LHCb Note 2007-008
TRACKING

The Trajectory Model for Track Fitting and Alignment

E. Bos[1], M. Merk[1], G. Raven[1], E. Rodrigues[1], J. van Tilburg[2]

[1] *NIKHEF, Amsterdam, The Netherlands,*

[2] *University of Zürich, Switzerland*

Abstract

This note introduces the concept of Trajectories. The LHCb trajectory model and the implementation in the track fitting and tracking sub-detector code as in Brunel v31r2 are described. The possible use of trajectories for alignment is outlined.

Contents

1 Introduction **3**

2 Fitting with Trajectories **3**

2.1 Decoupling Geometry from the Track Fit 3

2.2 Trajectories in the Track Fit 4

3 The LHCb Trajectory Model **5**

3.1 Trajectory class 5

3.2 LineTraj class 6

3.3 CircleTraj class 7

3.4 ParabolaTraj class 8

3.5 StateTraj class 8

3.6 DifTraj class 8

3.7 TrajPoca tool 9

4 Use in Tracking Sub-detectors **9**

4.1 ST trajectories 10

4.2 VELO trajectories 10

4.3 OT trajectories 11

4.4 Trajectories ownership and std::auto_ptr 11

5 Trajectories for Alignment **11**

6 Conclusion **12**

1 Introduction

In the past two years the whole of the tracking model and software have been reviewed. Additional functionality has been included to accommodate the need for a realistic detector geometry description. The LHCb tracking event model is described in detail elsewhere [1]. Here the focus lies on the track fitting and alignment part of the model, which has been developed to deal naturally with realistic and non-ideal detector geometries.

2 Fitting with Trajectories

2.1 Decoupling Geometry from the Track Fit

Track fitting is connected to the detector geometry through the use of sub-detector hits as input information. In LHCb, the tracking detectors are the Vertex Locator (VELO) [2], the Trigger Tracker (TT) [3], and the inner (IT) [3] and outer (OT) [4] trackers.

The LHCb detector comprises a large number of components whose locations and sizes are stored in a geometry XML database. For the tracking sub-detectors, the components are by design flat-surfaced boxes. In light of the effort to move the (old) ideal detector description to a both more realistic and align-able one, a number of changes and additions have been made to the software. The information stored in the XML database can now be updated such that the changes are automatically propagated to all affected data. Such an update affects the track fit, since that needs to know the geometry of the tracking sub-detector elements which recorded hits in order to determine the most accurate estimates of the track parameters. It is desirable for the track fitting code to be robust against changes in the XML database. Having to rewrite the fitting code whenever a new shape is implemented would be impractical. This "robustness goal" has been achieved by adopting and implementing the concept of trajectories from the BaBar experiment tracking software, of which a general account can be found in [5].

Trajectories are objects which serve to decouple the track reconstruction code from the specific implementation of the detector element's geometry description. To this end, a trajectory contains the geometrical information of the detector element, *i.e.* its location, size and shape, and can be thought of as a curve in global coordinate space. From this information the trajectory can deduce a parabolic expansion at any point along its length. Since the format of this expansion is independent of the curve considered, it is used as the input of geometry information to the track fit, thereby making the fit robust against changes in the shape of detector elements.

2.2 Trajectories in the Track Fit

The LHCb track fit (described in [6]) determines the best track parameters from a set of measurements. It extrapolates the track from the state at the previous measurement to the next measurement, which it needs to approach as closely as possible before projecting the track state onto the measurement space and subsequently determining the residual. Previously (see [7]), the track would be extrapolated until it reaches the z-position of the ideal detector plane which recorded the hit. The coordinate along the length of the strip or wire would be chosen such as to minimise the distance between the measurement and the track state. Since this approach depends upon the detector plane to be flat and vertical, this can not be used in a general and realistic scenario.

A measurement contains the precision coordinate value obtained from a hit, for instance the R-coordinate in case of a VELO R sensor hit. It does not know accurately where along the other two coordinates the particle traversed the detector element. Staying with the VELO R-hit example, the track fit will have to determine where the track comes closest to a curve shaped like a R strip, going through the centre of the charge deposition and being in the plane of the R sensor. This is where the concept of trajectories comes into play: the measurement contains a trajectory, obtained from the sub-detector code, corresponding to the hit type, which contains the curve just described. Inside, the trajectory knows the shape of this curve in global coordinates and it can provide a parabolic expansion to that curve at any requested point along it.

