write N-Tuple
76    tuple2->write () ;
77
78    return StatusCode::SUCCESS ;
79  };
80  // ============================================================================
```

8.3 Event Tag Collection example

```
1
2   // $Id: LUG.tex,v 1.4 2005/07/14 18:53:40 ibelyaev Exp $
3   // ============================================================================
4   // CVS Tag $Name: v3r14 $
5   // ============================================================================
6   // $Log: LUG.tex,v $
7   // Revision 1.4  2005/07/14 18:53:40  ibelyaev
8   //  v3r14
9   //
10  // Revision 1.3  2005/01/21 11:21:25  ibelyaev
11  //  preparation to v3r7
12  //
13  // Revision 1.2  2004/11/19 12:34:03  ibelyaev
14  //  update for v3r6
15  //
16  // Revision 1.1  2004/10/22 11:06:49  ibelyaev
17  // *** empty log message ***
18  //
19  // Revision 1.15  2004/08/05 09:57:49  ibelyaev
20  //  update for LoKi v3r0
21  //
22  // Revision 1.14  2004/07/23 13:55:27  ibelyaev
23  //  regular update
24  //
25  // Revision 1.13  2004/07/08 19:43:43  ibelyaev
26  //  update
27  // ============================================================================
28  // Include files
29  // LoKi
30  #include "LoKi/LoKi.h"
31  // Event
32  #include "Event/EventHeader.h"
33  // GaudiKErnel
34  #include "GaudiKernel/IRegistry.h"
35
36  // ============================================================================
37  LOKI_ALGORITHM( LoKi_EventTagTuple )
38  {
39    // avoid long names
40    using namespace LoKi       ;
41    using namespace LoKi::Cuts ;
42
43    // select all charged kaons with Confidence level in excess of 5 percent
44    Range kaons = select( "kaons" , 321 == abs( ID ) && 0.05 < CL ) ;
45
46    // subdivide sample of kaons to subsamples of positive and negative kaons
47    Range kplus  = select( "K+"  , kaons , Q >  0.5 ) ;
48    Range kminus = select( "K-"  , kaons , Q < -0.5 ) ;
```

```
49
50    // create the cut for phi :  abs(delta mass) < 15 MeV
51    Fun dmass = DMASS( "phi(1020)" , ppSvc() ) ;
52    Cut dmcut  = abs( dmass ) < 15 * MeV  ;
53
54    for( Loop phi = loop( "K+ K-" , "phi(1020)" ) ; phi ; ++phi )
55      {
56        // save "good" phi
57        if( dmcut( phi ) ) { phi->save("phi"); }
58      }
59
60    // get the event tag collection/ntuple
61    Tuple tuple = evtCol( "EvtCollection") ;
62
63    // get all saved phi
64    Range phi = selected("phi") ;
65
66    // fill the tuple
67    tuple->farray( "M"    , M     ,
68                   "DM"   , dmass ,
69                   "P"    , P     ,
70                   "PT"   , PT    ,
71                   phi.begin ()   ,
72                   phi.end   ()   ,
73                   "nPhi"         ,
74                   100            ) ;
75
76    // add the event information to the N-Tuple
77    const EventHeader* evt =
78      get( eventSvc() , EventHeaderLocation::Default , evt );
79
80    // add event header information to N-Tuple
81    tuple << Column ( "" , evt );
82
83    // get the event IOpaqueAddress
84    DataObject* event = get<DataObject>( "/Event" );
85
86    // add event address to N-Tuple
87    tuple->column ( "Address" , event->registry()->address() ) ;
88
89    //
90    setFilterPassed( true ) ;     /// <<< IMPORTANT !
91
92    // write N-Tuple
93    tuple->write  () ;
94
95    return StatusCode::SUCCESS ;
96  };
97  // ============================================================================
```

8.4 Monte Carlo match example

```
1
2   // LoKi
3   #include "LoKi/LoKi.h"
4
5   /** @file
6    *
7    *  Implementation file for class : LoKi_Pi0fromBdTo3pi
8    *
9    *  @date   2003-03-12
10   *  @author Vanya  BELYAEV Ivan.Belyaev@itep.ru
11   */
12
```

```
13 // ============================================================================
14 LOKI_ALGORITHM( LoKi_Pi0fromBdTo3pi )
15 {
16   using namespace LoKi        ;
17   using namespace LoKi::Cuts  ;
18   using namespace LoKi::Fits  ;
19
20   // select all MC pi0s
21   MCMatch match( mctruth() );
22   MCRange b0s   = match->findDecays( "B0 -> pi+ pi- pi0 ");
23
24   // get all pi0s from MC decay trees
25   MCMatch::MCParticleVector pi0s;
26   LoKi::Extract::getMCParticles( b0s.begin   () ,
27                                  b0s.end     () ,
28                                  std::back_inserter( pi0s ) ,
29                                  ParticleID( 111 ) );
30   if( pi0s.empty() )
31     { return LOKI_ERROR( "No MC pi0 is found ", StatusCode::SUCCESS ) ; }
32
33   // get all photons
34   Range gammas = select( "gamma" , 22 == ID ) ;
35
36   Tuple tuple = nTuple("My pi0 tuple");
37   for( Loop pi0 = loop( "gamma gamma" , "pi0" ) ; pi0 ; ++pi0 )
38     {
39       if( pi0->mass(1,2) > 200 * MeV ) { continue ; }
40
41       tuple -> column("fmass" , pi0     -> mass      (1,2) ) ;
42       tuple -> column("pi0p"  , pi0     -> momentum ( 0 ) ) ;
43       tuple -> column("ph1"   , pi0(1) -> momentum ()     ) ;
44       tuple -> column("ph2"   , pi0(2) -> momentum ()     ) ;
45
46       long m1  =
47         pi0s.end() - match->match( pi0(1) , pi0s.begin() , pi0s.end() ) ;
48       long m2  =
49         pi0s.end() - match->match( pi0(2) , pi0s.begin() , pi0s.end() ) ;
50       long m  =
51         pi0s.end() - match->match( pi0    , pi0s.begin() , pi0s.end() ) ;
52
53       tuple->column( "m1"  , m1  ) ;
54       tuple->column( "m2"  , m2  ) ;
55       tuple->column( "m"   , m   ) ;
56
57       tuple->write();
58
59     };
60
61   return StatusCode::SUCCESS ;
62 };
63 // ============================================================================
```

8.5 Advanced example

As it was already pointed out, to get the full access to all LoKi features, the actual LoKi user should inherit his own*algorithm* from **`LoKi::Algo`** class. General LoKi user needs only to reimplement the virtual method **`LoKi::Algo::analyse`** which is invoked from the default implementation of virtual method **`LoKi::Algo::execute`**.

```
1
2 /// From LoKi package
3 #include "LoKi/LoKi.h"
```

```
4  ///
5  /// all tedious mandatory stuff: algorithm body,
6  /// constructor, destructor and the factory
7  LOKI_ALGORITHM_( PhiGamma )
8  {
9    using namespace LoKi       ;
10   using namespace LoKi::Cuts ;
11   using namespace LoKi::Fits ;
12   /// get all primary vertices with number of tracks > 5 and good Chi2
13   VRange pvs = vselect( " Good PVs " ,
14        VTYPE == Vertex::Primary && VTRACKS > 5 && VCHI2 / VDOF < 10 );
15   //
16   plot( pvs , "Tracks per primary vertiex " , VTRACKS , 0 , 20 , 20 );
17   /// select only one good Primary Vertex with maximal number of tracks
18
19   const Vertex* pv = select_max( pvs , VTRACKS  );
20   if( 0 == pv ) { return LOKI_ERROR( "No PV found!", StatusCode::SUCCESS) ; }
21
22   /// construct kaon cuts: kaons do not point to the primary vertex
23   Cut cut    = IP( point( pv ) ) > 100 * micrometer &&
24   IPCHI2( point( pv ) ) > 16 ;
25   /// kaons: confidence level and momentum/transverse momentum cuts
26   Cut kaons  = abs( ID )  == 231 && CL > 0.05 && PT > 20 * MeV && P > 2 * GeV ;
27   select( "kaon+"   , cut && kaons && Q >  0.5 ) ;
28   select( "kaon-"   , cut && kaons && Q < -0.5 ) ;
29
30   select( "Gamma"   , pt > 2 * GeV && ID == 22  ) ;
31
32   /// Phi cuts: vertex cut (>3 sigma) and (>1*mm) and delta mass
33   Cut phiCuts = VDCHI2( point( pv ) ) > 9 &&
34   VD( point( pv ) ) > 1 * mm && abs( DMMASS( "phi(1020)" , ppSvc() ) ) < 5 * MeV ;
35
36   for( Loop phi = loop( "kaon+ kaon-" , 333 ) ; phi ; ++phi )
37   {
38     if( phiCuts( phi ) ) { phi->save("phi");}
39   };
40
41   Fun time      = VDTIME ( point( pv ) ) ;
42   Fun len       = VDSIGN ( point( pv ) ) ;
43   Fun lenChi2   = VDCHI2 ( point( pv ) ) ;
44
45   /// get N-tuple, (book it if not yet created)
46   Tuple tuple = nTuple("My N-tuple for Bs");
47   for( Loop Bs  = loop ( "phi Gamma" , 531 ) ; Bs   ; ++Bs   )
48   {
49     /// make a record(``row'') for N-tuple
50     Record( tuple , " m, pt, length , lenChi2 , time ",
51             M( Bs ) , PT( Bs ) , len( Bs ) , lenChi2( Bs ) ,  time( Bs ) );
52   }
53   return StatusCode::SUCCESS;
54 };
```

The preprocessor macro **`LOKI_ALGORITHM`** can be used to avoid the writing of all mandatory and uniform code fragments - default(almost empty) algorithm body, the standard algorithm's constructor and destructor and the declaration of static algorithm factory, which is nesessary for run-time instantiation of the algorithm.

Chapter 9: What's new?

This chapter shortly summarizes the major new features of recent releases of LoKi. Not all detailes are described here. Full list of new features can be found in `$LOKIDOC/release.notes` file.

9.1 What's new in version v3r6

This version of LoKi comes with updated notion of *Named range*. From now, all *ranges* has also the field `name`, which allows to write following code:

```
1
2  Range pion = select( "pi+"   , ... ) ;
3  Range kaon = select( "kaon-" , ... ) ;
4
5  Loop D0 = loop( kaon + pion , "D0" ) ;
```

9.2 What's new in version v2r8

New *"MC Vertex Functions"* are added into LoKi:

`MCVTYPE`, **`MCVTOF`**, **`MCVX`**, **`MCVY`**, **`MCVZ`**.

Few helper funtions for some navigation in MC decay trees are added:

```
1
2  bool LoKi::MCTrees::fromMCTree
3   ( const MCParticle* particle ,
4     const MCParticle* parent   )
5  bool LoKi::MCTrees::fromMCTree
6   ( const MCParticle* particle ,
7     const MCVertex*   parent   )
8  bool LoKi::MCTrees::fromMCTree
9   ( const MCParticle* particle ,
10    const Collision*  parent   )
```

9.3 What's new in version v2r7

From this version LoKi supports the full semantics of decay descriptors, used for `IMCDecayFinder` tool

9.4 What's new in version v2r5

A set of "meta-functions" is added to LoKi: **`INTREE`**, **`NINTREE`**, **`TREESUM`**, **`TREEMULT`**, **`MAXTREE`**, **`MINTREE`**.

9.5 What's new in version v2r4

New MC matching helepr function **`RCTRUTH`** is added.

,,

