LHCb note 2007-037

The Tsa Reconstruction Framework

M. Needham
CERN

March 30, 2007

Abstract

The classes comprising the Tsa reconstruction framework at the time of Brunel v31r1 are described together with their use.

1 Introduction

The development of pattern recognition algorithms is a difficult task. *A priori*, it is not clear that the chosen approach will work. There is a clear danger that during the development that frustration and pessimism sets in as unforeseen features are discovered. It is the task of the framework to provide tools and components that allow the user to prototype new ideas quickly and hence shorten the design process. In addition, pattern recognition problems are combinatoric in nature. For example in the development of the track seeding [1, 2] it has been observed that seemingly small changes in the way loops are implemented can have an order of magnitude effect on code speed. Ideas should not be rejected early in the development phase because the combinatorics looked intractable when with minor modifications the speed of the code can easily be improved. On the other hand the user may try to solve the combinatoric problems by techniques such as early breaking out of loops. In problems where loops are heavily nested this leads to code that is both difficult to understand and to maintain. It is the role of the framework to ensure the user can develop fast but maintainable code.

The Tsa framework [3] was developed with these ideas in mind. The performance of algorithms developed within this framework can be found elsewhere [1, 2, 3]. In this note the framework at the time of Brunel v31r1 is described. The purpose of the note is two-fold. First, to document the use of the code at the time of this version of Brunel. Second, to present the ideas that have driven the framework development. Many of these ideas have their origins elsewhere [4, 5, 6, 7]. The success of the Tsa framework in developing fast and efficient code shows that these idioms are applicable to the development of tracking software.

The note is organized as follows. First, the packaging, and structure of the released code is discussed. This is sub-divided into two parts. The first is a discussion of the core Tsa classes. The second is a discussion of the so called Seed extension. Finally, the results of metric tests on the code are described.

The framework should not be considered either finished or perfect. Therefore, features which can be improved or where more development is needed will be pointed out.

2 Core Tsa

In the following sections the packaging and use of the core Tsa data classes and tools will be described. Further details can found on the LHCb doxygen webpages [8].

2.1 Packaging

The framework consists of two packages under the Tr "hat". The first TsaKernel is a shared library consisting of data classes, free functions and interfaces. Many of the classes in this library can be re-used outside the Tsa framework. Currently, it is located in the LHCb project. The second package is TsaAlgorithms. This consists of component classes: tools and algorithms. It is located in the REC project. The packages are included in a cmt requirements file as follows:

```
use TsaKernel v* Tr
use TsaAlgorithms v* Tr
```

All the classes in the TsaKernel package are located in the Tsa namespace. This leads to more natural names for the data classes. In the future it is envisaged that algorithms and tools will also be moved to this namespace.

2.2 Data classes

The main data classes provided by the framework are cluster classes. Fig. 1 shows the class structure for the cluster classes. A base class **Cluster** expresses the common functionality between the detectors. Concrete implementations for the Silicon and Outer Tracker inherit from this class. The base class derives from **ContainedObject** [9] and it expected that objects of this type are created with **new** and stored in **ObjectVectors**. To remove any time penalty from **new** these objects are allocated using a Boost memory pool [10].

The **Tsa::Cluster** class contains the following information:

- A trajectory that describes the geometric shape of the cluster [11]. This is obtained directly from the detector element and is thus corrected for any misalignments

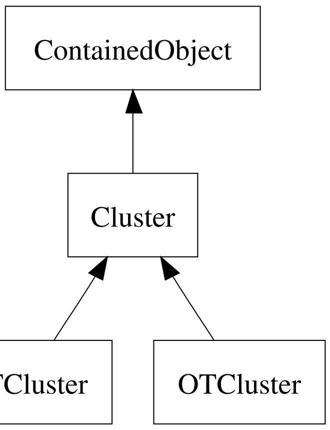

Figure 1: Class diagram for the cluster classes.

- The uncertainty on the cluster measurement.
- A flag indicating whether the cluster is in a hot region.
- Flags indicating the reconstruction status of the cluster. For example, whether it has been used on a track.

