ATHENA Data Classes and Algorithms for the ATLAS FCal Test Beam

E. Inrig

Carleton University, Ottawa, Ontario, Canada

August 30, 2003

Abstract

Descriptions of the ATHENA data classes and algorithms developed for the ATLAS FCal test beam are presented.

1 Track Fitting using the Beam Proportional Chambers

1.1 Experimental Setup

The tracks of particles incident on the calorimeter were reconstructed using data obtained from the Beam Proportional Chambers, or BPCs, which were positioned along the beam line upstream of the calorimeter. Six BPCs were used, with each one determining an x-coordinate and a y-coordinate, each in a different plane. The two chambers nearest the calorimeter were placed on a table, the height of which was adjusted so that the beam was centred in the chambers at each setting of the Bend 9 magnet.

1.2 Track Fitting Using Linear Regression

Because the particles all have non-zero momentum in the z direction, the equation of the line representing the track of a particle can be written in parametric form as

$$x = a1 + b1 \cdot t \qquad y = a2 + b2 \cdot t \qquad z = t$$

Because the x and y positions of the particle are determined at different z positions, linear fits in x and z and in y and z must be performed independently. The line in three dimensions is then given by the intersection of the two planes, parallel to the y and x axes respectively, determined by these linear fits.

In this case, the goal is to find the relationship between two variables, X and Z (or Y and Z) by fitting a straight line to the data. According to the linear regression model,

$$X = a + bZ + e$$

where e is a random variable with mean zero known as the residual. The values of the coefficients a and b can be determined using the condition that the sum of the squares of the residuals must be a minimum.If the equation of the line of best fit is given by $x(z) = a + bz$ and the actual data points by (z_i, x_i), then the function to be minimized is

$$f = \sum_i (x_i - x(z_i))^2$$

Taking $\frac{\partial f}{\partial a} = 0$ and $\frac{\partial f}{\partial b} = 0$ and solving for the coefficients a and b gives

$$a = \frac{S_x S_{zz} - S_z S_{xz}}{N S_{zz} - S_z^2} \qquad b = \frac{N S_{xz} - S_x S_z}{N S_{zz} - S_z^2}$$

where $S_x = \sum_i x_i$, $S_z = \sum_i z_i$, $S_{xz} = \sum_i x_i z_i$, $S_{zz} = \sum_i {z_i}^2$, and N is the number of data points.

1.3 Correcting the Relative Alignment of the BPCs

Each BPC has its own local coordinate system which is only approximately aligned with those of the other chambers. If no correction is made and a straight line is fit to the coordinates of the chambers, the distributions of the residuals have non-zero means, implying that the chambers are not perfectly aligned.

To correct this misalignment it is necessary to add a constant "offset" to the position given by each chamber. An additional complication arises from the fact that the last two chambers are located on the movable table, the height of which varies slightly from run to run.The offsets needed to correct the alignment of the chambers were determined relative to the first chamber, which is given an offset of zero. By varying the values of these offsets over a range of 2mm by steps of 0.5 mm, covering all possible combinations in this range, and finding the combination that minimized the sum of the squares of the residuals for each event, approximate optimal values for these offsets were obtained.

The accuracy of these values was improved by repeating the procedure, varying the offsets over a small range around the approximate values with a smaller step size. Muon runs were used for this analysis because the muons were likely to have cleaner tracks than would electrons or pions. When the offsets determined in this manner are added to all BPC coordinates before performing a straight-line fit, the residuals are normally distributed with mean zero.

Because the offsets are determined relative to the first BPC, there is only one "absolute" point in space. This introduces the possibility that the offsets determined would bring the chambers into alignment with each other, but not necessarily parallel to the beam line. However, the chambers were very nearly aligned to begin with, and the coordinates of BPC 6 were only allowed to vary over a range of a few millimetres, so this effect could only produce an error in the angle of the order of a few microradians.

The alignment of the chambers in y is not quite as good due to complications introduced by the varying table height. In this case, offsets were

first determined for the four stationary chambers only. The fifth and sixth chambers also have a constant difference in alignment, and this was estimated by finding the best offsets for all 6 chambers in y for several runs and taking the average difference between the offsets of chambers 5 and 6. To find the approximate table height for each beam position, the table height was varied and the height giving the smallest sum of squares of the residuals determined. This procedure was repeated over several runs to obtain average table heights at positions 1-3, 4L, and 4H.