```
1
2       // load all MC particles
3       typedef ::MCParticles MCPs;
4       const MCPs* mcps = get<MCPs>( MCParticleLocation::Default );
5
6       // get the reconstructed muon
7       const Particle* muon = ...;
8
9       // select *ALL* MC particles associted with given Partile:
10      MCContainer mcsels;
11      LoKi::select ( mcps->begin(), mcps->end(),
12      std::back_inserter ( mcsels ), RCTRUTH( mc, muon ) );
13
14      // ghost?
15      if ( mcsels.empty() ) { /* it is a ghost! */}
16
17      // loop over associated MC particles
18      for( MCContainer::const_iterator imc = mcsels.begin();
19                              mcsels.end() != imc; ++imc )
20      {
21        std::cout << " found " << MCID( *imc ) << std::endl;
22      }
```

9.6 What's new in version v2r2

Technical modifications doew to restructurisation of header file for **`IMCDecayFinder`** class.

9.7 What's new in version v2r1

This version of LoKi gets a lot of new function, dealing with different track types. In addition an powerful unary negater operator is introduces for predicates.

9.8 What's new in version v2r0

From the version v2r0 LOKI has lost all its histogram and ntuple facilities. All methods **`plot`** and **`ntuple`** are moved into the base classes **`GaudiHistoAlg`** and **`GaudiTupleAlg`** correspondingly. The class **`LoKi::Algo`** now inherits from **`GaudiTupleAlg`** algotithm. It results to few ***visible*** changes that need to be applied to the user code

- for all **`plot`** method the signature has beed changed: the histogram title and the quantity to be plotted change tehir positions in the argument list.
- **`ntuple`** method now is replaced with the menthod **`nTuple`**
- **`evtCollection`** method now is replaced with the menthod **`evtCol`**
- number of new metoods for inspection, booking, and retriving the hitstograms are introduces **`book`**. **`histo`**. **`histoExist`**.
- The maximal number of array-like columns for **`TupleObj`**. is reduced from 6 to 4

- There is no anymore direct possibility to put classes **`EventHeader`**, **`L0DUReport`** and **`L1Report`** into ntuple using method **`column`** .
- New utility **`Column`** allows to modyfy the default N-Tuple representation of objects, and to define the representationof new object
- Classes **`EventHeader`**, **`L0DUReport`** and **`L1Report`** can be put into N-Tuple using **`Column`** utility:

```
1
2       Tuple tuple = nTuple( ("My tuple" );
3       const EventHeader* hdr = get<EventHeader>( EventHeaderLocation::Default );
4       const L0DUReport*  l0  = get<L0DUReport> ( L0DUReportLocation ::Default );
5       const L1Report*    l1  = get<L1Report>   ( L1ReportLocation   ::Default );
6       tuple << Column( "" , hdr )
7             << Column( "" , l0  )
8             << Column( "" , l1  ) ;
```

New package DOC/LOKIDOC is created to keep all documentaton for LOKI. The LaTeX source of this manual is moved into directory `$LOKIDOCROOT/doc`.

9.9 What's new in version v1r9

The version **`v1r9`** of LOKI contains the new functions[1] `MVDCHI2` and `TCHI2NDF`: minimal $\chi 2$ of distance between particle decay vertex and all primary vertices and χ^2 per degree of freedom for track fit for charged particles correspondingly. alfo new functions `ABSID` `ADMASS` for evaluation of absolute values of `ID` and `DMASS` correspondingly. The new functions `VDDOT` and `VDDTIME` evaluate the projection of vertex distance to particle momentum direction, and the corresponding proper time difference. The new function `SUMQ` evaluates the charge of composed particle using the recursive sum over all all daughter particles.

Functions `VD`, `VDSIGN`, `VDTIME`, `VDCHI2`, `DIRA` have got more intuitive constructors. The previouse constructors will be removed from **v2r0**.

Class `Algo` gets new property `"Cuts"`, wich can be used for fast and simple switch of set of selection cuts between loose and tight.

Few fixed to simplify the "pythonization" of LOKI are apllied.

At last the LaTeX source of this manual is moved into directory `$LOKIDOC`.

9.10 What's new in version v1r8

The version **`v1r8`** contains the possibility of booking the histograms with forced assignement of histogram identifier. All methods of **`Algo::plot`** family have been duplicated to allow fix an integer histogram identifier (see section 5.1). Additional templated methods for filling histograms with information from argibrary sequences of arbitrary objects are provided.

[1] These functions were kindly proposed by Jeroen van Tilburg

9.11 What's new in version v1r7

The version **`v1r7`** of LOKI has no new functionality for physics analysis.

9.12 What's new in version v1r6

The version **`v1r6`** of LOKI comes with the new functionality of *loops over charge conjugated states* (see section 4.1.3):

```
1
2       CC ccK  ("K-","K+") ;
3       CC ccPi ("pi+","pi+") ;
4       for( LoopCC D0 = loop( ccK + ccPi , CC("D0","D~0") ) ; D0 ; ++D0 )
5       {
6       }
```

9.13 What's new in version v1r5

This version of LOKI comes with β**-versions** of Flavour Tag access and possibility to propagate the association of B-candidates with primary vertices outside the algorithm (see section 7.1).

The set of available kinematical fits includes now *lifetime fit* using the tool, implementing the `ILifetimeFitter` abstract interface (see section 4.5). In addition the accessor to `ILifetimeFitter` is provided by `LoKi::Algo` base class with the name `lifetimeFitter()`

New functions `PID`, `PIDe`, `PIDmu`, `PIDpi`, `PIDK`, `PIDp` for access to particle indentification information are provided.

New functions `MIP` and `MIPCHI2` are provides for easy evaluation of minimal particle impact parameter (and impact parameter χ^2) with respect to few vertices.

New functions `VIP` and `VIPCHI2` (see section 13.4) are provides for easy evaluation of vertex impact parameter (and impact parameter χ^2) with respect to the particle.

9.14 Next steps

9.14.1 Looping

The implementation of "correct" procedure for 1-particle looping is expected. It will allow to apply the whole LoKi machinery for particles which are already selected. In particular the possibility to (re)apply the kimenatical contraints to the selected particles is foreseen.

The comissioning of "loop over charge-conjugated states" is expected.

9.14.2 Kinematical constrains

The incorporation of "global fitter" tool developed by Bologna's team into LoKi is expected as soon the tool becomes public.

The unification of kinematical fitters throught the common abstarct interface is planned.

The procedure of correct re-fit need to be developed.

The total reimplementation of existing tools for vertex and mass-vertex fits is required.

9.14.3 Tagging

New B-favour tagging need to be designed from scratch and incorporated into LoKi in a proper way. The existing code is not valid anymore.

9.14.4 B-production vertices

The association of B-candidates with their B-production vertices need to be comissioned. All existing experience need to be collected and revisited.

9.14.5 MC-Decay finder

More tight integration of MCDecayFinder tool with LoKi is epxtected, namely the usage of ^ need to be supported for decay destrciptors in `findDecay` method

Additional methods for `mcselect` and `mcselected` for class `LoKi::Algo` are to be implemented to achieve more coherent treatment of reco and MC data

9.14.6 User-defined information

The possibily for dynamic object extension with used-defined information based on Relations package need to be developed. E.g. one needs to be able to attach some "USerInfo" for selected particles.

9.14.7 MC-associations

The examples of "alternative" MC-association with standard DaVinci associators need to be set.

9.14.8 Generic algorithms with "Cut on configure"

The development of the algorith which can aquire the Python-coded cuts, defined as "configure" phase.

Chapter 10: Frequently Asked Questions

Part II

Reference Manual

Only few topics are described here in details. For others see LOKI reference manual automatically generated from the sources by DOXYGEN tool.

Chapter 11: Access to LOKI sources

11.1 Import LOKI from CVS repository

LoKi now[1] resides in standard LHCb CVS repository under the hat `Tools` and and therefore standard tools, like the **`getpack`** are to be used for accessing LOKI sources.

```
1 >
2 >  cd ~/newmycmt
3 >  getpack Tools/LoKi v<XXXX>
4 >  cd LoKi/v<XXXX>/cmt
5
6 >
```

The special package LOKIEXAMPLE exists with few examples of simple analysis algorithms:

```
1 >
2 >  cd ~/newmycmt
3 >  getpack Ex/LoKiExample head
4 >
```

11.2 Building package and documentation

Unfortunately according to position of DAVINCI officials all available LOKI documentation is not referenced from the official DAVINCI documentation and web-pages and the DOXYGEN documentattion of LOKI source is not built officially. Therefore LOKI users need to built he documentation themselves. Building of the LOKI library, documentation, and CMT graphical representation of LOKI dependencies is performed with the help of new package DOC/LOKIDOC

```
1 >
2 >  DaVinciEnv <XXX?
3 >  cd ~/newmycmt
4 >  getpack Doc/LoKiDoc v<XXX>
5 >  cd Doc/LoKiDoc/v<XXX>/cmt
6 >  source setup.[c]sh
7 >
8 >  cd ../doc
9 >  make
10 >
```

The DOXYGEN generated documentation (in HTML format) is available at `$LOKIDOC/-html/index.html` and could be viewed with your favorite web-brouser. In parallel manual pages are created and one can use standard Linux(Unix) commands `man` or `info` to get information about LOKI:

[1] from end of April 2002

```
>
> man LoKi
> ...
> info Cuts.h
>
```

The produced file `$LOKIDOC/LoKi.gif` shows all CMT dependency of LOKI package.

This document ("LoKi User Guide and Reference Manual") is available in the directory `$LOKIDOC`. Four `ps` and `pdf` files with different formats and layouts of this document are generated: `LoKi.ps`, `LoKi.pdf`, `LoKi_2on1.ps` and `LoKi_2on.pdf`

11.3 Running standard examples

To run standard LOKI examples one need to build the standard DAVINCI executable with LOKI:

```
>
> cd ~/newmycmt
> DaVinciEnv v<XXXX>
> getpack Phys/DaVinci v<XXXX>
> cd Phys/DaVinci/v<XXXX>/cmt
```

Add the following line into the file requirements: `use LoKiExample v* Ex`

```
> cmt config
> cmt br make
> source setup.[c]sh
>
```

The directory `$LOKIEXAMPLESOPTS/options` contains few standard opions to run simple examples, e.g.

```
>
> cd ~/newmycmt/Phys/DaVinci/v<XXXX>/cmt
> DaVinciEnv v<XXXX>
> cmt config
> source setup.[c]sh
>
> ../rh73_gcc2952/DaVinci.exe $LOKIEXAMPLESOPTS/<FILE>
>
```

Possible values of `<FILE>` are:

- `DV_LoKi_Demo1.opts`
- `DV_LoKi_Histos.opts`
- `DV_LoKi_Tuple.opts`
- `DV_LoKi_EventTagTuple.opts`

- `DV_LoKi_Bs2PhiGamma.opts`
- `DV_LoKi_Pi0fromBdTo3pi.opts`
- `DV_LoKi_Bd2KStarGamma.opts`
- `DV_LoKi_Bs2PhiPhi.opts`

11.4 Configuration

The “typical” configuration of DaVinci for usage of LoKi-based algorithms could be:

```
1
2       // LoKi & LoKiExamples   itself
3       ApplicationMgr.DLLs     += {  "LoKi" ,  "LoKiExamples" } ;
4
5       ApplicationMgr.TopAlg   += { "LoKi_Bs2PhiGamma/PhiGamma"  } ;
6       PhiGamma.PhysDesktop.InputPrimaryVertices =
7             "/Event/Phys/PrimaryVertices"  ;
8       PhiGamma.PhysDesktop.InputLocations   =
9           {"/Event/Phys/Charged","/Event/Phys/Neutral" } ;
10      PhiGamma.PhysDesktop.OutputLocation   =
11            "/Event/Phys/Bs2PhiGamma"       ;
12      PhiGamma.PhotonTool.ScaleFactor = 10.0 ;
```

Chapter 12: Basic LOKI algorithm

The working horse of LOKI package is the basic LOKI algorithm - class **`Algo`**. It is a specialisation of general `GaudiAlgorithm` class from GAUDIALG package:

```
1
2  namespace LoKi
3  {
4    /** @class Algo Algo.h LoKi/Algo.h
5     *
6     *  The  working horse of LoKi package
7     *  basic LoKi algorithm
8     *
9     *  @author Vanya BELYAEV Ivan.Belyaev@itep.ru
10    *  @date   2002-07-12
11    */
12   class Algo : public GaudiAlgorithm
13   {
14   public:
15   ...
16   };
17 }; // end of namespace LoKi
```

12.1 Major methods of `Algo` class

The class **`Algo`** provides users with an easy access to almost all LOKI functionalities as well as all major DAVINCI tools from DAVINCITOOLS, DAVINCIMCTOOLS and DAVINCIASSOCIATORS packages.

Among the major and visible for users methods of **`Algo`** class one finds

- various **`select`** and **`vselect`** methods for selection/filtering of particles/vertices which satisfy certain selection/filter criteria
- **`selected`** and **`vselected`** methods for extraction collections of particles/vertices, selected previously
- the family of **`loop`** methods for general looping over combinations of particles
- the family of **`pattern`** method for looping over combinations of particles, selection of combination which satisfies the certain selection criteria and saving the results as new particles.
- the family of **`plot`** methods for booking and filling of the histograms, including helpful short-cuts for combination of filling histograms withing trivial loop over particles/vertices or their combinations
- **`tuple`** method, which is used for booking and filling of the N-tuples
- **`evtCollection`** method, which is used for booking and filling of Event Tag Collections

- **`mctruth`** method, which is used for convenient access to Monte Carlo truth information
- **`point`** method, which is used for instantiation of non-trivial functions and cuts with heavy usage of **`IGeomDispCalculator`**, e.g. impact parameter of the track with respect to the primary vertex

Each instance of class **`Algo`** is required to be initialized (and cleared) on event-by-event basis. Therefore it is assumed that LoKi client redefines the **`Algo::analyse`** method which in turn is called from standard **`Algo::execute`**, where all nesessary initialization is performed:

```
1
2  // ============================================================================
3  /** standard execution of the Algorithm
4   *  @see    Algorithm
5   *  @see  IAlgorithm
6   *  @return satus code
7   */
8  // ============================================================================
9  StatusCode LoKi::Algo::execute()
10 {
11   // reset the filter indicator
12   setFilterPassed( false );
13   // load particles to desktop
14   desktop()->getInput();
15   // clear all LoKi storages
16   LoKi::Algo::clear() ;
17   // call for actual analysis
18   StatusCode sc = analyse() ;   /// NB !
19   if( sc.isFailure() ) { return Error("Error from 'analyse()' method" , sc ) ; }
20   /// save everything
21   sc = desktop()->saveDesktop();
22   if( sc.isFailure() ) { return Error("DeskTop is not saved"          , sc ) ;}
23   // clear all LoKi storages at the end
24   LoKi::Algo::clear() ;
25   return StatusCode::SUCCESS ;
26 };
27 // ============================================================================
```

12.2 Major data types defined in `Algo` class

Class **`LoKi::Algo`** defines several useful types for easy manipulation of objects and methods. The most important data types are[1]

`Range` light-weight representation of container of pointers to **`Particle`** objects.

`VRange` light-weight representation of container of pointers to **`Particle`** objects.

`MCRange` light-weight representation of container of pointers to **`MCParticle`** objects.

All threse types are specialisation of generic templated class **`LoKi::Range_`**<**`ITERATOR`**>. For these types the following methods are defined[2]

[1] For external access (e.g. out of the scope of **`analyse`** method) one should prepend the type name with actual scope identifi er **`LoKi::Algo`**

[2] Standard STL interface for data sequences

`empty()` returns **`true`** if the sequence is empty

`size()` returns the actual size of the sequence

`begin()` returns **`begin`** iterator of the sequence

`end()` returns **`end`** iterator of the sequence

`rbegin()` returns the reverse **`begin`** iterator

`rend()` returns the reverse **`end`** iterator

`front()` returns the value of the first element. The result is undefined for empty ranges

`back()` returns the value of the last element. The result is undefined for empty ranges

`operator()` returns the value of i^{th} element. The result is undefined for invalid index

`operator[]` returns the value of i^{th} element The result is undefined for invalid index

`at()` returns the value of i^{th} element, throws the *exception* for invalid index

12.3 Implementation of analysis algorithms

12.3.1 Standard minimalistic algorithm

In general the algorithm, based on functionalities offered by LOKI package needs to overwrite only the default **`analyse`** method. The minimalistic header (`"*.h"`) file is:

```
1
2  // ===========================================================================
3  // $Id: LUG.tex,v 1.4 2005/07/14 18:53:40 ibelyaev Exp $
4  // ===========================================================================
5  // CVS tag $Name: v3r14 $
6  // ===========================================================================
7  // $Log: LUG.tex,v $
8  // Revision 1.4  2005/07/14 18:53:40  ibelyaev
9  //  v3r14
10 //
11 // Revision 1.3  2005/01/21 11:21:25  ibelyaev
12 //  preparation to v3r7
13 //
14 // Revision 1.2  2004/11/19 12:34:03  ibelyaev
15 //  update for v3r6
16 //
17 // Revision 1.1  2004/10/22 11:06:49  ibelyaev
18 // *** empty log message ***
19 //
20 // Revision 1.15  2004/08/05 09:57:49  ibelyaev
21 //  update for LoKi v3r0
22 //
23 // Revision 1.14  2004/07/23 13:55:27  ibelyaev
24 //  regular update
25 //
26 // Revision 1.13  2004/07/08 19:43:43  ibelyaev
27 //  update
```

```
28 // ============================================================================
29 #ifndef MYPACKAGE_MYALG_H
30 #define MYPACKAGE_MYALG_H 1
31 // Include files
32 // from LoKi
33 #include "LoKi/Algo.h"
34
35 /** @class MyAlg MyAlg.h GaudiDoc/MyAlg.h
36  *
37  *  Simple private analysis algorithm
38  *
39  *  @author Vanya BELYAEV Ivan.Belyaev@itep.ru
40  *  @date   2003-02-05
41  */
42 class MyAlg : public LoKi::Algo
43 {
44   /// friend factory for instantiation
45   friend class AlgFactory<MyAlgo>;
46 public:
47   /** Perform the analysis of current event.
48    *  The method is invked from standard LoKi::Algo::execute method
49    *  @see LoKi::Algo
50    *  @return statsu code
51    */
52   virtual StatusCode analyse () ;
53 protected:
54   /** Standard constructor
55    *  @see LoKi::Algo
56    *  @see GaudiAlgorithm
57    *  @see       Algorithm
58    *  @param name name of the algorithm's instance
59    *  @param svc  pointer to Service Locator
60    */
61   MyAlg( const std::string& name ,
62          ISvcLocator*       svc  ) ;
63   /// virtual detsructor
64   virtual ~MyAlg( ) ;
65 private:
66   /// default constructor is private
67   MyAlg() ;
68   /// copy constructor is private
69   MyAlg           ( const MyAlg& ) ;
70   /// assignement operator is private
71   MyAlg& operator=( const MyAlg& ) ;
72 };
73 // ============================================================================
74 // The END
75 // ============================================================================
76 #endif // MYPACKAGE_MYALG_H
77 // ============================================================================
```

The corresponding minimalistic implementation (`"*.cpp"`)file is

```
1
2  // ============================================================================
3  // $Id: LUG.tex,v 1.4 2005/07/14 18:53:40 ibelyaev Exp $/
4  // ============================================================================
5  // CVS tag $Name: v3r14 $
6  // ============================================================================
7  // $Log: LUG.tex,v $
8  // Revision 1.4  2005/07/14 18:53:40  ibelyaev
9  //  v3r14
10 //
11 // Revision 1.3  2005/01/21 11:21:25  ibelyaev
12 //  preparation to v3r7
```

```
13 //
14 // Revision 1.2  2004/11/19 12:34:03  ibelyaev
15 //  update for v3r6
16 //
17 // Revision 1.1  2004/10/22 11:06:49  ibelyaev
18 // *** empty log message ***
19 //
20 // Revision 1.15  2004/08/05 09:57:49  ibelyaev
21 //  update for LoKi v3r0
22 //
23 // Revision 1.14  2004/07/23 13:55:27  ibelyaev
24 //  regular update
25 //
26 // Revision 1.13  2004/07/08 19:43:43  ibelyaev
27 //  update
28 // ============================================================================
29 // Include files
30 // from Gaudi
31 #include "GaudiKernel/AlgFactory.h"
32 // local
33 #include "MyAlg.h"
34 /** @file
35  *  Implementation file for class MyAlg
36  *
37  *  @author Vanya BELYAEV Ivan.Belyaev@itep.ru
38  *  @date 2003-02-05
39  */
40 // ============================================================================
41 /** @var MyAlgFactory
42  *  Declaration of the Algorithm Factory
43  */
44 // ============================================================================
45 static const  AlgFactory<MyAlg>         s_Factory ;
46 const        IAlgFactory&MyAlgFactory = s_Factory ;
47 // ============================================================================
48
49 // ============================================================================
50 /** Standard constructor
51  *  @see LoKi::Algo
52  *  @see GaudiAlgorithm
53  *  @see      Algorithm
54  *  @param name name of the algorithm's instance
55  *  @param svc  pointer to Service Locator
56  */
57 // ============================================================================
58 MyAlg::MyAlg ( const std::string& name ,
59                ISvcLocator*       svc  )
60   : LoKi::Algo( name , pSvcLocator )
61 {};
62 // ============================================================================
63
64 // ============================================================================
65 /// destructor
66 // ============================================================================
67 MyAlg::~MyAlg() {};
68 // ============================================================================
69
70 // ============================================================================
71 /** Perform the analysis of current event.
72  *  The method is invked from standard LoKi::Algo::execute method
73  *  @see LoKi::Algo
74  *  @return status code
75  */
76 // ============================================================================
```

```
77  StatusCode MyAlg::analyse ()
78  {
79    // recommended to avoid long typeing
80    using namespace LoKi        ;
81    using namespace LoKi::Cuts  ;
82    using namespace LoKi::Fits  ;
83    //
84    Print("␣analyse()␣method␣is␣invoked!");
85    return StatsuCode::SUCCESS ;
86  };
87  // ============================================================================
88
89  // ============================================================================
90  // The END
91  // ============================================================================
```

12.3.2 Useful macros

LoKi offers a set of preprocessor macros to make the algoritms even more compact. The default body of the algorithm is generated by the following macro:

```
1
2  // ============================================================================
3  #include "LoKi/LoKi.h"
4  // ============================================================================
5  LOKI_ALGORITHM_BODY( MyAlg );
6  // ============================================================================
```

The default impementation of algorithm's factory, constructor and destructor is generated by another simple macro

```
1
2  // ============================================================================
3  #include "LoKi/LoKi.h"
4  // ============================================================================
5  #include "MyAlg.h"
6  // ============================================================================
7  LOKI_ALGORITHM_IMPLEMENT( MyAlg );
8  // ============================================================================
9  /** Perform the analysis of current event.
10  *  The method is invked from standard LoKi::Algo::execute method
11  *  @see LoKi::Algo
12  *  @return statsu code
13  */
14 // ============================================================================
15 StatusCode MyAlg::analyse ()
16 {
17   // recommended to avoid long typeing
18   using namespace LoKi        ;
19   using namespace LoKi::Cuts  ;
20   using namespace LoKi::Fits  ;
21   Print("␣analyse()␣method␣is␣invoked!");
22   return StatsuCode::SUCCESS ;
23 };
24 // ============================================================================
25
26 // ============================================================================
27 // The END
28 // ============================================================================
```

Both generation of default algorithm body and default implementation of algorithm factory, constructor and destructor is performed by the following macro:

```
1
2  // ============================================================================
3  #include "LoKi/LoKi.h"
4  // ============================================================================
5  LOKI_ALGORITHM_FULLIMPLEMENT( MyAlg );
6  // ============================================================================
7  /** Perform the analysis of current event.
8   *  The method is invoked from standard LoKi::Algo::execute method
9   *  @see LoKi::Algo
10  *  @return statsu code
11  */
12 // ============================================================================
13 StatusCode MyAlg::analyse ()
14 {
15   // recommended to avoid long typeing
16   using namespace LoKi        ;
17   using namespace LoKi::Cuts  ;
18   using namespace LoKi::Fits  ;
19   //
20   Print("analyse() method is invoked!");
21   return StatsuCode::SUCCESS ;
22 };
23 // ============================================================================
24
25 // ============================================================================
26 // The END
27 // ============================================================================
```

One could use even less verbose approach