In the case of the **Tsa::STCluster** the derived class provides access to the **LHCb::STLiteCluster** from which the cluster was created. The **Tsa::OTCluster** is slightly more complicated. In this case the derived class provides access to the **LHCb::OTTime** from which the cluster was created, a first calculation of the the drift-radius and a pointer to the top-level detector element **DeOTDetector**. To calculate the drift-radius a correction must be made for the propagation time of the signal along the wire. If no 3-D estimate of the cluster position is available the drift radius is estimated using the average distance of the hit along the wire. The pointer to the detector element provides access to the drift-time parameters allowing a more accurate estimation of the drift-radius once y is known.

The following code fragments show some examples of how to create and use an **Tsa::OTCluster**:

```
#include ''TsaKernel/OTCluster.h''
#include ''DeOTDetector/DeOTModule.h''
#include ''Event/OTTime.h''

//...

// create an OTCluster
DeOTModule* module; DeOTDetector* det;
```

```
double error;
LHCb::OTTime* time;
// The average distance along the wire is 77.3 cm
// Need this to calculate the drift radius
double distAlongWire = 773.*Gaudi::Units::mm;
bool isHot;

// make a cluster
Tsa::OTCluster* aCluster = new Tsa::OTCluster(module, error,
                                               det, time,
                                               distAlongWire, isHot);

// do some things

if (aCluster->isHot() == false){
  double driftRadius = aCluster->driftRadius();
}

// test if a y value is covered by the trajectory
double yTest;
bool isInside = aCluster->isInside(yTest, 10.0);

// get the midPoint of the wire
Gaudi::XYZPoint point = aCluster->midPoint();

const double z = aCluster->zMid();

// you have to delete it !
delete aCluster
```

Data classes are also provided that represent a **Spacepoint** and a **Track**. These are currently not used and should be looked upon as legacy code from the implementation described in [3].

2.3 Data services

A key feature of the Tsa framework from its inception [3] is that of data services. Typically, during the pattern recognition phase fast access is needed to the data from part of a detector — for example a layer or module. The role of the data service is to facilitate this task and to shield the user from the details of how the data access is actually done. In the design it is also foreseen that user may want to access the data at more than one level. This is very similar to the view pattern discussed in [12]. To minimize copying and to allow for multiple views internally the data service caches iterators. The LoKi package [13] provides a templated wrapper class ("**LoKi::Range_ <TYPE>**") which converts a pair of iterators to lightweight container. This allows the data service to return an object with the look and feel of a standard STL container.

Three tools are provided, one for the Trigger Tracker (**ITTDataSvc**), one for the Inner Tracker (**IITDataSvc**) and one for the Outer Tracker (**IOTDataSvc**). The example below shows how to get and use the **IITDataSvc** class:

```
#include ``TsaKernel/IITDataSvc.h''

// ...

// get the tool
IITDataSvc* svc = tool<IITDataSvc>(``ITDataSvc'');

// initialize it once each event
StatusCode sc = svc->initializeEvent();

// get data of station 1, layer 2, box 2 (=C side)
Tsa::STRange aRange = svc->partition(1,2,2);
Tsa::STRange::iterator iterC = aRange.begin();
for (; iterC != aRange.end(); ++iterC){
  // just like an STL vector !
  Tsa::STCluster* clus = *iterC;
}

// get the data of the entire layer
Tsa::STRange aRange2 = svc->layer(1,2);
Tsa::STRange::iterator iterC2 = aRange2.begin();
for (; iterC2 != aRange2.end(); ++iterC2){
  // just like an STL vector !
  Tsa::STCluster* clus2 = *iterC2;
}
```

The usage of the other data services is similar. In the future by using the Gaudi incident service it should be possible to remove the need for the user to initialize the tool once per event. The possibility of replacing the integers in the interface with channelIDs will also be investigated.

For the user the above discussion is sufficient. The more interested reader may ask how internally the partitioning of the data is performed. As discussed in [3] this is done by using the fact that the data is sorted by channel and using the STL lower_bound algorithm [7].

2.4 Utility classes

The framework also provides classes that represent lines and parabolas in 2-D. This is done for several reasons. First, this means algorithmic code is not duplicated. Second, for the parabolic case by encapsulating the concept of a parabola within a class it can be ensured that Horner's form [14], which is computationally optimal, is used for evaluation. Finally, it allows to provide constructors and factories for conversion between these and other objects in the framework.