1.4 Determining the Correct Table Height

To increase the accuracy of the alignment correction in y, the proper table height must be determined on a run-by-run basis. Unfortunately, this is complicated by the fact that ATHENA processes only one event at a time. The table height must be determined by an algorithm analyzing a large number of events, so it is not possible to apply the results of the algorithm to each event.

This problem can be solved by building a database to allow the alignment of the chambers in y to be corrected for variations in table height for each group of runs. The algorithm that reconstructs the BPC coordinates checks the database to see if a value has been recorded for the table height correction for the current run. If so, it applies this correction to the BPC coordinates; if not, it uses the default (approximate) table height for the run, but executes an algorithm to determine the difference between the actual and approximate table height for the current set of runs and records this in the database. Although this result cannot be applied to the current run, it will be available for future runs. Because a large number of runs are often taken with the same table setting, the algorithm will only be executed when the first of a set of runs is analyzed.

2 Data Classes, Monitoring and Reconstruction Algorithms for Beam Line Detectors

This section provides a reference manual for the data classes for the beam line detectors as well as their associated reconstruction and monitoring algorithms.

2.1 Scintillator Data Classes and Algorithms

In this section, the function of each data class is given along with its public methods. Reconstruction and monitoring algorithms are described, including available job options and histograms produced.

2.1.1 The `LArFCalTBScintillatorBase<T>` class

The `LArFCalTBScintillatorBase<T>` class is a templated base class for raw and reconstructed Scintillator objects. The `LArFCalTBScintillatorRaw` class is derived from `LArFCalTBScintillatorBase<unsigned int>`, while the `LArFCalTBScintillator` class is derived from `LArFCalTBScintillatorBase<double>`.

All methods are provided by the base class; in fact, the only difference between the raw and reconstructed classes is that the raw data class stores an integer ADC count, while the reconstructed class stores an ADC count of type double, to allow for pedestal subtraction. For this reason, only the constructors and methods of the base class are given here; the methods are identical except that `LArFCalTBScintillator` or `LArFCalTBScintillatorRaw` must be substituted for `LArFCalTBScintillatorBase<T>` accordingly in the constructor.

2.1.1.1 `LArFCalTBScintillatorBase<T>` Constructors:

```
LArFCalTBScintillatorBase<T>()
LArFCalTBScintillatorBase<T>(const std::string& ID, T count)
LArFCalTBScintillatorBase<T>(const std::string& ID, T count,
bool overflow)
```

These are templated constructors for a scintillator object. Note that the base class constructor should not be called directly; rather, the constructor for the derived class (either `LArFCalTBScintillator` or `LArFCalTBScintillatorRaw`) should be used.

2.1.1.2 `LArFCalTBScintillatorBase<T>` Access Methods:

```
void setID(const std::string& ID)
void setCount(T count)
void setOverflow(bool overflow)
```

These methods allow the user to modify the private data members of the `LArFCalTBScintillatorBase<T>` class. The argument of the `setCount` method is of type unsigned int for a `LArFCalTBScintillatorRaw` object, double for `LArFCalTBScintillator`.

```
std::string getID()
T getCount()
bool isOverflow()
```

The above methods can be used to retrieve the string ID, ADC count, and overflow status of a scintillator respectively.

2.1.2 The LArFCalTBScintillatorContainer Class

There are two container classes designed to hold all scintillator objects for one event. The container, rather than each individual scintillator object, is recorded in StoreGate.

LArFCalTBScintillatorRawContainer: This container holds objects of type LArFCalTBScintillatorRaw. (Class ID 2766)

LArFCalTBScintillatorContainer: This container holds objects of type LArFCalTBScintillator. (Class ID 27100) Because the constructors and public methods of the two classes are identical, only the LArFCalTBScintillatorContainer class will be described here. The LArFCalTBScintillatorContainer class is derived from both ITBDataObject and DataVector<LArFCalTBScintillator>, so the methods of these classes are also available.

2.1.2.1 LArFCalTBScintillatorContainer Constructors:

```
LArFCalTBScintillatorContainer()
LArFCalTBScintillatorContainer(LArFCalTBScintillator* scintillator)
```

A LArFCalTBScintillatorContainer object can be created as an empty container, or containing one scintillator object.