```
1
2  // ============================================================================
3  #include "LoKi/LoKi.h"
4  // ============================================================================
5  LOKI_ALGORITHM( MyAlg )
6  {
7    // recommended to avoid long typeing
8    using namespace LoKi        ;
9    using namespace LoKi::Cuts  ;
10   using namespace LoKi::Fits  ;
11   //
12   Print("analyse() method is invoked!");
13   return StatsuCode::SUCCESS ;
14 };
15 // ============================================================================
16
17 // ============================================================================
18 // The END
19 // ============================================================================
```

If one uses this macro, there is no necessity to have a header file. Both algorithm declaration and implementation go into the same file.

12.4 Standard properties of `Algo` class

The standard configuration properties of class **`Algo`** are listed in table 12.1. All properties from base classes **`Algorithm`** (GAUDIKERNEL package), **`GaudiAlgorithm`** **`GaudiHistoAlg`** and **`GaudiTupleAlg`** (GAUDIALG package) could also be used for the configuration of algorithm.

Table 12.1: The standard properties of `LoKi::Algo` and their default values.

Property Name	Default Value
Properties defined for class **`Algorithm`**	
`"OutputLevel"`	`4`
`"Enable"`	`true`
`"ErrorMax"`	`10`
`"ErrorCount"`	`0`
`"AuditInitialize"`	`false`
`"AuditExecute"`	`true`
`"AuditFinalize"`	`false`
Properties defined for class **`GaudiHistoAlg`**	
`"HistosProduce"`	`true`
`"HistoCheckForNaN"`	`true`
`"HistoSplitDir"`	`true`
`"HistoOffSet"`	`0`
`"HistoTopDir"`	`""`
`"HistoDir"`	instance name
Properties defined for class **`GaudiTupleAlg`**	
`"NTupleProduce"`	`true`
`"NTupleSplitDir"`	`true`
`"NTupleOffSet"`	`0`
`"NTupleLUN"`	`"FILE1"`
`"NTupleTopDir"`	`""`
`"NTupleDir"`	instance name
`"EvtColsProduce"`	`true`
`"EvtColSplitDir"`	`true`
`"EvtColOffSet"`	`0`
`"EvtColLUN"`	`"EVTCOL"`
`"EvtColTopDir"`	`""`
`"EvtColDir"`	instance name
Properties defined in class **`LoKi::Algo`**	
`"Cuts"`	`LoKi::SelectionCuts::NotDefined`
`"Desktop"`	`"PhysDesktop"`
`"MassVertexFitters"`	`{"LagrangeMassVertexFitter/MassVrtxFit"}`

(Continuation on the next page)

Table 12.1 *(Continuation)*

Property Name	Default Value
`"VertexFitters"`	{`"UnconstVertexFitter/VertexFit"`}
`"DirectionFitters"`	{`"LagrangeDirectionFitter/DirectionFit"`}
`"LifetimeFitters"`	{`"LifetimeFitter/TimeFit"`}
`"Stuffers"`	{`"ParticleStuffer/Stuffer"`}
`"Filters"`	`"ParticleFilter/Filter"`
`"FilterCriteria"`	{} (empty vector)
`"GeomDispCalculator"`	`"GeomDispCalculator/GeometryTool"`
`"TaggingTools"`	{} (empty vector)
`"DecayFinder"`	`"DecayFinder"`
`"UseMCTruth"`	`true`
`"MCDecayFinder"`	`"MCDecayFinder"`
`"MCpwAssociators"`	{} (empty vector)
`"MCnwAssociators"`	{} (empty vector)
`"MCppAssociators"`	{ ”AssociatorWeighted<ProtoParticle,MCParticle,double>/NPPs2MCPs” , ”AssociatorWeighted<ProtoParticle,MCParticle,double>/ChargedPPs2MCPs” }
`"DecayDescriptor"`	`""` (empty string)
`"AvoudSelResults"`	`false`
`"SelResultLocation"`	`"/Event/Phys/Selections"`
`"OutputLocation"`	`"/Event/Phys`⟨name⟩`"`

Chapter 13: *Functions*

13.1 More about *Functions*/*Variables*

LOKI offers a large set of *functions* or *variables*. They are naturally subdivided into the *particle functions* and *vertex functions*. The former can be applied for objects of type **`const Particle*`** and the latter can be applied for the objects of type **`const Vertex*`**. All LOKI *Particle functions* are listed in table 13.2 and all LOKI *Vertex functions* are listed in table 13.3.

Since the objects of type **`Loop`** are implicitely convertable both to **`const Particle*`** and **`const Vertex*`** types, all LOKI *functions* can be directly applied to **`Loop`** objects:

```
1
2       for( Loop D0 = loop( "K- pi+", "D0" ) ; D0 ; ++D0 )
3       {
4         const Particle* p = D0->particle () ;
5         double mass1      = M  ( p  ) ;
6         double mass2      = M  ( D0 ) ; // use the implicit conversion to Particle*
7         const Vertex* v   = D0->vertex   () ;
8         double vx1        = VX ( v  ) ;
9         double vx2        = VX ( D0 ) ; // use the implicit converrion to Vertex*
10      }
```

The result of elementary mathematical operations on LOKI *functions* are again *LoKi functions* and the result can be directly assigned to `Fun` or `VFun` for *particle functions* and *vertex functions* correspondignly:

```
1
2       //
3       const Particle* p = ... ;
4       Fun fun1 = ( sqrt( M / GeV ) + atan( 1 / PT ) ) / log( MM / P ) ;
5       Fun fun2 = fun1 / log( log( log ( acos( E + 1 * MeV ) ) ) ) ;
6       Fun fun3 = fun2 + IP( point( vx ) ) / sin( VD( point( vx ) ) ) ;
7       double result = fun3( p ) ;
8       //
9       const Vertex*  v = ... ;
10      VFun vfun1 = VX / VZ + VZ / VX / exp( VY / 0.1 * mm ) ;
11      VFun vfun2 = vfun1 + VTYPE / VCHI2 * sin( VDOF ) ;
12      double vresult = vfun2( v ) ;
```

All *functions* implements the abstract interface **`LoKi::Function<TYPE>`** and are equipped with the method **`double operator()( const TYPE& ) const`**. The design of LOKI *functions* is very similar to the design of templated **`GenFunctor`** class from LOKI [10] library and the design of **`AbsFunction`** abstract class hierarchy from CLHEP [9] library.

The actual implementations of LOKI *functions* are scattered through several LOKI files and different namespaces, but all of them are finally collected in the file **`LoKi/Cuts.h`** inside the namespace **`LoKi::Cuts`**.

13.2 Operations with *functions*

The result of the addition, subtraction, multiplication and division of two *functions* is the *function*. Also application of the most of elementary functions results in a new *functions* object. All available elementary mathematical functions, which can be applied to LOKI *functions* are listed in table 13.1.

```
1
2       Fun fun1 = sqrt( E * E - PX * PX - PY * PY - PZ * PZ );
3       Fun fun2 = fun1 - M ;
4       const Particle* p = ... ;
5       double value = fun2( p ) ; // should always be 0 within rounding errors
```

Here `Fun` is a predefined type of a function holder object for *particle functions*. Any *function* object can be assigned to the objects of the type `Fun`.

Table 13.1: Elementary mathematical functions defined for all LOKI *functions*.All *functions* are defined in header file `LoKi/Functions.h` inside the namespace `LoKi::Functions`.

`abs`	absolute value
`log`	natural logarithm
`log10`	base 10 logarithm
`exp`	exponentiation
`sqrt`	square root
`sin`	sine
`cos`	cosine
`tan`	tangent
`sinh`	hyperbolic sine
`cosh`	hyperbolic cosine
`tanh`	hyperbolic tangent
`asin`	inverse sine
`acos`	inverse cosine
`atan`	inverse tangent
`pow`	power functions of two variables
`atan2`	inverse tangent function of two variables

The similar type `VFun` is defined for *vertex functions*:

```
1
2       VFun fun1  = sqrt( VX * VX + VY * VY + VZ * VZ )
3       VFun fun2  = VX / fun1 ;
4       const Vertex* v = ... ;
5       double value = fun2( v ) ;
```

All *functions* can be applied directly to the sequence of objects:

```
1
2       ParticleVector particles = ... ;
3       Fun            function  = ... ;
4
5       // the first way (not very efficient, includes coping)
6       std::vector<double> result1 =
7          function( particles.begin () ,
8                    particles.end   () ) ;
```

