

Seed Finding and Bezier Tracking in LArSoft - Technical Manual

Ben Jones, MIT

November 20, 2013

Abstract

This note describes the Seed Finding and Bezier Tracking TPC reconstruction algorithms in LArSoft. I give a conceptual overview of seeds and bezier tracks, provide instructions for how to use these objects for reconstruction and analysis applications, and then give a detailed technical description of the algorithms and data products involved in the SeedFinding and BezierTracking process.

Contents

1	Introduction	2
2	Conceptual Overview	2
2.1	Seeds	2
2.2	Bezier Tracks	4
3	Seed Finding	4
3.1	Using Seeds in LArSoft	4
3.1.1	Producing seeds with SeedFinderModule	4
3.1.2	Calling SeedFinderAlgorithm	5
3.2	The recob::Seed Data Product	6
3.3	Seeds in the Event Display	7
3.4	Seed Finding Performance	7
3.5	Technical Description of the SeedFinderAlgorithm	7
3.5.1	Public Interfaces	7
3.5.2	Parameters of the SeedFinderAlgorithm	7
3.5.3	Cluster overlap checking	11
3.5.4	Internal book keeping tables	11
3.5.5	The Seed finding loop	12
3.5.6	Determining seed centers and directions	12
3.5.7	Consolidating and extending seeds	13
3.5.8	Final attempt for unseeded combinations	13
4	Bezier Tracking	13
4.1	Using Bezier Tracks in LArSoft	13
4.1.1	Producing Tracks with BezierTrackerModule	13
4.1.2	Retrieving BezierTracks from the Event	14
4.2	The trkf::BezierTrack Analysis Object	14
4.2.1	Internal Structure and Organization of the Bezier Track	14
4.2.2	Bezier Interpolation and the BezierCurveHelper	15
4.3	Bezier Tracks in the Event Display	16
4.4	Bezier Tracking Performance	16
4.5	Technical Description of the BezierTrackerAlgorithm	16

4.5.1	Parameters of the BezierTrackerAlgorithm	17
4.5.2	Producing the organized hit collection	18
4.5.3	Cluster Overlap Checking and Hit Filtering	18
4.5.4	Making Bezier Tracks	18
4.5.5	Overlap Filtering and Track Joining	20
4.5.6	Bezier Vertexing	21
4.5.7	Bezier Calorimetry	21

5	Conclusions	21
----------	--------------------	-----------

1 Introduction

Liquid argon time projection chambers give an unprecedented level of position resolution with which to study neutrino interactions in the 10 MeV - 100 GeV energy range. There are many benefits provided by this high resolution, which include the following: electromagnetic showers can be classified into electron and photon induced cases by examination of the first few millimeters of the evolving shower; low energy interaction products and nuclear fragments around an event vertex, which in previous detectors would have been simply labelled as “vertex activity”, can be analyzed to explore the nuclear physics of neutrino interactions; and assumptions relating to lepton kinematics for the extraction of differential neutrino scattering cross sections can be relaxed by examining the hadronic final state of a neutrino interaction in detail.

However, this additional level of detail presents major challenges for event reconstruction, especially when coupled with the problem of transforming from the TPC coordinate system, which consists of several planes of partially degenerate yet still partially incomplete information, into the 3D coordinate system. Whilst much information can be obtained from 2D image analysis, the transformation into 3D is a necessary precursor to pitch corrected calorimetric reconstruction of an event.

This technote describes one approach to finding well specified 3D regions of event activity, called Seed Finding, and one tracking algorithm which makes use of these regions called Bezier Tracking. These algorithms have been developed for the MicroBooNE experiment within the LArSoft framework. We begin with a brief conceptual overview of both seeds and bezier tracks in section 2. Sections 3 and 4 give detailed descriptions of the seed finding and bezier tracking algorithms, respectively. In both cases the section begins with a “tutorial” style subsection on how an analyzer / algorithm developer can make use of seed and bezier track information within LArSoft. A description of the relevant data products is given, followed by a brief overview of the performance of each algorithm using simulated MicroBooNE events, and finally a full technical description of the algorithm implementation is given.