2.1.2.2 LArFCalTBScintillatorContainer Access Methods:

```
void setMap(LArFCalTBScintillator* scintillator)
LArFCalTBScintillator* getScintillator(const std::string& name)
```

The setMap method is used to insert a scintillator object into the container. Scintillators are retrieved from the container using the getScintillator method, which takes the string ID of the required scintillator as an argument. It is important to use the setMap method to add a scintillator to the container (as opposed to the push_back method from the DataVector class) or it will not be possible to retrieve the scintillator object from the container using the getScintillator method.

```
void setName( const std::string& name)
std::string getName()
```

These two methods are required by the ITBDataObject interface. They are used to set and retrieve the name of the container.

2.1.3 Reconstruction: The `LArFCalTBScintillatorBuilder` Algorithm:

This algorithm retrieves the `LArFCalTBScintillatorRawContainer` for each event from `StoreGate` and applies the pedestal subtraction to the `LarFCalTBScintillatorRaw` objects, creating a new container of `LArFCalTBScintillator` objects and registering the container with `StoreGate`.

2.1.3.1 `LArFCalTBScintillatorBuilder` Job Options: The following job options are available to the user. It is not normally advisable to change the container keys, as described in the first two job options, as ATHENA will be unable to retrieve the `LArFCalTBScintillatorRawContainer` without the correct key, and other algorithms may not run properly without the default `LArFCalTBScintillatorContainer` key.

`InputScintillatorRawContainers`**:** A list of keys (strings) for containers can be provided here by the user.

`ScintillatorContainer`**:** A key (string) for the container for the reconstructed scintillator objects can be provided.

`Scintillators`**:** A list of the IDs (strings) of the scintillators can be given here.

`Pedestals`**:** A list of pedestal values (doubles) to be subtracted from the scintillator raw ADC counts can be provided. These should be given in the order of the above list (`Scintillators`).

2.1.4 Monitoring: The `LArFCalTBRawScintMon` Algorithm:

The `LArFCalTBRawScintMon` algorithm produces histograms of the raw ADC counts for each scintillator.

2.1.4.1 `LArFCalTBRawScintMon` Job Options: The following job options are available to the user. The container key should not normally be changed from the default, nor should the histogram directory.

`RawScintContainer`**:** The key (string) for the `LArFCalTBScintillatorRawContainer`.

`HistogramDirectory`**:** A string giving the directory where the histogram files should be written.

2.1.4.2 `LArFCalTBRawScintMon` Histograms: The following histograms are produced by the `LArFCalTBRawScintMon` algorithm.

Histograms	
100001	Scintillator S1 Raw ADC Counts
100002	Scintillator S2 Raw ADC Counts
100003	Scintillator S3 Raw ADC Counts
100004	Scintillator Veto Raw ADC Counts
100005	Scintillator Muon Raw ADC Counts
100006	Scintillator SFibreAmp Raw ADC Counts

2.2 Tail Catcher Data Classes and Algorithms

The tail catcher consists of a collection of scintillators, so the associated data classes are very similar to those for the other beam line scintillators. One major difference is that the tail catcher is itself a collection of scintillators, so there is no need for a container class.

2.2.1 The `LArFCalTBTailCatcherBase<T>` class

The `LArFCalTBTailCatcherBase<T>` class is a templated base class for raw and reconstructed tail catcher objects. It is derived from the ITBDataObject class. Two classes are derived from the base class:

`LArFCalTBTailCatcherRaw :` A class containing raw data from the tail catcher scintillators, derived from `LArFCalTBTailCatcherBase<unsigned int>`. (Class ID 27101)

`LArFCalTBTailCatcher :` A class containing reconstructed tail catcher data (with pedestal subtraction), derived from `LArFCalTBTailCatcherBase<double>`. (Class ID 27102)

As with the scintillators, all methods are provided by the base class, so only these methods will be described here.

2.2.1.1 `LArFCalTBTailCatcherBase<T>` Constructors:

```
LArFCalTBTailCatcherBase<T>()
LArFCalTBTailCatcherBase<T>(const std::string& name,
const std::vector<T>& signalHigh,
const std::vector<T>& signalLow)
LArFCalTBTailCatcherBase<T>(const std::string& name,
const std::vector<T>& signalHigh,
const std::vector<t>& signalLow,
const std::vector<bool>& overflow)
```

These are templated constructors for a tail catcher object. As with the scintillator base class, these constructors should not be called directly; they are called by the identical constructors of the derived classes above. Note that the template parameter T must be resolved appropriately. Each tail catcher scintillator has both a high-gain and a low-gain signal. The tail catcher also stores a signal that may be a combination of high and low gains; the high gain signal is used except

when the signal is in overflow, in which case the low gain signal is used. The vectors `signalHigh` and `signalLow` are expected to be of equal size.