```
9
10      // the second way (efficient)
11      std::vector<double> result2( particles.size() );
12      function( particles.begin () ,
13                particles.end   () ,
14                result2.begin   () ) ; // output
15
16      // the third way (explicit usage of STL algorithm, efficient)
17      std::vector<double> result3( particles.size() ) ;
18      std::transform( particles.begin () ,
19                      particles.end   () ,
20                      result3.begin   () ,   // output
21                      function           ) ; // transformer
```

13.3 *Particle functions*

The predefined *particle functions* are listed in table 13.2. All of them get the argument of type **`const Particle*`**.

13.4 *Vertex functions*

The predefined LoKi *vertex functions* are listed in table 13.3. All of them get the argument of type **`const Vertex*`**.

13.5 *Monte Carlo Particle functions*

The predefined *MC Particle functions* are listed in table 13.4. All of them get the argument of type **`const MCParticle*`**.

13.6 *Monte Carlo Vertex functions*

The predefined LoKi *MC Vertex functions* are listed in table 13.5. All of them get the argument of type **`const MCVertex*`**.

Table 13.2: Predefined LoKi *Particle functions*. All *functions* are defined in header file `LoKi/ParticleCuts.h` inside the namespace `LoKi::Cuts`

Function			The Actual type	Description
ONE	*F*	*O*	`LoKi::Constant<const Particle*>`	Function which always evaluates to **`1`**
PONE	*F*	*O*	`LoKi::Constant<const Particle*>`	Another name for **`ONE`**
PZERO	*F*	*O*	`LoKi::Constant<const Particle*>`	Function which always evaluates to **`0`**
PALL	*P*	*O*	`LoKi::BooleanConstant<const Particle*>`	Predicate which always evaluates to **`true`**
PTRUE	*P*	*O*	`LoKi::BooleanConstant<const Particle*>`	Another name for **`PALL`**
PNONE	*P*	*O*	`LoKi::BooleanConstant<const Particle*>`	Predicate which always evaluates to **`false`**
PFALSE	*P*	*O*	`LoKi::BooleanConstant<const Particle*>`	Another name for **`PNONE`**

MIN	*F*	*T*	`LoKi::Min<const Particle*>`	Function which evaluates the minimum value from few other functions
				`1`
				`2 Fun pt1 = CHILD ( PT , 1 ) ;`
				`3 Fun pt2 = CHILD ( PT , 2 ) ;`
				`4 Fun pt3 = CHILD ( PT , 3 ) ;`
				`5 Fun mn  = MIN( pt1 , pt2 , pt3 ) ;`

PMIN	*F*	*T*	`LoKi::Min<const Particle*>`	Another name for **`MIN`**
MAX	*F*	*T*	`LoKi::Max<const Particle*>`	Function which evaluates the maximum value from few other functions (similar to **`MIN`**)
PMAX	*F*	*T*	`LoKi::Max<const Particle*>`	Another name for **`MAX`**

(Continuation on the next page)

Table 13.2 *(Continuation)*

Function			The Actual type	Description

ID	*F*	*O*	`LoKi::Particles::Identifier`	Particle Identifier
				The function relies on `Particle::particleID().pid()` mehtod
				`const Particle* p = ... ;`
				`if ( 211 == ID( p ) ) { ... } ;`
				The function can be used in comparison with the particle string identifier/name
				`Range kaons = select ( "Kaon+" , "K+" == ID ) ;`

ABSID	*F*	*O*	`LoKi::Particles::AbsIdentifier`	Absolute value of Particle Identifier The function relies on evaluation of `abs(Particle::particleID().pid())` symbolic expression
CL	*F*	*O*	`LoKi::Particles::ConfidenceLevel`	Confidence Level The function relies on `Particle::confLevel` method
Q	*F*	*O*	`LoKi::Particles::Charge`	Charge `Particle::charge` method is used
SUMQ	*F*	*O*	`LoKi::Particles::SumCharge`	Charge evaluated (recursively) from the sum of charges for all daughter particles. For the basic particles is evaluates as **Q**
P	*F*	*O*	`LoKi::Particles::Momentum`	$\|\vec{\mathbf{p}}\|$ The method `Particle::p` is used for evaluation
P2	*F*	*O*	`LoKi::Particles::Momentum2`	$\vec{\mathbf{p}}^2$ The function relies of `Particle::momentum().vect().mag2()` method
PT	*F*	*O*	`LoKi::Particles::TransverseMomentum`	$\mathrm{p_T}$ The method `Particle::pt` is used for evaluation

(Continuation on the next page)

Table 13.2 *(Continuation)*

Function			The Actual type	Description
PX	*F*	*O*	`LoKi::Particles::MomentumX`	p_x The symbolic expression `Particle::momentum().px()` is used for evaluation
PY	*F*	*O*	`LoKi::Particles::MomentumY`	p_y The symbolic expression `Particle::momentum().py()` is used for evaluation
PZ	*F*	*O*	`LoKi::Particles::MomentumZ`	p_z The symbolic expression `Particle::momentum().pz()` is used for evaluation
E	*F*	*O*	`LoKi::Particles::Energy`	Energy The symbolic expression `Particle::momentum().e()` is used for evaluation
M	*F*	*O*	`LoKi::Particles::Mass`	"Kinematical" mass: $\sqrt{E^2 - \vec{p}^2}$ The symbolic expression `Particle::momentum().m()` is used for evaluation
MM	*F*	*O*	`LoKi::Particles::MeasuredMass`	"Measured" mass The method `Particle::mass` used for evaluation

DMASS	*F*	*T*	`LoKi::Particles::DeltaMass`	ΔM
				The function can be constructed in many ways:
				`1`
				`2 Fun dm0 = DMASS ( 1.864 * GeV ) ;`
				`3 const ParticleProperty* pp = .. ;`
				`4 Fun dm1 = DMASS ( pp )`
				`5 Fun dm1 = DMASS ( "D0" ) ;`
				`6 Fun dm2 = DMASS ( "D0" , ppSvc() ) ;`
				`7 Fun dm3 = DMASS ( ParticleID ( 421 ) ) ;`
				`8 Fun dm4 = DMASS ( ParticleID ( 421 ) , ppSvc() ) ;`

(Continuation on the next page)

Table 13.2 *(Continuation)*

Function			The Actual type	Description
DMMASS	*F*	*T*	`LoKi::Particles::DeltaMeasuredMass`	ΔMM The function **MM** is used for mass evaluation. The function can be constructed in many ways - the set of constructors is similar to **DMASS** function.
ADMASS	*F*	*T*	`LoKi::Particles::AbsDeltaMass`	$\lvert\Delta\mathrm{M}\rvert$ The function can be constructed in many ways - the set of constructors is similar to **DMASS** function.
CHI2M	*F*	*T*	`LoKi::Particles::DeltaMeasuredMassChi2`	χ^2_{m} : The symbolic expression $(\Delta\mathrm{M}/\texttt{Particle::massErr}())^2$ is used for function evaluation

MASS	*F*	*T*	`LoKi::Particles::InvariantMass`	Invariant mass of variouse sub-combinations of daughter particles:
				`Fun m12 = MASS ( 1 , 2 ) ;`
				`Fun m123 = MASS ( 1 , 2 , 3 ) ;`
				`Fun m431 = MASS ( 4 , 3 , 1 ) ;`
				Index `0` is used as indicator for mother particle. For invalid indices the `HepLorentzVector()` is used for mass evaluation

M12	*F*	*O*	`LoKi::Particles::InvariantMass`	Mass of 1^{st} and 2^{nd} daughter particles:
				`const Particle* p = ... ;`
				`double m12 = M12 ( p ) ;`

M13	*F*	*O*	`LoKi::Particles::InvariantMass`	Mass of 1^{st} and 3^{rd} daughters
M14	*F*	*O*	`LoKi::Particles::InvariantMass`	Mass of 1^{st} and 4^{th} daughters
M23	*F*	*O*	`LoKi::Particles::InvariantMass`	Mass of 2^{nd} and 3^{rd} daughters
M24	*F*	*O*	`LoKi::Particles::InvariantMass`	Mass of 2^{nd} and 4^{th} daughters
M34	*F*	*O*	`LoKi::Particles::InvariantMass`	Mass of 3^{rd} and 4^{th} daughters

(Continuation on the next page)

Table 13.2 *(Continuation)*

Function			The Actual type	Description

LV0	*F*	*T*	`LoKi::Particles::RestFrameAngle`	Cosine of decay angle between momentum direction of daughter particle in the rest frame of mother particle and the direction of Lorentz boost from laboratoiry frame to mother particle rest frame
				`1`
				`2 // the first daughter particle to be used`
				`3 // for angle evaluation:`
				`4 Fun lv01 = LV0( 1 ) ;`

LV01	*F*	*O*	`LoKi::Particles::RestFrameAngle`	Cosine of decay angle between momentum direction of the first daughter particle in the rest frame of mother particle and the direction of Lorentz boost from laboratoiry frame to mother particle rest frame.
LV02	*F*	*O*	`LoKi::Particles::RestFrameAngle`	Cosine of decay angle between momentum direction of the second daughter particle in the rest frame of mother particle and the direction of Lorentz boost from laboratoiry frame to mother particle rest frame.
LV03	*F*	*O*	`LoKi::Particles::RestFrameAngle`	Cosine of decay angle between momentum direction of the third daughter particle in the rest frame of mother particle and the direction of Lorentz boost from laboratoiry frame to mother particle rest frame.
LV04	*F*	*O*	`LoKi::Particles::RestFrameAngle`	Cosine of decay angle between momentum direction of the fourth daughter particle in the rest frame of mother particle and the direction of Lorentz boost from laboratoiry frame to mother particle rest frame.

PID	*F*	*T*	`LoKi::Particles::ParticleIdEstimator`	The method to extract the detector PID value from `ProtoParticle` The method `ProtoParticle::detPIDvalue` is used for evaluation
				`1`
				`2 Fun richK = PID( ProtoParticles::RichKaon ) ;`
				`3 Fun shape = PID( ProtoParticle::ShowerShape ) ;`

(Continuation on the next page)

Table 13.2 *(Continuation)*

Function			The Actual type	Description

PIDe	*F*	*O*	`LoKi::Particles::ParticleIdEstimator`	Combined PID/DLL for $e^{\pm}$
				`1`
				`2 const Particle* p = ... ;`
				`3 const double eDLL = PIDe( p ) ;`

PIDmu	*F*	*O*	`LoKi::Particles::ParticleIdEstimator`	Combined PID/DLL for $\mu^{\pm}$
PIDpi	*F*	*O*	`LoKi::Particles::ParticleIdEstimator`	Combined PID/DLL for $\pi^{\pm}$
PIDK	*F*	*O*	`LoKi::Particles::ParticleIdEstimator`	Combined PID/DLL for $K^{\pm}$
PIDp	*F*	*O*	`LoKi::Particles::ParticleIdEstimator`	Combined PID/DLL for $p/\bar{p}$
TCHI2NDF	*F*	*O*	`LoKi::Particles::TrackChi2PerDoF`	χ^2/nDoF for tracks. The function uses `TrStoredTrack::lastChiSq` method and number of degrees of freedom evaluated from number of measurements
TRCHI2NDF	*F*	*O*	`LoKi::Particles::TrackChi2PerDoF`	Another name for **TCHI2NDF**
TLASTCHI2	*F*	*O*	`LoKi::Particles::TrackLastChi2`	χ^2 for tracks. The function relies on `TrStoredTrack::lastChiSq` method.
TRCHI2	*F*	*O*	`LoKi::Particles::TrackLastChi2`	Another name for **TLASTCHI2**
TRDOF	*F*	*O*	`LoKi::Particles::TrackDof`	nDoF for tracks. The symbolic formula `TrStoredTrack::measurements()-.size()-5` is used for evaluation

IP	*F*	*T*	`LoKi::Vertices::ImpPar`	Impact parameter for the particle with respect to the vertex or 3D-point
				`1`
				`2 const Vertex*  v = ... ;`
				`3 Fun ipV = IP ( point( v ) ) ;`
				`4 const HepPoint3D& p = ... ;`
				`5 Fun ipP = IP ( point( p ) ) ;`

(Continuation on the next page)

Table 13.2 *(Continuation)*

Function			The Actual type	Description

MIP	*F*	*T*	`LoKi::Vertices::MinImpPar`	Minimal impact parameter for the particle with respect to several vertices (e.g. all primary vertices)
				`1`
				`2 VRange pvs =`
				`3 vselect ( "PVs" , Vertex::Primary == VTYPE );`
				`4 Fun mip = MIP( pvs , geo() ) ;`

IPCHI2	*F*	*T*	`LoKi::Vertices::ImpParChi2`	Impact parameter χ^2 of the particle with respect to the vertex or 3D-point
				`1`
				`2 const Vertex* v = ... ;`
				`3 Fun ipV = IPCHI2 ( point( v ) ) ;`
				`4 const HepPoint3D& p = ... ;`
				`5 Fun ipP = IPCHI2 ( point( p ) ) ;`

CHI2IP	*F*	*T*	`LoKi::Vertices::ImpParChi2`	Another name for **IPCHI2**

MIPCHI2	*F*	*T*	`LoKi::Vertices::MinImpParChi2`	Minimal impact parameter χ^2 for the particle with respect to several vertices (e.g. all primary vertices)
				`1`
				`2 VRange pvs =`
				`3 vselect ( "PVs" , Vertex::Primary == VTYPE ) ;`
				`4 Range pions =`
				`5 select ( "pi+" ,`
				`6 "pi+" == ID && MIPCHI2( pvs , geo() ) > 4 ) ;`

VD	*F*	*T*	`LoKi::Vertices::VertexDistance`	distance between the decay vertex of the particle and vertex or 3D-point
				`1`
				`2 const Vertex* v = ... ;`
				`3 Fun dV = VD ( point( v ) ) ;`
				`4 const HepPoint3D& p = ... ;`
				`5 Fun dP = VD ( point( p ) ) ;`

(Continuation on the next page)

Table 13.2 *(Continuation)*

Function			The Actual type	Description
VDCHI2	*F*	*T*	`LoKi::Vertices::VertexDistanceChi2`	χ^2 distance between the decay vertex of the particle and vertex or 3D-point

```
1
2  const Vertex*      v = ... ;
3  Fun dV = VDCHI2 ( point( v ) ) ;
4  const HepPoint3D& p = ... ;
5  Fun dP = VDCHI2 ( point( p ) ) ;
```

CHI2VD	*F*	*T*	`LoKi::Vertices::VertexDistanceChi2`	Another name for **VDCHI2**
MVDCHI2	*F*	*T*	`LoKi::Vertices::MinChi2Distance`	Minimal χ^2 distance between the decay vertex of the particle and set of vertices (e.g. all primary vertices)

```
1
2  VRange pvs    =
3   vselect ( "PVs" , Vertex::Primary == VTYPE ) ;
4  Range  pions =
5   select ( "pi+" ,
6            "pi+" == ID && MVDCHI2( pvs ) > 4 ) ;
```

VDDOT	*F*	*T*	`LoKi::Vertices::VertexDistanceDot`	Distance between the decay vertex of the particle and vertex or 3D-point along the particle momentum direction: $\frac{(\Delta\vec{\mathbf{v}}\cdot\vec{\mathbf{p}})}{\lvert\vec{\mathbf{p}}\rvert}$

```
1
2  const Vertex*      v  = ... ;
3  Fun fV = VDDOT ( v ) ;
4  const HepPoint3D& p  = ... ;
5  Fun fP = VDDOT ( p ) ;
```

(Continuation on the next page)

Table 13.2 *(Continuation)*

Function			The Actual type	Description
VDSIGN	*F*	*T*	`LoKi::Vertices::SignedVertexDistance`	Signed distance between the decay vertex of the particle and vertex or 3D-point (as **VD**. The sign comes from the sign of Δz $$\left\|\Delta\vec{\mathbf{v}}\right\| \times \mathrm{sign}\left(\Delta\mathbf{z}\right)$$

```
1
2  const Vertex*      v   = ... ;
3  Fun fV = VDSIGN ( v ) ;
4  const HepPoint3D& p   = ... ;
5  Fun fP = VDSIGN ( p ) ;
```

VDTIME	*F*	*T*	`LoKi::Vertices::SignedTimeDistance`	Signed propertime ($c\tau$) of the particle evaluated from the distance between the decay vertex (**VDSIGN**) of the particle and vertex or 3D-point. The sign comes from the sign of Δz $$\left\|\Delta\vec{\mathbf{v}}\right\| \times \mathrm{sign}\left(\Delta\mathbf{z}\right) \frac{\mathbf{m}}{\left\|\vec{\mathbf{p}}\right\|}$$