We assume familiarity with the general design principals of the LArSoft framework, including the roles of producer modules, analyzer modules, data products and fhicl configuration scripts. Familiarity with the TPC coordinate system and its relation to the 3D coordinate system is also useful. In order that divergences between the code repository and the code described in this note can be understood, the version numbers of each LArSoft package at the time of writing are given in figure 1. A list of LArSoft files which have significant sections devoted to Seed Finding and Bezier Tracking is given in table 1.

2 Conceptual Overview

2.1 Seeds

Seeds represent unambiguous, straight, three dimensional regions of event activity. A seed is defined by a three dimensional point, and a direction vector with a length which extends in both directions. This is shown as a cartoon in figure 2, left. The goal of the SeedFinder algorithm is to identify regions where the TPC activity across all planes in the TPC indicates an unambiguous straight section, and provide information about these sections as instances of the `recob::Seed` data product.

root / trunk

Name	Size	Revision
AnalysisAlg		4887
AnalysisBase		5011
AnalysisExample		5085
CalData		5064
Calorimetry		5039
ClusterFinder		5040
DetSim		5040
EventDisplay		5105
EventFinder		5039
EventGenerator		5039
Filters		5039
Genfit		4887
Geometry		5073
HitFinder		5087
LArG4		5074
LArPandoraAlgorithms		5098
LArPandoraInterface		5100
MCCheater		5046
Monitoring		4887
OpticalDetector		5080
OpticalDetectorData		4887
ParticleIdentification		5039
PhotonPropagation		5040
RawData		4952
RecoAlg		5107
RecoBase		5060
RecoObjects		5106
SRT_LAR		5104
ShowerFinder		5039
SimpleTypesAndConstants		4947
Simulation		5075
SummaryData		4887
TrackFinder		5097
TriggerAlgo		5081
Utilities		5092
VertexFinder		5040
setup		5091

Figure 1: Version numbers of various packages in the LArSoft repository at the time of writing this note

Package	File	Description
RecoBase	Seed.h / .cxx Track.h / .cxx	recob::Seed data object. Seed data product, including geometrical methods recob::Track data product. Bezier tracks are stored in this format
RecoObjects	BezierCurveHelper.h / .cxx BezierTrack.h / .cxx	Helper object for finding bezier curve trajectories between seeds Bezier track analysis object
RecoAlg	SeedFinderAlgorithm.h / .cxx BezierTrackerAlgorithm.h / .cxx	Set of algorithms for finding and refining Seeds based on hit collections Set of algorithms for finding and refining Bezier Tracks from hits, clusters or seeds
TrackFinder	SeedFinderModule.h / .cxx BezierTrackerModule.h / .cxx SeedAna.h / .cxx *	LArSoft module to find seeds and store into the event LArSoft module to find bezier tracks and store into the event Ana module to check performance of Seed Finding against MC truth
Calorimetry	BezierCalorimetry.h / .cxx	Module for performing pitch-corrected bezier calorimetry
EventDisplay	RecoBaseDrawer.h / .cxx	Methods for drawing seeds and bezier tracks in the larsoft event display
Key: Seeding BezierTracking Both		* SeedAna module written by Herb Greenlee

Table 1: List of LArSoft files with large sections devoted to Seed Finding and / or Bezier Tracking

2.2 Bezier Tracks

A BezierTrack represents a three dimensional trajectory which is formed by interpolating between seeds. A cartoon is shown in figure 2, right. A BezierTrack may be formed from any combination of one or more seeds. A single-seed bezier track is a straight line, and has a trajectory which is exactly identical to the seed from which it was formed. A multiple-seed bezier track is in general a curved object. The track trajectory is expressed in terms of a single parameter s , which varies smoothly from 0 to 1 along the length of the track. The trajectory can be sampled at any S value, and intervals of uniform S represent intervals of uniform distance along the track (so, for example, $s=0.25$ will always give the point which is one quarter of the way along the trajectory). The goal of the BezierTracker algorithm is to identify sets of seeds which can be sensibly interpolated and produce `trkf::BezierTrack` analysis objects, which are each appropriately parameterized by a smoothly varying S coordinate.

3 Seed Finding

3.1 Using Seeds in LArSoft

Seeds are constructed from sets of hits using the `SeedFinderAlgorithm` in the `RecoAlg` package. There are two ways to produce seeds for use in an algorithm / analysis.