2.2.1.2 `LArFCalTBTailCatcherBase<T>` Access Methods:

```
void setSignalHigh(const std::vector<T>& signalHigh)
void setSignalLow(const std::vector<T>& signalLow)
void setSignal()
```

These methods can be used to set the high-gain and low-gain signals of the tail catcher scintillators. The `setSignal` method sets a mixed-gain signal, using the high-gain signal by default and the low-gain signal only if the high-gain signal is in overflow.

```
std::vector<T> getSignalHigh()
std::vector<T> getSignalLow()
std::vector<T> getSignal()
```

The above methods return vectors holding the high-gain, low-gain, or mixed-gain signals respectively.

```
bool setOverflow(const std::vector<bool>& overflow)
std::vector<bool> listOverflow()
bool setOverflow(unsigned int i, bool overflow)
bool isOverflow(unsigned int i)
```

These methods are used to set and retrieve the overflow status of the tail catcher scintillators. The first two methods set and retrieve the overflow status of all scintillators using vectors of type `bool`. The last two set and retrieve the overflow status of the scintillator at index `i`.

```
void setName(const std::string& name)
std::string getName()
```

These methods are required by the `ITBDataObject` interface, and are used to set and retrieve the name of the tail catcher object.

2.2.2 Reconstruction: The `LArFCalTBTailCatcherBuilder` Algorithm:

The `LArFCalTBTailCatcherBuilder` algorithm retrieves the `LArFCalTBTailCatcherRaw` for each event from `StoreGate` and applies the pedestal subtraction

to the `LarFCalTBTailCatcherRaw` object, creating a new `LArFCalTBTailCatcher` object and registering it with `StoreGate`.

2.2.2.1 `LArFCalTBTailCatcherBuilder` Job Options: The following job options are available to the user. As with the `LArFCalTBScintillator` reconstruction algorithm, the values of the keys should not normally be modified by the user.

`InputTailCatcher`**:** The key (string) for the `LArFCalTBTailCatcherRaw` object can be provided here by the user.

`TailCatcher`**:** The key for the reconstructed tail catcher object can be provided here.

`HighGainPedestals`**:** The pedestal values (doubles) for the high-gain signal can be given here.

`LowGainPedestals`**:** The pedestal values for the low-gain signal can be given here.

`GainFactor`**:** A scaling factor can be provided by the user here. The low-gain signal is multiplied by the gain factor so that its amplitude is equal to that of the high-gain signal.

2.2.3 Monitoring: The `LArFCalTBRawTCMon` Algorithm:

The `LArFCalTBRawTCMon` algorithm produces histograms of the raw low-gain and high-gain ADC counts for each tail catcher scintillator, as well as the total from all tail catcher scintillators.

2.2.3.1 `LArFCalTBRawTCMon` Job Options: The following job options are available to the user.

`RawTailCatcher`**:** The key (string) for the `LArFCalTBTailCatcherRaw`.

`HistogramDirectory`**:** A string giving the directory where the histogram files should be written.

2.2.3.2 `LArFCalTBRawTCMon` Histograms: The following histograms are produced by the `LArFCalTBRawTCMon` algorithm:

Histograms	
10001	TailCatcher 1 Raw ADC Counts - Low Gain
10002	TailCatcher 2 Raw ADC Counts - Low Gain
10003	TailCatcher 3 Raw ADC Counts - Low Gain
10004	TailCatcher 4 Raw ADC Counts - Low Gain
10005	TailCatcher 5 Raw ADC Counts - Low Gain
10006	TailCatcher 6 Raw ADC Counts - Low Gain
10100	TailCatcher Total Raw ADC Counts - Low Gain
20001	TailCatcher 1 Raw ADC Counts - High Gain
20002	TailCatcher 2 Raw ADC Counts - High Gain
20003	TailCatcher 3 Raw ADC Counts - High Gain
20004	TailCatcher 4 Raw ADC Counts - High Gain
20005	TailCatcher 5 Raw ADC Counts - High Gain
20006	TailCatcher 6 Raw ADC Counts - High Gain
20100	TailCatcher Total Raw ADC Counts - High Gain

2.3 BPC Data Classes and Algorithms

The data classes and algorithms for the beam proportional chambers (BPCs) are outlined in this section. As well as the reconstruction and monitoring algorithms, special algorithms designed to determine the relative alignment of the BPCs are described.