2.4.1 Line class

This class represents a line in 2-D. The class provides the three possible constructors that can be envisaged:

```
#include ''TsaKernel/Line.h''

//...

// constructor from slope and constant
double m, c;
Tsa::Line aLine = Tsa::Line(m,c);

// constructor from slope and point
double x1, z1;
Tsa::Line aLine2 = Tsa::Line(m,x1,z1);

// constructor from two points
double x2, z2;
Tsa::Line aLine2 = Tsa::Line(x1,x2, z1,z2);
```

To find a point on the line is then trivial:

```
Tsa:Line aLine;
double z;
const double x = aLine.value(z);
```

Finally, a free function is provided to convert a pair of 2-D lines to the standard 3-D line class provided in LHCbMath package:

```
Tsa::Line xLine, yLine;
// Tsa::Line3D is a typedef to the standard line class
Tsa::Line3D line = Tsa::createLine3D(xLine,yLine);
```

2.4.2 Parabola class

The parabola class provides one constructor:

```
#include ''TsaKernel/Parabola.h''

//...

// x = az^2 + bz + c
double a, b ,c ;
Tsa::Parabola parabola = Tsa::Parabola(a,b,c);
```

Constructors from other objects are provided via free functions in the header files “ParabolaFactory.h”. The interface of the class is similar to that of the Line class. The examples below illustrate its use:

```
#include ''TsaKernel/Parabola.h''
// x = az^2 + bz + c
double a, b ,c ;
Tsa::Parabola parabola;
```

```

// value at z
double z;
const double x = parabola.value(z);

// derivative at z
const double tx = parabola.derivative(z);

// get the line that is tangential at a given z
Tsa::Line aLine = parabola.tangent(z);
```

2.4.3 Glue functions

Many pattern recognition tasks can be seen as an interaction between clusters, the geometric classes just described and the geometry functions located in LHCbMath. To facilitate this some wrapper functions are provided that glue the various components together and hide technical details from the user. For example:

```
#include ``TsaKernel/ClusFun.h''
#include ``TsaKernel/Cluster.h''

//...

// get two x clusters
Tsa::Cluster clus1, clus2;

// these clusters define a plane
Gaudi::Plane3D aPlane =  Tsa::ClusFun::plane(clus1,clus2);

// get the intersection of a stereo cluster with the plane
Tsa::Cluster clus3;
Gaudi::XYZPoint iPoint;
bool isOK = Tsa::ClusFunr::intersection(clus3,aPlane, iPoint);


// we just built a spacepoint, iPoint !
```

2.5 Expectation tools

A powerful variable to discriminate between ghost and real tracks is the number of hits expected on a track compared to the number actually found. The framework provides two tools, one for the Inner Tracker (**IITExpectedHits**) and one for the Outer Tracker (**IOTExpectedHits**), that allow to test whether a hit is expected in a given layer. This information can then be used either to simply count the number of faults or to build a discriminating variable such as a likelihood. The use of the tool for the Outer Tracker case is illustrated by the following code fragments:

```
#include ``TsaKernel/IOTExpectedHits.h''

```

```
// ...

// get the tool
IOTExpectedHits* hits = tool<IOTExpectedHits>(``IOTExpectedHits'');

// variables needed
typedef std::vector<IOTExpectedHits::OTPair> OTPairs;
OTPairs output; // output container

int sect; // sectoring
LHCb::OTChannelID chan = OTChannelID(1,1,0,0,0); // station 1, layer 1
Tsa::Line aLine; // track in y
Tsa::Parabola aParab; // track in z

hits->collect(aParab,aLine,testChan,output, sect);

}
```

The following comments should be made on the arguments passed to the tool:

- The line and parabola parameterize the track in the region of interest.
- The test channel identifies which detector element is to be searched. For the Outer Tracker at least the station and layer should be given. If more information is given it will be used to reduce the number of modules that have to be searched.
- The output is a list of hit channels and drift distances.
- The sectoring gives some additional hint of where to look. Sector 3 corresponds to the lower half of the Outer Tracker and sector 4 to the upper half [1].