As a first step, the present track fit extrapolates the track to the z-position associated to the centre of the detector element. Given the installation precision of the tracking sub-detectors, the state created at this z-position is arguably close to the actual hit. From the track state and magnetic field vector $\vec{\mathbf{B}}$ at this (conventional) z-position, a "state trajectory" is made that parameterises the local shape of the track (see subsection 3.5). The problem of determining the point where the track is closest to the measurement is now reduced to calculating the closest approach between the curves contained within the trajectory of the measurement and the trajectory of the track state. Since the trajectories can provide parabolic expansions of these curves, a dedicated tool (see subsection 3.7) requires but one method in order to determine both the two points on the curves where they are closest to each other and the distance (vector) between those points.

The minimisation tool also knows how the distance vector changes as a function of the state trajectory's closest point position and the state trajectory knows how the closest point moves along the curve as a function of the state parameters. The product of these derivatives is the so-called projection matrix, which projects a state vector onto the measurement space. The residual between this projection and the measurement value is then determined.

3 The LHCb Trajectory Model

The trajectory concept is translated into a C++ base class called `Trajectory`. It declares the basic functionality as required to perform its designed task: a trajectory parameterises a curve in terms of an "arc length" along itself. The shape and position of an object is used as input when constructing a trajectory, which in practice is always a curve in space.

A number of trajectory classes derived from the `Trajectory.h` base class are available and cover the present needs (see figure 1). The trajectory for a track state, `StateTraj`, resides in the `Tr/TrackFitEvent` package [8] whereas the detector-related trajectory classes `LineTraj`, `CircleTraj` and `ParabolaTraj` can be found in the `Kernel/LHCbKernel` package [9]. All these concrete classes derive from `Trajectory`; except for `StateTraj` which in addition derives a derivatives method from the `DifTraj` class 3.6, included in `Kernel/LHCbKernel`.

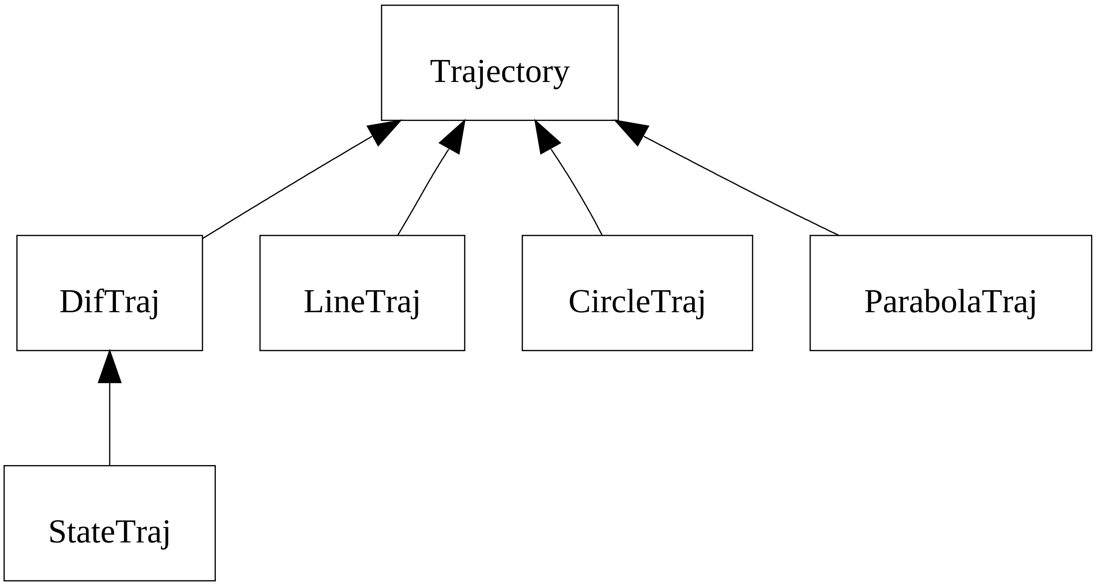

Figure 1: Trajectories inheritance diagram.

The next sub-sections are dedicated to the description of the trajectory model classes. The `TrajPoca` minimisation tool, located in `Kernel/LHCbAlgs` [10], is also described below. The corresponding tool interface, `ITrajPoca`, is included in `Kernel/LHCbKernel`.

3.1 Trajectory class

Trajectories are "associated" to objects – a track state or a sub-detector hit – and constructed from the object's shape and position.

Any location along the trajectory's curve can be referred to using the `arclength` variable. The `arclength` is zero at a pre-defined point on the curve and can run in both positive and negative directions. The range of valid `arclength` values is accessible through `std :: pair < double, double > range()`. One can also obtain the minimum and maximum validity values through the `double beginRange()` and `double endRange()` methods. Any `arclength` value can be limited to this validity range using the `double restrictToRange(arclength)` method. From the range the length of the curve can be determined, which is available via the `double length()` method.