2.3.1 The `LArFCalTBBPCRaw` class

The beam projection chambers produce four TDC signals (left, right, up, down) and two ADC signals (X and Y). The `LArFCalTBBPCRaw` class simply stores these signals as integers, along with a string ID for the detector and the overflow status of each signal.

2.3.1.1 `LArFCalTBBPCRaw` Constructors:

```
LArFCalTBBPCRaw()
LArFCalTBBPCRaw(const std::string& ID,
const std::vector<unsigned short>& ADC,
const std::vector<unsigned short>& TDC)
LArFCalTBBPCRaw(const std::string& ID,
const std::vector<unsigned short>& ADC,
const std::vector<unsigned short>& TDC,
const std::vector<bool>& overflowADC,
const std::vector<bool>& overflowTDC)
```

These are the constructors for a raw BPC object. The ADC signal is expected to be a vector with two elements, (ADC X, ADC Y), while the TDC signal is expected to contain four values, (TDC right, TDC left, TDC up, TDC down). The ADC and TDC overflow vectors must be the same size as the ADC and TDC signal vectors respectively.

2.3.1.2 `LArFCalTBBPCRaw` Access Methods:

```
void setID(const std::string& ID)
std::string getID()
```

These methods are used to set and retrieve the string ID of the BPC.

```
void setADC_x(unsigned short ADC_x)
void setADC_y(unsigned short ADC_y)
unsigned short getADC_x()
unsigned short getADC_y()
```

The above methods allow the user to set and retrieve the values of the ADC signals.

```
void setTDC_right(unsigned short TDC_right)
void setTDC_left(unsigned short TDC_left)
void setTDC_up(unsigned short TDC_up)
void setTDC_down(unsigned short TDC_down)
unsigned short getTDC_left()
unsigned short getTDC_right()
unsigned short getTDC_up()
unsigned short getTDC_down()
```

The above methods allow the user to set and retrieve the values of the TDC signals.

```
void setOverflowADC_x(bool overflowADC_x)
void setOverflowADC_x(bool overflowADC_x)
void setOverflowADC_y(bool overflowADC_y)
void setOverflowTDC_right(bool overflowTDC_right)
...
bool isOverflowADC_x()
bool isOverflowADC_y()
bool isOverflowTDC_right()
...
```

These methods set and retrieve the overflow status of the signals. One example is given for the TDC overflow methods; the other methods follow the same pattern.

2.3.2 The `LArFCalTBBPC` class

Using calibration constants provided, the TDC signals are converted into x and y positions. The `LArFCalTBBPC` class stores the string ID of the BPC, the ADC signal (with pedestals subtracted), the beam position, and the overflow status of the signals.

2.3.2.1 `LArFCalTBBPC` Constructors:

```
LArFCalTBBPC()
LArFCalTBBPC(const std::string& ID,
const std::vector<double>& ADC,
const std::vector<double>& beamPosition)
LArFCalTBBPCRaw(const std::string& ID,
const std::vector<double>& ADC,
const std::vector<double>& beamPosition),
const std::vector<bool>& overflowADC,
const std::vector<bool>& overflowBeamPosition)
```

These are the constructors for a reconstructed BPC object. The ADC signal is again expected to be a vector with two elements, (ADC X, ADC Y). The beam position is also a two-element vector, (x, y). The ADC and beam position overflow vectors must also contain two elements.

2.3.2.2 `LArFCalTBBPC` Access Methods:

```
void setID(const std::string& ID)
std::string getID()
```

These methods are used to set and retrieve the string ID of the BPC.

```
void setADC_x(double ADC_x)
void setADC_y(double ADC_y)
double getADC_x()
double getADC_y()
```

The above methods allow the user to set and retrieve the values of the ADC signals.

```
void setBeamPosition_x(double beamPosition_x)
void setBeamPosition_y(double beamPosition_y)
double getBeamPosition_x()
double getBeamPosition_y()
```

The above methods allow the user to set and retrieve the x and y beam positions.

```
void setOverflowADC_x(bool overflowADC_x)
void setOverflowADC_x(bool overflowADC_x)
void setOverflowADC_y(bool overflowADC_y)
void setOverflowBeamPosition_x(bool overflowBeamPosition_x)
void setOverflowBeamPosition_y(bool overflowBeamPosition_y)
bool isOverflowADC_x()
bool isOverflowADC_y()
bool isOverflowBeamPosition_x()
bool isOverflowBeamPosition_y()
```

These methods set and retrieve the overflow status of the ADC signals and beam positions.