```
1
2  const Vertex*      v   = ... ;
3  Fun fV = VDTIME ( v ) ;
4  const HepPoint3D& p   = ... ;
5  Fun fP = VDTIME ( p ) ;
```

(Continuation on the next page)

Table 13.2 *(Continuation)*

Function			The Actual type	Description
VDDTIME	*F*	*T*	`LoKi::Vertices::DotTimeDistance`	The propertime ($c\tau$) of the particle evaluated from the distance between the decay vertex of the particle and vertex or 3D-point along the particle momentum direction (**VDDOT**). $\frac{(\Delta\vec{\mathbf{v}}\cdot\vec{\mathbf{p}})}{\lvert\vec{\mathbf{p}}\rvert}\frac{\mathbf{m}}{\lvert\vec{\mathbf{p}}\rvert}$
PLTIME	*F*	*T*	`LoKi::Particles::LifeTime`	The propertime of the particle evaluated using `ILifetimeFitter` tool by Gerhard Raven
PLTERR	*F*	*T*	`LoKi::Particles::LifeTimeError`	The error in propertime of the particle evaluated using `ILifetimeFitter` tool by Gerhard Raven
PLTCHI2	*F*	*T*	`LoKi::Particles::LifeTimeChi2`	The propertime χ^2 of the particle evaluated using `ILifetimeFitter` tool by Gerhard Raven

```
1
2  const Vertex*      v   = ... ;
3  Fun fV = VDDTIME ( v ) ;
4  const HepPoint3D& p   = ... ;
5  Fun fP = VDDTIME ( p ) ;
```

```
1
2  ILifetimeFitter* fitter = lifetimeFitter() ;
3  const Vertex* primary = ... ;
4  Fun   time = PLTIME( fitter , vertex ) ;
```

```
1
2  ILifetimeFitter* fitter = lifetimeFitter() ;
3  const Vertex* primary = ... ;
4  Fun   terr = PLTERR( fitter , vertex ) ;
```

```
1
2  ILifetimeFitter* fitter = lifetimeFitter() ;
3  const Vertex* primary = ... ;
4  Fun   chi2 = PLTCHI2( fitter , vertex ) ;
```

(Continuation on the next page)

Table 13.2 *(Continuation)*

Function			The Actual type	Description
DTR	*F*	*T*	`LoKi::Vertices::ClosestApproach`	The distance of closest approach between the particle and another particle `1` `2` **const** Particle* particle = ... ; `3` Fun dist = DTR(particle) ;
DTRCHI2	*F*	*T*	`LoKi::Vertices::ClosestApproachChi2`	The χ^2 distance of closest approach between the particle and another particle `1` `2` **const** Particle* particle = ... ; `3` Fun chi2 = DTRCHI2(particle) ;
CHI2DTR	*F*	*T*	`LoKi::Vertices::ClosestApproachChi2`	Another name for **DTRCHI2**
DIRA	*F*	*T*	`LoKi::Vertices::DirectionAngle`	The cosine of the angle between particle momentum and the direction vector from the given vertex (e.g. primary vertex) or 3D-point to particle's decay vertex $\frac{(\Delta\vec{\mathbf{v}}\cdot\vec{\mathbf{p}})}{\lvert\vec{\mathbf{p}}\rvert\,\lvert\Delta\vec{\mathbf{v}}\rvert}$ `1` `2` **const** Vertex* v = ... ; `3` Fun aV = DIRA (v) ; `4` **const** HepPoin3D& p = ... ; `5` Fun aP = DIRA (p) ;
DDANG	*F*	*T*	`LoKi::Vertices::DirectionAngle`	Another name for **DIRA**

(Continuation on the next page)

Table 13.2 *(Continuation)*

Function			The Actual type	Description

CHILD	*F*	*T*	`LoKi::Particles::ChildFunction`	Adaptor function, which delegates the actual function to one of the daughter particles
				`const Particle* Ks = ... ;`
				`// evaluator of transverse momemntum`
				`// of the first daughter particle`
				`Fun pt1 = CHILD( PT , 1 ) ;`

VXFUN	*F*	*T*	`LoKi::Adapters::VFuncAsFun`	Adaptor function which allows for coherent usage of Particle and Vertex functions

FILTER	*F*	*T*	`LoKi::Adapters::FilterAsCut`	Adaptor function which allows to use `IFilterCriterion` tools as LoKi's predicate
				`IFilterCriterion* criterion = ... ;`
				`Cut cut = FILTER( criterion ) ;`

FROMTREE	*P*	*T*	`LoKi::Particles::FromTree`	Predicate which allows to detect mother-daugher relations between particles
NOOVERLAP	*P*	*T*	`LoKi::Particles::NoOverlap`	Predicate which detects "overlap" between the particles

TRFLAG	*P*	*T*	`LoKi::Particles::TrackFlags`	Predicate which provides access to variouse "boolean" methods of class `TrStoredTrack`
				`Cut b1 = TRFLAGS( LoKi::Tracks::IsLong ) ;`
				`Cut b2 = TRFLAGS( LoKi::Tracks::Follow ) ;`
				`Cut b3 = TRFLAGS( LoKi::Tracks::KsTrack ) ;`

TRISLONG	*P*	*O*	`LoKi::Particles::TrackFlags`	Evaluates to `true` for basic particle, constructed from "long"-track The method `TrStoredTrack::isLong` is used
				`const Particle* p = ... ;`
				`const bool = TRISLONG( p ) ;`

(Continuation on the next page)

Table 13.2 *(Continuation)*

Function			The Actual type	Description
TRISUP	*P*	*O*	`LoKi::Particles::TrackFlags`	Evaluates to `true` for basic particle, constructed from "upstream"-track The method `TrStoredTrack::isUpstream` is used
TRISDOWN	*P*	*O*	`LoKi::Particles::TrackFlags`	Evaluates to `true` for basic particle, constructed from "downstream"-track The method `TrStoredTrack::isDownstream` is used
TRISVELO	*P*	*O*	`LoKi::Particles::TrackFlags`	Evaluates to `true` for basic particle, constructed from "velo"-track The method `TrStoredTrack::isVelotrack` is used
TRISBACK	*P*	*O*	`LoKi::Particles::TrackFlags`	Evaluates to `true` for basic particle, constructed from "backward"-track The method `TrStoredTrack::isBackward` is used
TRIST	*P*	*O*	`LoKi::Particles::TrackFlags`	Evaluates to `true` for basic particle, constructed from "T"-track The method `TrStoredTrack::isTtrack` is used
TRUNIQUE	*P*	*O*	`LoKi::Particles::TrackFlags`	Evaluates to `true` for basic particle, constructed from "unique"-track The method `TrStoredTrack::unique` is used
TRVELO	*P*	*O*	`LoKi::Particles::TrackFlags`	Evaluates to `true` for basic particle, constructed from "velo"-track The method `TrStoredTrack::velo` is used
TRSEED	*P*	*O*	`LoKi::Particles::TrackFlags`	Evaluates to `true` for basic particle, constructed from "seed"-track The method `TrStoredTrack::seed` is used
TRMATCH	*P*	*O*	`LoKi::Particles::TrackFlags`	Evaluates to `true` for basic particle, constructed from "match"-track The method `TrStoredTrack::match` is used
TRFORWARD	*P*	*O*	`LoKi::Particles::TrackFlags`	Evaluates to `true` for basic particle, constructed from "forward"-track The method `TrStoredTrack::forward` is used
TRFOLLOW	*P*	*O*	`LoKi::Particles::TrackFlags`	Evaluates to `true` for basic particle, constructed from "follow"-track The method `TrStoredTrack::follow` is used
TRVELOTT	*P*	*O*	`LoKi::Particles::TrackFlags`	Evaluates to `true` for basic particle, constructed from "velo-TT"-track The method `TrStoredTrack::veloTT` is used

(Continuation on the next page)

Table 13.2 *(Continuation)*

Function			The Actual type	Description
TRVELOBACK	*P*	*O*	`LoKi::Particles::TrackFlags`	Evaluates to `true` for basic particle, constructed from "velo-back"-track The method `TrStoredTrack::veloBack` is used
TRKSTRACK	*P*	*O*	`LoKi::Particles::TrackFlags`	Evaluates to `true` for basic particle, constructed from "Ks"-track The method `TrStoredTrack::ksTrack` is used

TRALLFLAG	*P*	*O*	`LoKi::Particles::AllTrackFlags`	Evaluates to `true` for particle, constructed from tracks with the same 'flag'. E.g. "LL"-category of K^0_S can be selected with followiing predicate:
				`1`
				`2 Cut ll = "KS0" == ID &&`
				`3 TRALLFLAG( LoKi::Tracks::IsLong ) ;`

MINTREE	*F*	*T*	`LoKi::TreeFunctions::MinTree`	Meta-function which evaluates to a minimal value of another function over the decay tree of the particle.
				E.g. the minimal value for kaon identification variable for all positive kaons can be obtained using the construction:
				`1`
				`2 Fun kid = MINTREE( PIDK , "K+" == ID ) ;`

MAXTREE	*F*	*T*	`LoKi::TreeFunctions::MaxTree`	Meta-function which evaluates to a maximal value of another function over the decay tree of the particle (similar to **MINTREE**)

TREEMULT	*F*	*T*	`LoKi::TreeFunctions::AccTreeMult`	Meta-function which accumulates (by multiplication) the values of another function over the decay tree of the particle.
				E.g. the product of all confidence levels for basic particles can be obtained using the construction:
				`1`
				`2 Fun kid = TREEMULT( CL , HASPROTOP ) ;`

(Continuation on the next page)

Table 13.2 *(Continuation)*

Function			The Actual type	Description
TREESUM	*F*	*T*	`LoKi::TreeFunctions::AccTreeMult`	Meta-function which accumulates (by summing) the values of another function over the decay tree of the particle E.g. the sum of all tarck chi^2's for basic particles can be obtained using the construction: `Fun kid = TREESUM( TRCHI2 , HASTRACK ) ;`
NINTREE	*F*	*T*	`LoKi::TreeFunctions::NinTree`	Meta-function which evaluates to a number of particles, fot those the given predicate is fulfi lled in the decay tree of the particle. E.g. number of "long" tracks in the decay tree `Fun nLong = NINTREE( TRISLONG ) ;`

INTREE	*P*	*T*	`LoKi::TreeFunctions::NinTree`	Meta-predicate which evaluates to `true` if there exist at least one particle whcih satisfy the certain predciate in the decay tree of the particle. E.g. one can evaluate if at least one treack from the given B-candidate has been used fro Primary Vertex reconstruction:
				`VRange pvs = vselect ( "PVs" , Vertex::Primary == VTYPE ) ;`
				`Cut used = INTREE( INPVFIT( pvs ) ) ;`

HASORIGIN	*P*	*O*	`LoKi::Particles::HasOrigin`	The predicate which evaluates to `true` for "basic" particles The symbolic expression `0 != Particle::origin()` is used for evaluation
HASPROTOP	*P*	*O*	`LoKi::Particles::HasProtoParticle`	The predicate which evaluates to `true` for "basic" particles, created from `ProtoParticle`
HASTRACK	*P*	*O*	`LoKi::Particles::HasTrack`	The predicate which evaluates to `true` for "basic" particles, created from `TrStoredTrack`
INPVFIT	*P*	*T*	`LoKi::Particles::UsedForPrimaryVertex`	The predicate which evaluates to `true` for the particles used for primary vertex reconstruction

(Continuation on the next page)

Table 13.2 *(Continuation)*

Function			The Actual type	Description
PIDAGREE	*P*	*T*	`LoKi::Particles::PPIsPIDCompatible`	The predicate which evaluates to `true` for the basic particles, which are in agreement with particle identification hypothesis The result from the method `ProtoParticle::pIDList()` is used
BESTPID	*F*	*O*	`LoKi::Particles::PPbestPID`	The best PID for the basic particle The method `ProtoParticle::bestPID()` is used
BESTPIDPR	*F*	*O*	`LoKi::Particles::PPbestPID`	Probabilty of the best PID for the basic particle The method `ProtoParticle::probOfBestPID()` is used

PIDBITS	*P*	*T*	`LoKi::Particles::PPbitsPID`	The predicate which provides access to variouse "boolean" methods of class `ProtoParticle`
				`1`
				`2 Cut c1 = PIDBITS( LoKi::ProtoParticles::NoneBit ) ;`
				`3 Cut c2 = PIDBITS( LoKi::ProtoParticles::RichBit ) ;`
				`4 Cut c3 = PIDBITS( LoKi::ProtoParticles::HasRich ) ;`

PPBITNONE	*P*	*O*	`LoKi::Particles::PPbitsPID`	The predicate which evaluates the method `ProtoParticle::noneBit()` for "basic" particles
				`1`
				`2 const Particle* p = ... ;`
				`3 const bool res = PPBITNONE( p ) ;`

PPBITRICH	*P*	*O*	`LoKi::Particles::PPbitsPID`	The predicate which evaluates the method `ProtoParticle::richBit()` for "basic" particles (similar to **PPBININONE**)
PPBITMUON	*P*	*O*	`LoKi::Particles::PPbitsPID`	The predicate which evaluates the method `ProtoParticle::muonBit()` for "basic" particles (similar to **PPBININONE**)
PPBITCALO	*P*	*O*	`LoKi::Particles::PPbitsPID`	The predicate which evaluates the method `ProtoParticle::caloeBit()` for "basic" particles (similar to **PPBININONE**)

(Continuation on the next page)

Table 13.2 *(Continuation)*

Function			The Actual type	Description
PPRICHCOMB	*P*	*O*	`LoKi::Particles::PPbitsPID`	The predicate which evaluates the method `ProtoParticle::richCombined()` for "basic" particles (similar to **`PPBININONE`**)
PPMUONCOMB	*P*	*O*	`LoKi::Particles::PPbitsPID`	The predicate which evaluates the method `ProtoParticle::muonCombined()` for "basic" particles (similar to **`PPBININONE`**
PPCALOCOMB	*P*	*O*	`LoKi::Particles::PPbitsPID`	The predicate which evaluates the method `ProtoParticle::caloeCombined()` for "basic" particles (similar to **`PPBININONE`**
PPHASRICH	*P*	*O*	`LoKi::Particles::PPbitsPID`	The predicate whcih check the availability of existence of `RichPID` object for the given particle. The symbolic expression `0 != ProtoParticle::richPID()` is used for evaluation
PPHASMUON	*P*	*O*	`LoKi::Particles::PPbitsPID`	The predicate whcih check the availability of existence of `MuonID` object for the given particle. The symbolic expression `0 != ProtoParticle::muonPID()` is used for evaluation
PPHASCALO	*P*	*O*	`LoKi::Particles::PPbitsPID`	The predicate whcih check the availability of existence of `CaloHypo` objects for the given particle. The symbolic expression `!ProtoParticle::calo().empty` is used for evaluation
MCTRUTH	*P*	*T*	`LoKi::Particles::MCTruth`	Simple predicate which evaluated to **`true`** for particles , which "has MC -matching" with certain Monte Carlo particles