The use of the Inner Tracker tool is very similar:

```
#include ``TsaKernel/IITExpectedHits.h''

// ...

// get the tool
IITExpectedHits* hits = tool<IITDataSvc>(``IITExpectedHits'');

// variables needed
typedef std::vector<IITExpectedHits::ITPair> ITPairs;
ITPairs output; // output container

int sect; // sectoring
LHCb::STChannelID chan = STChannelID(1,1,1,0,0); // station , layer , box
Tsa::Line aLine; // track in y
Tsa::Parabola aParab; // track in z

m_expectedHits->collect(aParab,aLine,testChan,output, sect);

}
```

In this case:

- The test channel identifies which detector element is to be searched. For the Inner Tracker this is the station layer and box.
- The output is a list of all possible hit strips. The cluster to which each strip belongs is flagged by the second argument.
- The sectoring gives some hint of where to look. Sector 0 corresponds to the left/right boxes in the Inner Tracker, sector 1 to the bottom box and sector 2 to the top box [1].

Clearly, these tools have a large potential to be used outside the Tsa framework. To encourage this, the code should be refactored so that the internal sectoring convention used in the seeding is removed from the interface [1].

3 Seed Extension

The Tsa framework was originally developed to build spacepoints in the Inner Tracker [3]. For DC' 06 it was used to port the FORTRAN implementation of the seeding described in [15] to C++. During this process several classes were developed in collaboration with Roger Forty which will described here. Subsequently, this code was adopted as the baseline tracking for the L0 confirmation strategy in the T-stations that is being developed for the HLT. This has led to the re-structuring of the **TsaSeed** algorithm as a 'tool-box' with the consequence that these classes are more visible to the user. In the following sections the data classes, algorithms and tools used in the seeding are described.

4 Algorithms

A full description of the seeding algorithm can be found in [1]. The algorithm flow is shown in Fig. 3. The steps are:

Data Preparation Clusters are made for the Outer and Inner Tracker.

[1]In fact since the test channel also gives some hint of which detector element to search the sector number duplicates information that can easily be obtained from elsewhere.

Initialization The data services are initialized.

Main Algorithm Seed tracks are created.

Output The seed tracks are translated to objects of **LHCb::Track** type.

The main algorithm consists of a projection based search in both the Inner and Outer Tracker. After this step any unused hits in the Inner Tracker are used to build spacepoints (refered to as stubs). These are then linked to make track candidates. Finally, any remaining unused spacepoints are extended into the Outer Tracker.

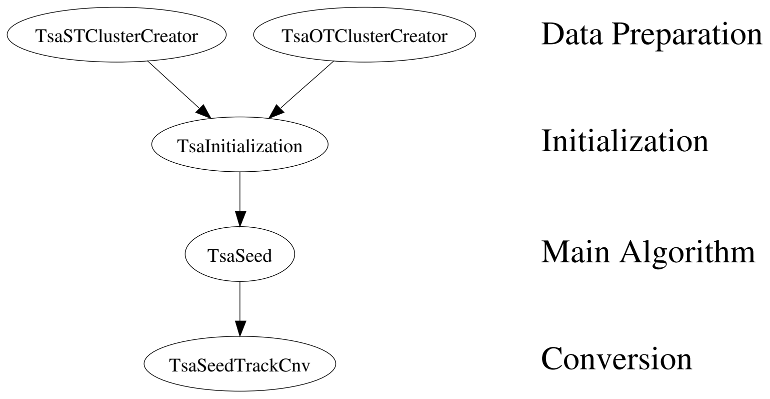

Figure 2: Algorithms Sequence for the Tsa seeding.

4.1 Data Classes

In an early state of the algorithm the input **Tsa::Clusters** are converted to objects of type **SeedHit**. These objects to a large extent mimic the use and the functionality of the **Tsa::Clusters**. Their existence is due to convenience. In FORTRAN based frameworks it is normal to copy input information from banks or common blocks into local working objects. It seems feasible, with some refactoring of the code, to merge these classes into one.