2.3.3 The `LArFCalTBBPCContainer` Class

The `LArFCalTBBPCRawContainer` and `LArFCalTBBPCContainer` classes are virtually identical to the `LArFCalTBScintillatorContainer` class described in section 2.1.2. The only difference is that the `getScintillator` method is replaced by the `getBPCRaw` and `getBPC` methods accordingly.

There two BPC container classes hold all BPC objects for one event. As with the scintillators, the container, rather than each individual BPC object, is recorded in `StoreGate`.

`LArFCalTBBPCRawContainer`: This container holds `LArFCalTBBPCRaw` objects. (Class ID 2764)

`LArFCalTBBPCContainer`: This container holds `LArFCalTBBPC` objects. (Class ID 2765)

2.3.4 The `LArFCalTBLine` class

The `LArFCalTBLine` class stores the parameters for a line reconstructed from the beam positions and positions along the beam line of the BPCs. The beam centre line is taken as the origin for the x-y plane, and z=0 at BPC1X, the chamber furthest upstream from the cryostat.

The parametric form of the equation of the line is described in section 1.2. This can also be written as follows: $(x, y, z) = (x_0, y_0, z_0) + (x_1, y_1, z_1) \cdot t$ The vector (x_0, y_0, z_0) is referred to as the position vector, and (x_1, y_1, z_1) as the direction vector. The line is stored as two vectors, the position vector and the normalized direction vector, called the direction cosines.

2.3.4.1 `LArFCalTBLine` Constructors:

```
LArFCalTBLine()
LArFCalTBLine(const std::string& name,
const std::pair<double, double>& Xparams,
const std::pair<double, double>& Yparams)
LArFCalTBLine(const std::string& name, const
std::vector<double>& posVec,
const std::vector<double>& dirVec)
LArFCalTBLine(const LArFCalTBLine& L)
```

These are the constructors for a `LArFCalTBLine` object. The second constructor takes the line parameters as described in 1.2 as two pairs, $< a1, b1 >$ and $< a2, b2 >$. The third constructor takes the position vector and direction vector of the line, as described above. The last constructor takes a `LArFCalTBLine` object and constructs a copy.

2.3.4.2 `LArFCalTBLine` Access Methods:

```
void setName(const std::string& name)
std::string getName()
void setPositionVector(const std::vector<double>& posVec)
void setDirectionCosines(const std::vector<double>& dirVec)
std::vector<double> getPositionVector()
std::vector<double> getDirectionCosines()
```

The first two methods are used to set and retrieve the string ID of the BPC. The remaining methods allow the user to set and retrieve the position vector and direction cosines of the line. The vectors `posVec` and `dirVec` each must have three elements (x, y, z). The

`setDirectionCosines` method normalizes the vector `dirVec` before storing it. The direction vector is always normalized so that it has a positive z component.

2.3.4. 3 Other `LArFCalTBLine` Methods and Operators:

```
bool intersects(const LArFCalTBLine& L)
```

Returns `true` if the line has a point of intersection with line L.

```
bool yGreater(const LArFCalTBLine& L, double z)
```

Returns `true` if the line has a greater y coordinate at the given z position than does line L.

```
bool operator==(const LArFCalTBLine& L) const
```

Tests for equality of two lines. Two lines are equal if they have the same position vector and direction vector.

```
double findPerpDistToPoint(const std::vector<double>& point)
```

Returns the perpendicular distance between the given point and the line.

```
double findLengthParamAtPoint(const std::vector<double>& point)
```

If $(x, y, z) = (x_0, y_0, z_0) + (x_1, y_1, z_1) \cdot t$, returns the value of t at the intersection of the line and the perpendicular line passing through the given point.

```
double findMinDist(const LArFCalTBLine& L)
```

Returns the minimum distance between two lines.

```
std::vector<double> findXY(double z)
```

Returns the x and y coordinates, (x, y), at a given z position.