```
1
2  const Particle* p = ... ;
3  MCRange mcps = .... ;
4  Cut cut = MCTRUTH( mctruth() , particles ) ;
5  const bool fromMC = cut( p ) ;
```

Table 13.3: Predefined LoKi *Vertex functions*. All *functions* are defined in header file `LoKi/VertexCuts.h` inside the namespace `LoKi::Cuts`

Function			The Actual type	Description
VONE	*F*	*O*	`LoKi::Constant<const Vertex*>`	Function which always evaluates to **1**
VZERO	*F*	*O*	`LoKi::Constant<const Vertex*>`	Function which always evaluates to **0**
VALL	*P*	*O*	`LoKi::BooleanConstant<const Vertex*>`	Predicate which always evaluates to **true**
VTRUE	*P*	*O*	`LoKi::BooleanConstant<const Vertex*>`	Another name for **VALL**
VNONE	*P*	*O*	`LoKi::BooleanConstant<const Vertex*>`	Predicate which always evaluates to **false**
VFALSE	*P*	*O*	`LoKi::BooleanConstant<const Vertex*>`	Another name for **VNONE**
VMIN	*F*	*T*	`LoKi::Min<const Vertex*>`	Function which evaluates the minimum value from few other functions
VMAX	*F*	*T*	`LoKi::Max<const Vertex*>`	Function which evaluates the maximum value from few other functions (simlar to **VMIN**)

VCHI2	*F*	*O*	`LoKi::Vertices::VertexChi2`	The vertex χ^2. The function `Vertex::chi2` is used for evaluation
				`// get the vertex`
				`const Vertex* vertex = ... ;`
				`const double chi2 = VCHI2( vertex ) ;`

VCHI2	*F*	*O*	`LoKi::Vertices::VertexChi2`	Another name for **CHI2V**

VTYPE	*F*	*O*	`LoKi::Vertices::VertexType`	The vertex type. The function `Vertex::type` is used for evaluation
				`// get the vertex`
				`const Vertex* vertex = ... ;`
				`const double type = VTYPE( vertex ) ;`

(Continuation on the next page)

Table 13.3 *(Continuation)*

Function			The Actual type	Description

VX	*F*	*O*	`LoKi::Vertices::VertexX`	The X-position of the vertex. The symbolic expression `Vertex::position().x()` is used for evaluation
				`// get the vertex`
				`const Vertex* vertex = ... ;`
				`const double x = VX( vertex ) ;`

VY	*F*	*O*	`LoKi::Vertices::VertexY`	The Y-position of the vertex. The symbolic expression `Vertex::position().y()` is used for evaluation
				`// get the vertex`
				`const Vertex* vertex = ... ;`
				`const double y = VY( vertex ) ;`

VZ	*F*	*O*	`LoKi::Vertices::VertexZ`	The Z-position of the vertex. The symbolic expression `Vertex::position().z()` is used for evaluation
				`// get the vertex`
				`const Vertex* vertex = ... ;`
				`const double z = VZ( vertex ) ;`

VPRONGS	*F*	*O*	`LoKi::Vertices::VertexProngs`	The multiplicity of Particles in the vertex. The symbolic expression `Vertex::products().size()` is used for evaluation
				`// get the vertex`
				`const Vertex* vertex = ... ;`
				`const double n = VPRONGS( vertex ) ;`

VTRACKS	*F*	*O*	`LoKi::Vertices::VertexTracks`	The multiplicity of TrStoredTrack in the primary vertex. The symbolic expression `PrimVertex::track().size()` is used for evaluation
				`// get the vertex`
				`const Vertex* vertex = ... ;`
				`const double tracks = VTRACKS( vertex ) ;`

(Continuation on the next page)

Table 13.3 *(Continuation)*

Function			The Actual type	Description
VCHI2D	*F*	*T*	`LoKi::Vertices::VertexChi2Distance`	The χ^2 distance between the vertex and another vertex or 3D-point

```
// get the point or (primary) vertex
const HepPoint3D& point    = ... ;
const Vertex*     primvx   = ... ;
// create functor(s)
VFun fun1 = VCHI2D( point  ) ;
VFun fun2 = VCHI2D( primvx ) ;
// use functors
const Vertex* vertex = ... ;
// evaluate the chi2 distances
const double d1 = fun1( vertex ) ;
const double d2 = fun2( vertex ) ;
```

VMCHI2D	*F*	*T*	`LoKi::Vertices::MinVertexChi2Distance`	Evaluator for minimal chi2 distance between the vertex and all primary vertices

```
// get all primary vertices:
VRange primaries =  vselect("Primaries" ,
                    VTYPE == Vertex::Primary )
// create functor
VFun fun = VMCHI2D( primaries ) ;
const Vertex* vertex = ... ;
// evaluate the minimal chi2
const double minChi2 = fun ( vertex ) ;
```

VIP	*F*	*T*	`LoKi::Vertices::VertexImpPar`	The vertex impact parameter with respect to a given particle (it is "reverse" to **IP**)

```
// get some particle , e.g B
const Particle* particle = ... ;
// create vertex function
VFun vip = VIP ( particle , geo() );
const Vertex* primary = ... ;
// use the function
const double vrtxImpPar = vip( primary );
```

(Continuation on the next page)

Table 13.3 *(Continuation)*

Function			The Actual type	Description
VIPCHI2	*F*	*T*	`LoKi::Vertices::VertexImpParChi2`	The vertex impact parameter $\chi^2$with respect to a given particle (it is "reverse" to **IPCHI2**)

```
// get some particle , e.g B
const Particle* particle = ... ;
// create vertex function
VFun vip = VIPCHI2 ( particle , geo() );
const Vertex* primary = ... ;
// use the function
const double vrtxImpParChi2 = vip( primary );
```

CHI2VIP	*F*	*T*	`LoKi::Vertices::VertexImpParChi2`	Another name for **VIPCHI2**
VPSD	*F*	*T*	`LoKi::Vertices::VertexParticleSignedDistance`	The signed distance between the decay vertex of the particle and the given vertex