The **SeedHits** are used to build **SeedTracks**. These consist of a list **SeedPnts**, track parameters and other information such as the likelihood. A **SeedPnt** is a **SeedHit** with some additional information due to the fact it is on a track. For example due to the tilt the z coordinate of the measurement in the global coordinate system is only known when it is added to a track. Additionally, for the Outer Tracker it holds the result of the resolution of the drift-time ambiguity.

Finally, a **SeedStub** class represents a stub or spacepoint. This is a list of the four hits that make a stub, plus the determined parameters (x,y,tx,ty) of the stub.

All the classes have a common look and feel. All accept the **SeedPnt** class derive from KeyObject and use Boost pool allocators for memory allocation. Functors are provided for sorting and also for searching using the STL lower_bound algorithm. In addition, some wrapper functions are provided that hide the functors from the user. The example below shows how to collect the **SeedHit**s from a stereo-layer that are consistant with a given track found in the x search:

```
#include ``TsaKernel/SeedTrack.h''
#include ``TsaKernel/SeedHit.h''
#include ``TsaKernel/SeedFun.h''

 //
 SeedTrack* seed;  std::vector<SeedHit*> hits;

 // make a plane tangent to the track --> this should be a function !
 const double zFirst;
 const double slope = seed->xSlope(zFirst,TsaConstants::z0);
 Gaudi::XYZVector vec(1., TsaConstants::tilt*slope, -slope);
 Gaudi::XYZPoint point(seed->x(zFirst,TsaConstants::z0),
                       hits.front()->yMid(), zFirst );
 Gaudi::Plane3D plane = Gaudi::Plane3D(vec,point);

 // point and some tolerance
 Gaudi::XYZPoint iPoint;
 const double tol =  0.1*hits.front()->clus()->length();

 // find the first stereo hit that is compatible
 // ie has some some overlap in x
 std::vector<SeedHit*>::iterator it = SeedFun::startStereo(hits,point.x(),tol);
 std::vector<SeedHit*>::iterator stop = SeedFun::endStereo(hits,point.x(),tol);

 for (; it != stop; ++it){
   SeedHit* hit = *it;
   if ( !Tsa::ClusFun::intersection(hit->clus(),plane,iPoint) ) continue;

   // check for some overlap
   if (hit->clus()->isInside(iPoint.y(),20.0)){
     // do something !
   }

 } //it
```

All these classes and functions are located in the TsaKernel package.

4.2 Tool Interfaces

The projection-based track search proceeds by first looking for hits in the x-measuring layers, confirming them in the stereo-layers, calculating a likelihood and then making a final selection of tracks (Fig. 3).

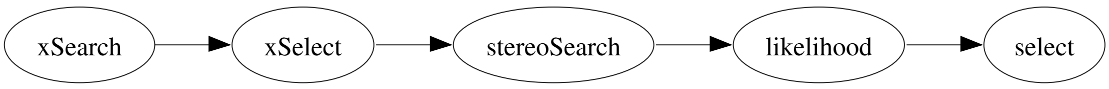

Figure 3: Internal flow of the projection based search in the **TsaSeed** algorithm.

All of these steps either produce or update a set of tracks given a set of hits and (optionally) a track state that gives some hint of what tracks are being searched for. This suggests that the seeding can be written as set of tools that are executed in sequence. The majority of the tools can naturally inherit from a common interface (**ITsaSeedStep**). The interface for this tool is given below:

```
class ITsaSeedStep: virtual public IAlgTool {
public:

  /// Retrieve interface ID
  static const InterfaceID& interfaceID() { return IID_ITsaSeedStep ; }

  virtual StatusCode execute(std::vector<SeedTrack*>& seeds,
                             std::vector<SeedHit*>
                             hits[6] = 0 ) = 0;

  virtual StatusCode execute(LHCb::State& hint,
                             std::vector<SeedTrack*>& seeds,
                             std::vector<SeedHit*> hits[6] = 0 ) = 0;

};
```

Fig. 4 shows the class diagram for all the tools that inherit from this class. All of the steps mentioned above translate to a to an instance of a tool. In addition, base classes are used to express common functionality between the Inner and Outer Tracker. It should be noted that the presence of the vector of **SeedHit** in the interface is not ideal. By further refactoring the seeding code it should be possible to remove this from the interface.