3.1.1 Producing seeds with SeedFinderModule

The first way to produce seeds is to use the `SeedFinderModule`. This is the recommended method for most applications, and only needs to happen once in the reconstruction chain. The module has two parameters which specify what raw materials to use for producing seeds (`InputSource` and `InputModuleLabel`), and the module configuration accepts a `parameterset` for the `SeedFinderAlgorithm` which specifies all the parameters relating to the seed finding operation (described in section 4.5.1).

This module can be run in two modes, dictated by the `InputSource` parameter. If configured with `InputSource=0`, the module will retrieve a hit collection from the event and produce all possible seeds from this collection. If configured with `InputSource=1`, the module will retrieve a cluster collection from the event and produce seeds based on combinations of overlapping clusters. The latter mode allows the `SeedFinder` to use the results of pattern finding algorithms in 2D, which have collected hits corresponding to distinct

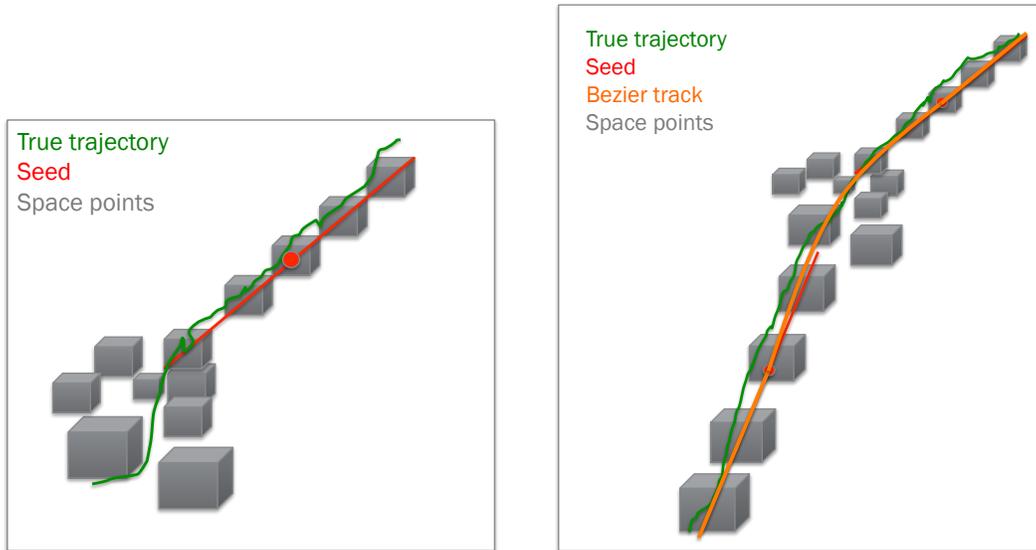


Figure 2: Cartoon showing a seed (left) and bezier track (right), overlaid on the true particle trajectory, and the relevant set of possible spacepoints obtained from the TPC

features in the event into clusters. This both improves the computational speed of the SeedFinderAlgorithm by reducing the combinatorics of hit-overlaps, and can also help mitigate the effect of fake spacepoints from spurious cluster overlaps near a vertex. In both source modes, the `InputModuleLabel` parameter specifies the name of the producer from which the hits or clusters originated, and is used to look up the relevant collection in the event record. Once seeds have been produced and stored in the event, they can be retrieved using the standard ART tools, which are described in the LArSoft wiki. A sample of code lifted from SeedAna is given below.

```
art::Handle< std::vector<recob::Seed> > seedh;
evt.getByLabel(fSeedModuleLabel, seedh);
if(seedh.isValid()) {
    int nseed = seedh->size();
    for(int i = 0; i < nseed; ++i) {
        art::Ptr<recob::Seed> pseed(seedh, i);
        pseed->[etc...]
    }
}
```