2.3.5 Reconstruction: The `LArFCalTBBPCBuilder` Algorithm:

The `LArFCalTBBPCBuilder` algorithm first retrieves the `LArFCalTBBPCRawContainer` for each event from `StoreGate`. It then converts the TDC signals into x and y beam positions using predetermined calibration constants $C1X$, $C2X$, $C1Y$, $C2Y$ and the following formula:

$x =$ (TDC_left - TDC_right - C2X)· C1X
$y =$ (TDC_down - TDC_up - C2Y)· C1Y
The beam position (x or y) is considered to be in overflow if either one of the two TCD signals is in overflow.

As discussed in section 1.3, corrections must be made to the coordinates to compensate for misalignment of the detectors. To this end, predetermined "offsets" are added to the x and y coordinates of BPC. An adjustment must also be made for the variation in table height for each set of runs, as discussed in section 1.4. The `LArFCalTBBPCBuilder` algorithm checks a database to see if the y offsets for BPCs 5 and 6 (those located on the table) have previously been calculated; if so, these are added to the y coordinates of these detectors to ensure the best possible alignment.

2.3.5.1 `LArFCalTBBPCBuilder` Job Options: The following job options are available to the user. As mentioned in section 2.1.3, the container keys should not normally be modified by the user.

`RawBPCContainers`**:** A list of keys (strings) for the input containers.

`ReconstructedBPCContainer`**:** A key (string) for the container for the reconstructed BPC objects.

`BPCDetectors`**:** A list of the IDs (strings) of the available BPCs can be given here.

`MaxPosDiff`**:** Because the BPCs are positioned in three sets of two closely spaced detectors, the x or y coordinates of each pair of BPCs should be very close to the same. If this is not the case, the particle may have scattered and the BPC coordinates will not produce a good line fit. If the x or y coordinates of a pair of detectors differ by more than `MaxPosDiff` (in mm), the beam position overflow flags for the detectors are set to `true` so that downstream algorithms will not attempt a line fit for that event.

`XOffsets`**:** A list of the predetermined offsets for the x coordinates of the detectors. These are added to the x coordinates of each corresponding BPC to correct misalignment. This list should appear in the same order at `BPCDetectors`.

`YOffsets`**:** A list of the predetermined offsets for the y coordinates of the detectors. The 5th and 6th values on the list should normally be zero,

as the offsets for the BPCs located on the table are retrieved from a database.

`TableHeight`: The approximate value of the table height at the current position. This will be corrected using the offsets from the database, calculated by the `LArFCalTBBPCTableOffsetCalc` algorithm. The values should be set as follows, depending on the beam position:

Table Heights	
Position	Height Setting (mm)
1, 2, 3	2.3
4L	-49.0
4H	39.0

2.3.6 Reconstruction: The `LArFCalTBBPCOffsetCalc` Algorithm:

The `LArFCalTBBPCOffsetCalc` algorithm is a special algorithm that should only be executed when the offsets for the BPC coordinates have not been determined. This algorithm calculates these offsets and prints them out in the `finalize` method. It is advisable to run the algorithm on several different runs to ensure that the values are consistent.

2.3.6.1 `LArFCalTBBPCOffsetCalc` Job Options: The following job options are available to the user.

`BPCs`: A list of IDs (strings) for the detectors.

`ZPositionsX`: A list of doubles, giving the z positions of the x components of the BPCs.

`ZPositionsY`: A list of doubles, giving the z positions of the y components of the BPCs.

`InitialXOffsets`: A list of doubles representing the starting values for the offsets. The first time the algorithm is executed these should all be zero, but once approximate values have been determine, it is advisable to run the algorithm again, varying the offsets by a small `StepSize` over a small range around these approximate values.

`InitialYOffsets`: See `InitialXOffsets`.

`StepSize`: The offsets are varied by increments of `StepSize`.

`OffsetRange`: The offsets added to the coordinates cover the range [`-OffsetRange`, `OffsetRange`] around the initial offsets.

`InputBPCContainer`: The key for the `LArFCalTBBPCContainer`.

2.3.7 Reconstruction: The `LArFCalTBBPCTableOffsetCalc` Algorithm:

The `LArFCalTBBPCTableOffsetCalc` algorithm first checks in the database to see if values for the y offsets for BPC 5 and BPC 6 have already been calculated for the current run. If they have, the algorithm does nothing. If they have not, the algorithm adds a variable offset to the y coordinates of BPC 5 and BPC, finding the offsets that give the best fit.