The exceptions to this rule are the tools related to the stub finding. In this case it was not possible to design a common interface. Therefore, separate interfaces exists for the steps involved: stub finding (**ITsaStubFind**), linking (**ITsaStubLinker**) and extending (**ITsaStubExtender**). By further

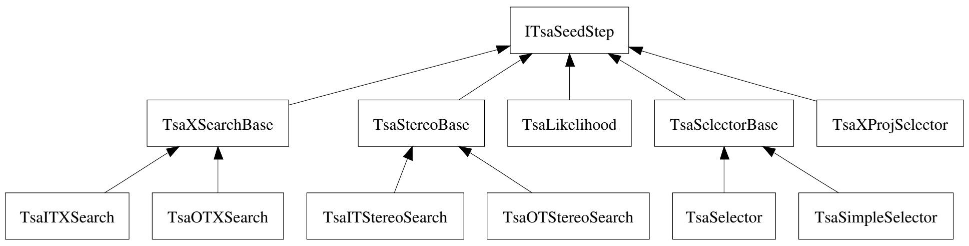

Figure 4: Class diagram for the tools inheriting from **ITsaSeedStep**.

refactoring the code it should be possible to converge on a common interface for all the tools. The interfaces for all the tools discussed can be found in TsaKernel whilst the implementations are located in TsaAlgorithms.

5 Metric Testing

Metric testing of the code has been done using the cccc package [16]. The results are summarized in Table 1. From these numbers it can be seen that:

Package	LOC	LOC/COM	MCC/module
TsaKernel	2482	3.8	3.8
TsaAlgorithms	3575	4.8	11.4

Table 1: Summary of metric tests. The abbreviations used are: LOC (Lines Of Code), COM (Lines of Comment) and MCC (McCabe Cyclomatic Complexity).

- The framework including the Seed extension and legacy code amounts to 6137 lines of code.
- The code is well commented.
- The McCabe Cyclomatic Complexity, which is a measure of the decision complexity of the functions making up the class or module, is also reasonable [2].

[2]Ideally this number should be between three and seven.

The numbers in the Table clearly give the 'average' picture. Looking in more detail it can be seen that a few classes have significantly less comments than the average. For example, the **Parabola** and **Line** classes have a comment for every 50 lines of code. This information can be fed back and used to improve the documentation of the code.

Examination of the MCC per module also reveals some interesting information. The most complicated modules are the x searches in the Outer and Inner Tracker which have MCCs of 62 and 48 respectively. For comparison the likelihood calculation has an MCC of 35.

References

[1] M. Needham and R. Forty. Standalone Track Reconstruction in the T-Stations. LHCb-note 2007-022.

[2] R. Forty and M. Needham. Updated Performance of the T seeding. LHCb-note 2007-023.

[3] M. Needham. Tsa: Fast and Efficient reconstruction for the Inner Tracker. LHCb-Note 2004-075.

[4] S. Meyers. *More Effective C++*. Addision-Wesley, 1996.

[5] S. Meyers. *Efficient STL*. Addision-Wesley, 2001.

[6] D. Vandevoorde and N. Josuttis. *C++ Templates*. Addision-Wesley, 2003.

[7] N. Josuttis. *The C++ Standard Library*. Addision-Wesley, 1999.

[8] http://lhcb-release-area.web.cern.ch/LHCb-release-area/DOC/brunel.

[9] M. Cattaneo *et al.* GAUDI LHCb Data Processing Applications Framework User Guide. LHCb-Note, 1999.

[10] http://www.boost.org.

[11] E. Bos *et al.* The Trajectory Model for Track Fitting and Alignment. LHCb-note 2007-008.

[12] D. Abrahams and A. Gurtovsky. *C++ Template Metaprogramming*. Addision-Wesley, 2004.

[13] I. Belyaev. LoKi: Smart and Friendly C++ Physics Analysis Toolkit. LHCb-Note 2004-023.

[14] W. Horner. A new method of solving numerical equations of all orders by continous approximation. *Philisophical Transactions of the Royal Society of London*, pages 308–335, 1819.

[15] R. Forty. Track Seeding. LHCb-Note 2001-109.

[16] http://cccc.sourceforge.net/.