3.1.2 Calling SeedFinderAlgorithm

The second way of making seeds is to call the SeedFinderAlgorithm directly. This approach is favored by the Bezier Tracker and Kalman Filter algorithms, since these use an iterative method which involves searching for seeds in various different hit collections, rather than the entire collection for the event or cluster combination. There are two public methods of SeedFinderAlgorithm supplied for this purpose. First, the `GetSeedsFromUnsortedHits` method:

```
std::vector<recob::Seed> GetSeedsFromUnsortedHits (
    art::PtrVector<recob::Hit> const& HitsFlat,
    std::vector<art::PtrVector<recob::Hit> >& CataloguedHits,
    unsigned int StopAfter)
```

This method accepts a `PtrVector` of `recob::Hits` and returns a vector of `recob::Seed` objects. The first argument supplies the list of hits to be used for seed finding. The second argument is a reference to a vector of `PtrVectors` to be filled, returning `art::Ptrs` to the original hits but organized in terms of their association to each seed (for example, the second seed in the return vector is associated with the hits collected in the second entry in `CataloguedHits`). This vector should be supplied empty to avoid unpredictable behaviour. Finally, the `StopAfter` parameter allows the user to specify how many seeds to search for in this collection. In situations where only one seed is required from a particular set of hits, significant computational time may be saved by stopping the search for seeds after one is found. If zero or no value is supplied, seed finding continues until the hit collection is exhausted. In general, seeds are found from high to low Z (= collection plane wire coordinate).

The `GetSeedsFromSortedHits` method is designed to find seeds in cluster combinations. In this case, a more organized collection of hits is supplied. The `SortedHits` object provides hits organized into collections in each view.

```
std::vector<std::vector<recob::Seed> > GetSeedsFromSortedHits(
    std::vector<std::vector<art::PtrVector<recob::Hit> > > const& SortedHits,
    std::vector<std::vector<art::PtrVector<recob::Hit> > >& HitsPerSeed,
    unsigned int StopAfter)
```

The index of the top level vector in `SortedHits` is an integer representing the view, which for `MicroBooNE` can be either 0 for `geo::kU`, 1 for `geo::kV` or 2 for `geo::kW`. This splits the collection at the highest level into hits from each view. The next level of organization is a vector of `PtrVectors`, with each element representing one cluster in the given view. Finally, each cluster contains many hits, which are stored as the elements of the lowest level `PtrVector`.

The returned seed vector from this method has an additional level of structure relative to the unsorted method, with one vector of seeds being returned for each cluster combination. This vector has one entry per possible overlap, and where no seeds are produced or there is no 3D overlap, the entry is an empty seed vector. As such, the size of the top level return vector is always $nU * nV * nW$, where nU , nV , nW are the numbers of clusters which were supplied in each view, even in the extreme case where no seeds are actually found and every element corresponds to an empty seed vector. The `HitsPerSeed` object has the same interpretation as in the unsorted method but with a similar per-cluster-combination structure. The `StopAfter` parameter provides the option to search for a finite number of seeds per cluster combination (for example, `StopAfter=1` would return at most one seed per UVW cluster combination). An example showing how to produce the structured hit object from an unstructured set of clusters can be found in the `SeedFinderModule::GetSortedHitsFromClusters` method.

3.2 The `recob::Seed` Data Product

The `recob::Seed` is a very simple data product. The most important data members are a `SeedPoint`, a `SeedDirection` and errors on both of these values. These data members, their getter and setter methods as well as the seed constructor are self explanatory from the `Seed.h` header file. Note that the present implementation of `SeedFinderAlgorithm` does not set values for the seed errors, but they are included in the data structure for possible future use cases. There is one further data member, the boolean flag `flsValid`. This flag is set to true for any seed constructed with a point and a direction supplied, but false for a seed constructed using the default constructor with these values not set. This is used for book-keeping during seed finding, and in general no seeds with `IsValid=false` should ever be stored into the event record. The value of this flag can be controlled after seed construction using the `SetValidity` method.

As well as simple data members, the `Seed` class provides various geometrical methods for evaluating the relationship between seeds and points in 3D space. These are methods listed and illustrated in table 2. As a `LArSoft` design principal, methods in a stored data product should not require interface to external services to perform calculations, so the geometrical methods in the seed object are limited to strictly 3D geometry calculations, and cannot evaluate projections into wireplane coordinate systems or find distances

to `recob::Hits`, etc. These more advanced geometrical operations which require information about the TPC geometry are generally performed in the Geometry service or within algorithm or modules.