2.3.7.1 `LArFCalTBBPCTableOffsetCalc` Job Options: The following job options are available to the user.

`BPCs`: A list of IDs (strings) for the detectors.

`ZPositions`: A list of doubles, giving the z positions of all BPCs.

`StepSize`: The offsets are varied by increments of `StepSize`.

`OffsetRange`: The offsets added to the y coordinates cover the range [`-OffsetRange`, `OffsetRange`].

`InputBPCContainer`: The key for the `LArFCalTBBPCContainer`.

2.3.8 Reconstruction: The `LArFCalTBLineBuilder` Algorithm:

The `LArFCalTBLineBuilder` algorithm uses the methods described in section 1.2 to fit a straight line to the points in space determined by the BPC coordinates. Each `LArFCalTBBPCContainer` holds all of the reconstructed BPCs from one event, and the algorithm uses these to produce one `LArFCalTBLine` object.

2.3.8.1 `LArFCalTBLineBuilder` Job Options: The following job options are available:

`InputBPCContainer`: The key for the input BPC container.

`Line`: A key (string) for the `LArFCalTBLine` created by the algorithm.

`ZPositions`: A list of doubles representing the z positions of the detectors in the following order: BPC1X, BPC1Y, BPC2X, BPC2Y, ... BPC6Y.

`MinPoints`: The minimum number of BPCs giving valid coordinates required to create a `LArFCalTBLine`. BPC coordinates are not considered valid if the signal is in overflow.

2.3.9 Monitoring: The `LArFCalTBRawBPCMon` Algorithm:

The `LArFCalTBRawBPCMon` algorithm produces histograms of the raw ADC counts for each BPC, the sum of the x TDC signals, and the sum of the y TDC signals. Looking at the sum of the signals can provide a valuable tool for filtering events. The sum should be approximately constant, so a low sum can indicate two particles hitting the detector at the same time.

2.3.9.1 `LArFCalTBRawBPCMon` Job Options: The following job options are available to the user. As mentioned in section **??**, the container keys should not normally be changed from the defaults, nor should the histogram directory.

`RawBPCContainer`: The key (string) for the `LArFCalTBBPCRawContainer`.

`HistogramDirectory`: A string giving the directory where the histogram files should be written.

2.3.9.2 `LArFCalTBRawBPCMon` Histograms: The following histograms are produced by the `LArFCalTBRawBPCMon` algorithm, where i is the number of the BPC (1-6).

Histograms	
$1000i$	BPC i Raw ADC Counts - X
$1010i$	BPC i Summed TDC Counts - X
$2000i$	BPC i Raw ADC Counts - Y
$2010i$	BPC i Summed TDC Counts - Y

2.3.10 Monitoring: The `LArFCalTBBPCMon` Algorithm:

The `LArFCalTBBPCMon` algorithm produces XY scatter plots, both with and without veto. It also generates histograms of the beam position in x and y.

2.3.10.1 `LArFCalTBBPCMon` Job Options: The following job options are available to the user.

`BPCContainer`**:** The key (string) for the `LArFCalTBBPCContainer`.

`EventHeaderKey`**:** The key for the event header.

`HistogramDirectory`**:** A string giving the directory where the histogram files should be written.

2.3.10.2 `LArFCalTBBPCMon` Histograms: The following histograms are produced by the `LArFCalTBBPCMon` algorithm, where i is the number of the BPC (1-6).

Histograms	
$100i$	BPC i XY Scatter Plot
$200i$	BPC i XY Scatter Plot - No Veto
$300i$	BPC i XY Scatter Plot - Veto
$400i$	BPC i X Profile
$500i$	BPC i Y Profile

2.3.11 Monitoring: The `LArFCalTBLineCalibMon` Algorithm:

The `LArFCalTBLineCalibMon` algorithm produces histograms of the residuals of the linear fit.

2.3.11.1 `LArFCalTBLineCalibMon` Job Options: The following job options are available to the user.

`BPCContainer`**:** The key (string) for the `LArFCalTBBPCContainer`.

`Line`**:** The key for the `Line`.

`HistogramDirectory`**:** A string giving the directory where the histogram files should be written.

2.3.11.2 `LArFCalTBLineCalibMon` Histograms: The following histograms are produced by the `LArFCalTBBPCMon` algorithm, where i is the number of

the BPC (1-6).

Histograms	
$100i$	BPC i Line Fit Residuals
$200i$	BPC i Line Fit Residuals