At any value of `arclength` the position, direction, and curvature vectors – `XYZPoint position(arclength)`, `XYZVector direction(arclength)` and `XYZVector curvature (arclength)` – can be obtained, being them the zeroth-, first- and second-order derivatives of the curve at the specified `arclength` value. This curve parameterisation reflects the fact that all trajectories are locally described as parabolas. These local second-order expansion parameters at a specific `arclength` value can be obtained through the method `void expansion(arclength, XYZPoint& position, XYZVector& direction, XYZVector& curvature)`, where one provides references to the expansion parameters which will be assigned to.

In order to quantify the distance in arc length for which an expansion describes the curve to within a user-defined tolerance, the `double distTo1stError(double arclength, double tolerance, int pathDirection)` method is available. It returns the maximal arc length value in the specified direction (+1 or -1) for which the second-order term of the expansion is smaller than the tolerance. The `double distTo2ndError(double arclength, double tolerance, int pathDirection)` method returns the maximal value for which the third-order term is smaller than the tolerance.

3.2 LineTraj class

A linear trajectory is created in case the curve we are concerned with is a straight line. The `LineTraj` class derives from the `Trajectory` base class and offers four constructors as shown in table 1. The first one takes the `XYZPoint` at the middle of the line, a `XYZVector` pointing along either direction of the line and the arc length `range` as arguments. When the direction vector is normalised, one can use the somewhat faster second constructor by additionally supplying a boolean whose value is not used. The third constructor allows one to explicitly specify whether the direction vector offered is normalised or not. Note that this is done through the `isNormalised` enum, which contains the values `yes` and `no`. Finally, one can construct a `LineTraj` from the positions of the two end-points of the line.

LineTraj(	const XYZPoint& middle, const XYZVector& dir, const std::pair<double,double>& range);
LineTraj(	const XYZPoint& middle, const XYZVector& dir, const std::pair<double,double>& range, bool normalized);
enum isNormalized {yes,no} LineTraj(	 const XYZPoint& middle, const XYZVector& dir, const std::pair<double,double>& range, isNormalized boolvalue);
LineTraj(	const XYZPoint& begPoint, const XYZPoint& endPoint);

Table 1: Constructors for the `LineTraj` class.

3.3 CircleTraj class

The `CircleTraj` class is designed to be used when dealing with a curve which is a part of a circle. It offers two constructors as shown in table 2. The first one expects references to the centre point of the circle section, the vectors from there pointing towards the beginning and end of the circle section and the radius. It will consider the circle section to be the shorter arc between the end-points derived from the input description. The second constructor takes references to the centre point of the circle section, the normal vector to the plane spanned by the circle section, a vector pointing from the centre towards the starting point of the circle section and the range of the product of the radius and the opening angle corresponding to the circle section as input.

CircleTraj(	const XYZPoint& origin, const XYZVector& dir1, const XYZVector& dir2, double radius);
CircleTraj(	const XYZPoint& origin, const XYZVector& normal, const XYZVector& origin2point, const std::pair<double,double>& range);

Table 2: Constructors for the `CircleTraj` class.

3.4 ParabolaTraj class

When dealing with a parabola, one can create a parabolic trajectory. The `ParabolaTraj` class offers a single constructor, shown in table 3. It takes references to the centre point, the first and second derivatives at the centre point and a set of begin- and end-points of the parabola as arguments. At present this trajectory type is not used in the tracking code.

ParabolaTraj(	const XYZPoint& middle, const XYZVector& dir, const XYZVector& curve, const std::pair<double,double>& range);

Table 3: Constructor for the `ParabolaTraj` class.

3.5 StateTraj class

The trajectory class designed to parameterise the shape of a track at a state's position is called `StateTraj`. A `StateTraj` trajectory can be constructed from a track `State` and the local magnetic field vector, see table 4. The second constructor requires a state vector, the z-coordinate and the magnetic field vector at the position of the state. It uses the magnetic field vector and the charge from the state vector to determine the derivative and curvature for the trajectory expansion.

StateTraj(	const State& state, const XYZVector& bField);
StateTraj(	const Gaudi::Vector5D& stateVector, double z, const XYZVector& bField);

Table 4: Constructors for the `StateTraj` class.

3.6 DifTraj class

As stated in section 2.2, a state trajectory can calculate the change in location of its closest point as a function of the state parameters. In order to offer the possibility for each trajectory type to calculate the change in location of their closest point as a function of their parameterisation, they can inherit from a class derived from `Trajectory`, called `DifTraj`. It is templated and takes the number of parameters defining the inheriting trajectory type as a construction argument. The derivatives matrix is a `ROOT::Math::SMatrix<double,3,N>` returned by the `derivative(arclength)` method. Through the `ROOT::Math::SVector<double,N> parameters()` method, the values of the parameters to which the derivatives are taken can be obtained. In case of a state trajectory these would be the state parameters.