```
// some particle
const Particle* B = ... ;
// create the vertex function
VFun vpsd = VPSD( B , geo() ) ;
// use created function
const Vertex* myvx = ... ;
double distance = vpsd( myvx );
```

Table 13.4: Predefined LOKI *MCParticle functions*. All *functions* are defined in header file `LoKi/MCParticleCuts.h` inside the namespace `LoKi::Cuts`

Function			The Actual type	Description
MCONE	*F*	*O*	`LoKi::Constant<const MCParticle*>`	Function which always evaluates to **1**
MCZERO	*F*	*O*	`LoKi::Constant<const MCParticle*>`	Function which always evaluates to **0**
MCALL	*P*	*O*	`LoKi::BooleanConstant<const MCParticle*>`	Predicate which always evaluates to **true**
MCTRUE	*P*	*O*	`LoKi::BooleanConstant<const MCParticle*>`	Another name for **MCALL**
MCNONE	*P*	*O*	`LoKi::BooleanConstant<const MCParticle*>`	Predicate which always evaluates to **false**
MCFALSE	*P*	*O*	`LoKi::BooleanConstant<const MCParticle*>`	Another name for **MCNONE**
MCMIN	*F*	*T*	`LoKi::Min<const MCParticle*>`	Function which evaluates the minimum value from few other functions
MCMAX	*F*	*T*	`LoKi::Max<const MCParticle*>`	Function which evaluates the maximum value from few other functions (simlar to **MCMIN**)
MCP	*F*	*O*	`LoKi::MCParticles::Momentum`	The momentum of Monte Carlo particle. The symbolic expression `MCParticle::momentum().vect().mag()` is used for evaluation
MCE	*F*	*O*	`LoKi::MCParticles::Energy`	The energy of Monte Carlo particle. The symbolic expression `MCParticle::momentum().e()` is used for evaluation
MCPT	*F*	*O*	`LoKi::MCParticles::TransverseMomentum`	The transverse momentum of Monte Carlo particle. The symbolic expression `MCParticle::momentum().pt()` is used for evaluation
MCPX	*F*	*O*	`LoKi::MCParticles::MomentumX`	X-component of Monte Carlo particle's momentum. The symbolic expression `MCParticle::momentum().px()` is used for evaluation

(Continuation on the next page)

Table 13.4 *(Continuation)*

Function			The Actual type	Description
MCPY	*F*	*O*	`LoKi::MCParticles::MomentumY`	Y-component of Monte Carlo particle's momentum. The symbolic expression `MCParticle::momentum().py()` is used for evaluation
MCPZ	*F*	*O*	`LoKi::MCParticles::MomentumZ`	Z-component of Monte Carlo particle's momentum. The symbolic expression `MCParticle::momentum().pz()` is used for evaluation
MCMASS	*F*	*O*	`LoKi::MCParticles::Mass`	The mass of Monte Carlo particle. The symbolic expression `MCParticle::momentum().m()` is used for evaluation
MCID	*F*	*O*	`LoKi::MCParticles::Identifier`	The identifier of Monte Carlo particle. The symbolic expression `MCParticle::particleID().pid()` is used for evaluation
MCABSID	*F*	*O*	`LoKi::MCParticles::AbsIdentifier`	The absolute value of identifier of Monte Carlo particle. The symbolic expression `MCParticle::particleID().abspid()` is used for evaluation
MC3Q	*F*	*O*	`LoKi::MCParticles::ThreeCharge`	The charge (multiplied by 3) of Monte Carlo particle. The symbolic expression `MCParticle::particleID().threeCharge()` is used for evaluation
MCTIME	*F*	*T*	`LoKi::MCParticles::ProperLifeTime`	The proper lifetime of of Monte Carlo particle. The distance between *origin vertex* and *the first nd-vertex* is used for distance evaluation.
CHARGED	*P*	*O*	`LoKi::MCParticles::IsCharged`	Simple predicate which evaluated to **`true`** for charged Monte Carlo particles. The symbolic expression `0 != MCParticle::particleID().threeCharge()` is used for evaluation.

(Continuation on the next page)

Table 13.4 *(Continuation)*

Function			The Actual type	Description
NEUTRAL	*P*	*O*	`LoKi::MCParticles::IsNeutral`	Simple predicate which evaluated to **`true`** for neutral Monte Carlo particles. The symbolic expression `0 == MCParticle::particleID().threeCharge()` is used for evaluation.
LEPTON	*P*	*O*	`LoKi::MCParticles::IsLepton`	Simple predicate which evaluated to **`true`** for Monte Carlo leptons. The symbolic expression `MCParticle::particleID().isLepton()` is used for evaluation.
MESON	*P*	*O*	`LoKi::MCParticles::IsMeson`	Simple predicate which evaluated to **`true`** for Monte Carlo mesons. The symbolic expression `MCParticle::particleID().isMeson()` is used for evaluation.
BARYON	*P*	*O*	`LoKi::MCParticles::IsBaryon`	Simple predicate which evaluated to **`true`** for Monte Carlo baryons. The symbolic expression `MCParticle::particleID().isBaryon()` is used for evaluation.
HADRON	*P*	*O*	`LoKi::MCParticles::IsHadron`	Simple predicate which evaluated to **`true`** for Monte Carlo baryons. The symbolic expression `MCParticle::particleID().isHadron()` is used for evaluation.
NUCLEUS	*P*	*O*	`LoKi::MCParticles::IsNucleus`	Simple predicate which evaluated to **`true`** for Monte Carlo nuclea. The symbolic expression `MCParticle::particleID().isNucleus()` is used for evaluation.

(Continuation on the next page)

Table 13.4 *(Continuation)*

Function			The Actual type	Description

MCQUARK	*P*	*T*	`LoKi::MCParticles::HasQuark`	Simple predicate which test the presence of quark of given type in the MC particle.
				The quark identified need to be specified in the constructor.
				`MCCut hasStrange = MCQUARK( ParticleID::strange ) ;`
				`MCCut hasBottom = MCQUARK( ParticleID::bottom ) ;`

BEAUTY	*P*	*O*	`LoKi::MCParticles::HasQuark`	Simple predicate which evaluated to **`true`** for Monte Carlo beauty particles.
CHARM	*P*	*O*	`LoKi::MCParticles::HasQuark`	Simple predicate which evaluated to **`true`** for Monte Carlo charm particles.
STRANGE	*P*	*O*	`LoKi::MCParticles::HasQuark`	Simple predicate which evaluated to **`true`** for Monte Carlo strange particles.

MCMOTHER	*F*	*T*	`LoKi::MCParticles::MCMotherFunctionk`	Simple function which evaluates another function for "mother" Monte Carlo particle .
				`const MCParticle* p = ... ;`
				`MCFun fun = MCMOTHER( MCABSID , 0 ) ;`
				`const double mcMothID = fun( p ) ;`

FROMMCTREE	*P*	*T*	`LoKi::MCParticles::FromMCDecayTree`	Simple predicate wich evaluates if Monte Carlo particle comes from decay tree of (an)other Monte Carlo particle(s)
				`const MCParticle* p = ... ;`
				`const MCParticle* parent = ... ;`
				`MCCut cut = FROMMCTREE( parent ) ;`
				`const bool fromTree = cut( p ) ;`

(Continuation on the next page)

Table 13.4 *(Continuation)*

Function			The Actual type	Description

NINMCDOWN	*F*	*T*	`LoKi::MCParticles::NinMCdownTree`	Simple predicate which evaluated to **`true`** for Monte Carlo strange particles.
				`1`
				`2 const MCParticle* mcp = ... ;`
				`3 // number of beauty baryons in the decay tree`
				`4 MCFun fun = NIMCTDOWN( BEAYTY && BARYON ) ;`
				`5 const bool ok = 0 != fun( mcp ) ;`

RCTRUTH	*P*	*T*	`LoKi::MCParticles::RCTruth`	Simple predicate which evaluated to **`true`** for Monte Carlo, which "has MC -matching" with certain reconstructed particle
				`1`
				`2 const MCParticle* mcp = ... ;`
				`3 Range particles = .... ;`
				`4 MCCut cut = RCTRUTH( mctruth() , particles ) ;`
				`5 const bool fromRC = cut( mcp ) ;`

Table 13.5: Predefined LoKi *MCVertex functions*. All *functions* are defined in header file `LoKi/MCVertexCuts.h` inside the namespace `LoKi::Cuts`

Function			The Actual type	Description
MCVONE	*F*	*O*	`LoKi::Constant<const MCVertex*>`	Function which always evaluates to **`1`**
MCVZERO	*F*	*O*	`LoKi::Constant<const MCVertex*>`	Function which always evaluates to **`0`**
MCVALL	*P*	*O*	`LoKi::BooleanConstant<const MCVertex*>`	Predicate which always evaluates to **`true`**
MCVTRUE	*P*	*O*	`LoKi::BooleanConstant<const MCVertex*>`	Another name for **MCVALL**
MCVNONE	*P*	*O*	`LoKi::BooleanConstant<const MCVertex*>`	Predicate which always evaluates to **`false`**
MCVFALSE	*P*	*O*	`LoKi::BooleanConstant<const MCVertex*>`	Another name for **MCVNONE**
MCVMIN	*F*	*T*	`LoKi::Min<const MCVertex*>`	Function which evaluates the minimum value from few other functions
MCVMAX	*F*	*T*	`LoKi::Max<const MCVertex*>`	Function which evaluates the maximum value from few other functions (simlar to **MCVMIN**)
MCVTYPE	*F*	*O*	`LoKi::MCVertices::TypeOfMCVertex`	Function which returns the Monte Carlo Vertex type. The method `MCVertex::type()` is used for evaluation.
MCVX	*F*	*O*	`LoKi::MCVertices::VertexPositionX`	X-position of Monte Carlo vertex
MCVY	*F*	*O*	`LoKi::MCVertices::VertexPositionY`	Y-position of Monte Carlo vertex
MCVZ	*F*	*O*	`LoKi::MCVertices::VertexPositionZ`	Z-position of Monte Carlo vertex
MCVTOF	*F*	*O*	`LoKi::MCVertices::TimeOfFlight`	Function which evaluates the time of flight for Monte Carlo vertex. The method `MCVertex::timeOfFlight()` is used for evaluation.
MCVDIST	*F*	*T*	`LoKi::MCVertices::MCVertexDistance`	Function which evaluates distance he distance for MC vertex from given point.

Chapter 14: *Cuts*/*Predicates*

14.1 More about *Cuts*/*Predicates*

The logical operations with *funtions* result in *cuts*. *Cuts* are naturally subdivided to the *particle cuts* and *vertex cuts*. The former can be applied for objects of type **`const Particle*`** and the latter can be applied to the objects of type **`const Vertex*`**. Each *particle cut* can be directly assigned to the object of type **`Cut`** and each *vertex cut* can be directly assigned to the object of type **`VCut`**.
Since the objects of type **`Loop`** are implicitely convertable both to **`const Particle*`** and **`const Vertex*`** types, all LOKI *functions* can be directly applied to **`Loop`** objects:

```
1
2       Cut  cut  = abs( DMASS( 1.864 * GeV ) ) < 30 MeV ;
3       VCut vcut = VCHI2 / VDOF < 10  ;
4       for( Loop D0 = loop( "K- pi+", "D0" ) ; D0 ; ++D0 )
5       {
6         if( cut( D0 ) && vcut( D0 ) ) { D0->save("D0");}
7       }
```

LOKI *cuts* can be combined together using the basic logical operations '||' and '&&'.

```
1
2       Cut   cut1  = abs( DMASS( 1.864 * GeV ) ) < 30 MeV ;
3       Cut   cut2  = cut1 && ( PT > 100 * MeV || P > 10 * GeV );
4       Range D0s   = select ("D0s" , abs( ID ) == 241 && cut1 && cut2 );
```

All LOKI *cuts*/*predicates* fulfill all STL requirements for *functors* and therefore they effectively can be uses in a conjunction with all standard STL *algorithms* and *containers*:

```
1
2       /// arbitrary sequence
3       typedef std::vector<const Particle*> Container ;
4
5       Cut   cut = ( PT > 100 * MeV || P > 10 * GeV ) ;
6
7       Container input = .... ;
8       Container ouput ;
9
10      /// use standard algorithm from STL
11      std:copy_if( input.begin () ,   // begin of input sequence
12                   input.end   () ,   // end of input sequence
13                   std::back_inserter( output ) , // output
14                   cut            ) ; // cut/predicate
```

Here the particles which satisfy cut criteria are copied from the container **`input`** to the container **`output`**.

Chapter 15: Confi guration of MC truth matching

Of course LOKI itself does not perform any correspondence between reconstructed particles and their Monte Carlo truth information. Here LOKI relies on existing *associators* provided by DAVINCIASSOCIATORS package. LOKI uses all avilable *associators* to extract the MC matching information from them. LOKI is able to work with *accosiators*, which implement following abstract interfaces from RELATIONS subpackage of LHCBKERNEL package:

- **`IAssociatorWeighted<Particle,MCParticle,double>`**
- **`IAssociator<Particle,MCParticle>`**
- **`IAssociatorWeighted<ProtoParticle,MCParticle,double>`**

LOKI can accept multiple associators with the same interface. The example is the separate associators for neutral and charged protoparticles. Another example is the associator for particles wich uses χ^2 and MC links, or associator for composed particles. All these associators are available in DAVINCIASSOCIATORS package.

For matching between reconstructed particle and MC particle LOKI uses following strategy:

1. Scans over all available associators which implement **`IAssociatorWeighted<Particle,MCParticle>`** interface. If the pair is found in relation table, LOKI decides that the reconstructed particle and the Monte Carlo particle are matched.
2. Otherwise LOKI scans over all associators which implement **`IAssociator<Particle,MCParticle,double>`** interface. If the pair is found in relation table, LOKI decides that the reconstructed particle and the Monte Carlo particles are matched.
3. The further LOKI's action depends on the particle type.
   - For "basic" particles, which have the protoparticle as their originator
     (a) LOKI scans through all associators of type **`IAssociatorWeighted<ProtoParticle,MCParticle,double>`**. If the pair is found in the relation table LOKI decides that the particle and the Monte Carlo particle are matched.
     (b) Otherwise, LOKI expands the list of all daughter Monte Carlo particles for the given MC particle, checks whether there is MC match between the particle with any daughter Monte Carlo particles for a given Monte Carlo mother particle. It is a recursive procedure.
   - For "composite" particles
     – LOKI asks for MC matching of all daughter reconstructed particles for given composite particle with at least one MC particle from expanded list of all daughter Monte Carlo particles for a given Monte Carlo mother. Again it is recursive procedure.

The configuration of Monte Carlo matching capabilities of LoKi package is under the control by properties of **`Algo`** base class for each algorithm.

For future LoKi development it is foreseen the configuration of Monte Carlo matching for each `MCMatch` instance.

List of Tables

Bibliography

[1] S. Amato *et al.*, **CERN/LHCC 98-4**

[2] BRUNEL project
`http://cern.ch/lhcb-comp/Reconstruction`

[3] GAUSS project
`http://cern.ch/lhcb-comp/Simulation`

[4] PANORAMIX project
`http://cern.ch/lhcb-comp/Frameworks/Visualization`

[5] DAVINCI project
`http://cern.ch/lhcb-comp/Analysis/DaVinci`

[6] GAUDI project
`http://cern.ch/Gaudi`

[7] T. Glebe, “GCombiner 1.0”, HERA-B Note 01-001;
T. Glebe, “GCombiner”, HERA-B note 01-002

[8] T. Glebe, “Pattern - High Level Tools for Data Analysis”, HERA-B Note 02-002

[9] Clhep - A Class Library for High Energy Physics,
`http://cern.ch/clhep`

[10] A. Alexandrescu, “Modern C++ Design”,.... ;

[11] AIDA project
`http://aida.freehep.org`

Index