3.7 TrajPoca tool

The "Points Of Closest Approach" – `TrajPoca` – tool determines where two trajectories come closest to each other. It performs a distance minimisation in three dimensions; the `minimize` member function, see table 5, expects references to the two trajectories, corresponding arc length variables, to the distance vector to be determined, the user desired precision and whether or not the arc length validity range should be respected.

The minimisation is an iterative process, beginning by calling expansions at the starting values of the arc lengths `arclength1` and `arclength2`. It will determine which points on these expansions are closest to each other as well as the corresponding arc length values and distance vector. In over 99% of the cases the minimum is located during the first step of the iteration. However, if the two expansions keep approaching each other as the distance from the expansion points increases and either one passes their `distTo2ndError` value, new expansions will be created at those arc length values. This process continues until the minimum is reached within the desired accuracy, which is 10 μm for the OT and 2 μm for the other measurements. It is also possible to minimise the distance between a point and a trajectory with the TrajPoca tool.

During the minimisation the code performs self-monitoring in order to intercept attempts for trajectories which are (nearly) parallel and diverging minimisations and stops in those cases. Also oscillations around a minimum are caught, passing on as the result the better of the two arc length values. The code can also deal with trajectories containing a description of a curve composed of several pieces, e.g. a set of straight line segments at angles to each other, by pushing the iteration step over the border between the pieces, preventing the minimisation to oscillate at such a point.

The `TrajPoca` tool is located in `Kernel/LHCbAlgs` in order to be a generally available tool; *i.e.* non-tracking code using it will not need to link to a tracking tools package.

4 Use in Tracking Sub-detectors

As discussed above, the trajectory model uses trajectory objects – returned by the relevant tracking sub-detector's code – to describe the curves in global coordinate space corresponding to the recorded hits. The strips (for the VELO, TT and IT) and wires (for

StatusCode minimize(	const Trajectory& traj1, double& arclength1, bool restrictRange1, const Trajectory& traj2, double& arclength2, bool restrictRange2, XYZVector& distance, double precision);

Table 5: Minimisation method of the `TrajPoca` tool.

the OT) are used as reference information and supply the shape to the trajectory. In case of a silicon strip, the curve will be located at the centre of the charge deposition; for an OT hit the curve is located on the hit wire. In the present model linear trajectories are created for the VELO ϕ, TT, IT and the OT measurements; circular trajectories are used for the VELO R measurements.

4.1 ST trajectories

The TT and IT sub-detector code provides `std::auto_ptr` pointers (see section 4.4 for a technical reminder) to trajectories through a `trajectory` method defined in the `DeSTSector` class (`STDet` package [11]). The method takes the `STChannelID` of the closest strip to the centre of charge deposition and the offset in units of pitch as arguments. It internally uses a single `LineTraj` running orthogonal to the strips in the x-direction through the middle of the sensor as a reference when constructing the `LineTraj` which is to be returned. The middle point of that `LineTraj` is determined from the offset, the strip number as in the `STChannelID` and the position of the first strip along the reference `LineTraj`. The code knows the direction and length of a strip, which combined with the just determined point allows to create a `LineTraj` shaped like a strip, but at the location of the centre of the charge deposition as specified by the offset.

4.2 VELO trajectories

The VELO sub-detector code in the `VeloDet` package [12] offers a similar `trajectory` method to the ST, taking a `LHCbID` and the offset as arguments. It delegates to `DeVeloSensor`, passing a `VeloChannelID` and the offset. For ϕ strips the `DeVeloPhiType` class creates `LineTraj` trajectories. It uses the end-points of the strip corresponding to the `VeloChannelID` and depending on the sign of the offset it also gets the end-points of the next or previous strip. The offset times the difference between starting points of the strip and the called adjacent strip plus the start point of the called strip gives the start point of the desired `LineTraj`. The end-point is determined analogously. In case of a VELO R hit, the `DeVeloRType` class creates a `CircleTraj`. Using the `VeloChannelID` to identify the number of the nearest strip, both the radius and the minimum and maximum ϕ angles of that strip are obtained. The offset times the strip pitch is added to the radius and together with the ϕ range is used to make a `CircleTraj` at the centre of charge deposition. All trajectories are made such as to have increasing arc length in the counter-clockwise direction when looking in the direction of increasing z.

4.3 OT trajectories

The Outer Tracker code supplies `LineTraj` trajectories to describe OT measurements (`OTDet` package [13]). Unlike the silicon strip trajectories which represent the centre of the deposited charge, the OT trajectories are located on the wires themselves. It is only later on, during the track fit, that the drift distance in the gas of the straw, equivalent to the centre of charge deposition, is determined. The call for a trajectory `std::auto_ptr` can be made to the `DeOTDetector` class, whose `trajectory` method takes the `LHCbID` of the wire as argument. The call is passed on to the `DeOTModule` class together with the `OTChannelID` derived from the `LHCbID`. `DeOTModule` contains a `LineTraj` in x through the middle of the module similar to the ST case for each of the two monolayers of a module. As it knows the wire position, direction and length, it can make a `LineTraj` for the wire, which it then returns.

4.4 Trajectories ownership and std::auto_ptr

A trajectory object representing a measurement is created using a call to `new` and ownership is passed to the caller via a `std::auto_ptr` [14]. The caller can pass on references to the trajectory to other classes without loosing the ownership of the trajectory. Creating another std::auto_ptr which points to the address contained by the original one results in the original `std::auto_ptr` becoming a `NULL` pointer, *i.e.* only one `std::auto_ptr` can be valid per trajectory. Once the `std::auto_ptr` goes out of scope, the trajectory object is automatically deleted.

5 Trajectories for Alignment

The LHCb trajectory model is presently being expanded to provide a consistent event model usable not only in the Kalman fitting code, but also for the purpose of alignment. A dedicated `AlignTraj` trajectory class can be found in the `LHCbKernel` package. The

AlignTraj inherits relevant functionality from the DifTraj class, see 3.6. It takes a (external) trajectory as input, which it can both rotate and translate. It provides the derivatives with respect to these rotations and translations as a result for the alignment method to analyse.

6 Conclusion

Trajectories are presently used in the pattern recognition, track fitting, sub-detector and alignment code as well as in the Tsa [15] code for L0 confirmation. This note has described the status of the Trajectory Model as implemented in Brunel v31r2.

At present the ideal shapes of the detector elements are used, *i.e.* straight lines and circle sections. Changing to any other detector element shape is, seen from a tracking point of view, a matter of introducing the corresponding trajectories to describe them. The goal of decoupling the details of the geometry from the tracking code has thereby been achieved with the present trajectory model.

References

[1] J. A. Hernando, E. Rodrigues, *Tracking Event Model*, LHCb Tracking Note LHCb 2007-007

[2] P. R. Barbosa Martinho *et al.* (LHCb Collab.), LHCb VELO Technical Design Report, CERN/LHCC 2001-011

[3] P. R .Barbosa Martinho *et al.* (LHCb Collab.), LHCb Inner Tracker Technical Design Report, CERN/LHCC 2002-029

[4] P. R. Barbosa Martinho *et al.* (LHCb Collab.), LHCb Outer Tracker Technical Design Report, CERN/LHCC 2001-024

[5] D. N. Brown, W. A. Charles and D. A. Douglas, *The BaBar Track Fitting Algorithm*, Proceedings of CHEP 2000, Padova, Italy, Feb. 2000

[6] E. Rodrigues, *The LHCb Track Kalman Fit*, LHCb Tracking Note LHCb 2007-014

[7] R. Hierk, M. Merk, M. Needham and R. van der Eijk, *Performance of the LHCb OO track fitting software*, LHCb Tracking Note LHCb 2000-086

[8] *http://isscvs.cern.ch/cgi-bin/cvsweb.cgi/Tr/TrackFitEvent/?cvsroot=lhcb*

[9] *http://isscvs.cern.ch/cgi-bin/cvsweb.cgi/Kernel/LHCbKernel/?cvsroot=lhcb*

[10] *http://isscvs.cern.ch/cgi-bin/cvsweb.cgi/Kernel/LHCbAlgs/?cvsroot=lhcb*

[11] *http://isscvs.cern.ch/cgi-bin/cvsweb.cgi/Det/STDet/?cvsroot=lhcb*

[12] *http://isscvs.cern.ch/cgi-bin/cvsweb.cgi/Det/VeloDet/?cvsroot=lhcb*

[13] *http://isscvs.cern.ch/cgi-bin/cvsweb.cgi/Det/OTDet/?cvsroot=lhcb*

[14] B. Stroustrup, *The C++ Programming Language(3rd edition)* Addison Wesley, 1997, ISBN 0-201-88954-4, subsection 14.4.2

[15] M. Needham, *The Tsa Reconstruction framework,* LHCb Tracking Note LHCb 2007-037