3.3 Seeds in the Event Display

Seeds which have been stored in the event can be drawn in the LArSoft event display by setting the `DrawSeeds` parameter to true and setting the `SeedModuleLabel` string. In the two dimensional and ortho event display views, seeds are shown projected into the relevant two dimensional coordinate system as straight lines with the central position marked by an empty circle. When fully zoomed out, the linear extent of a small seed may not be visible, but the marker will remain large enough to show where the seed is placed. It should be remembered that even though they can be displayed in any 2D projection, seeds are 3D objects and each seed is present once in every view. In the three dimensional event display view, seeds are marked with white 3D lines which extend out of white spheres, marking the central seed point. If `SeedFinderModule` has been successful, a population of seeds which mimics the true 3D distribution of tracks should be visible, as shown in figure 3. In general, seeds will not extend all the way to the event vertex, since there is always a region of somewhat ambiguous directionality where several tracks emerge from a single point. It is the task of later reconstruction algorithms to extrapolate from regions of well-defined directionality back towards the event vertex.

3.4 Seed Finding Performance

The `SeedAna` module, developed by Herb Greenlee, can be used to evaluate the efficiency of seed finding for simple events. A track is classified as successfully seeded if a seed is found along the length of the track with a sufficient collinearity to the true trajectory at that point (the default collinearity cut $\cos(\theta) > 0.99$) and the seed center is within a short distance (default cut 2cm) of a true trajectory point. Only particles above a lower kinetic energy cut (default 50 MeV) are counted. The performance of the `SeedFinderAlgorithm` was evaluated using a monte carlo sample of 10,000 isotropic muons in the energy range 0.1 to 2.0 GeV. The algorithm has an efficiency of better than 99% for tracks longer than 20cm. For shorter tracks, some efficiency is lost for tracks parallel to the drift or wire directions. The efficiency of this algorithm for this sample is shown in figure 4, as compared with the previous version of the `SeedFinding` algorithm (not described in this note).

3.5 Technical Description of the `SeedFinderAlgorithm`

3.5.1 Public Interfaces

The `SeedFinderAlgorithm` is generally called via one of the two public interfaces described in section 3.1.2. Both of these call the main private method of the `SeedFinderAlgorithm`, `FindSeeds`, which acts on a flat, unstructured collection of hits. The `GetSeedsFromUnsortedHits` method is a simple wrapper function, which passes the arguments with which it is called directly to the `FindSeeds` method. Refer to section 3.1.2 for the meanings of these parameters. The `GetSeedsFromSortedHits` method calls `FindSeeds` once for each possible cluster combination, formed from one cluster in the U view, one in the V view and one in the W view. Technically this is achieved by producing a flat hit collection for each cluster combination and calling `FindSeeds` once for each combination.

3.5.2 Parameters of the `SeedFinderAlgorithm`

Table 3 gives the configurable parameters of the `SeedFinderAlgorithm` and their default values, as well as a short description of each. Some of the parameters are used for more than one purpose in the algorithm. A design decision was made to minimize the number of free tunable parameters, and use the same parameter for two purposes where the physical interpretation of the parameter is identical.

Within the `SeedFinderAlgorithm` parameterset, a parameterset for `SpacePointAlg` is also supplied. It is necessary for the correct functionality of `SeedFinderAlgorithm` that within this parameterset, `PreferColl` is

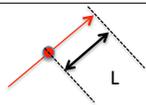
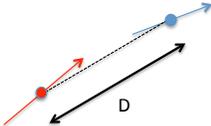
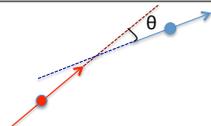
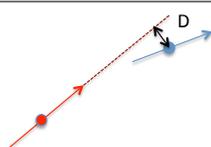
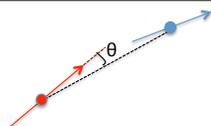
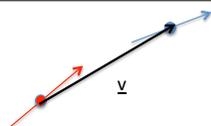
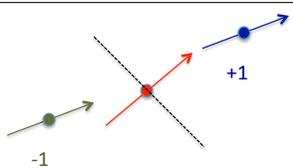
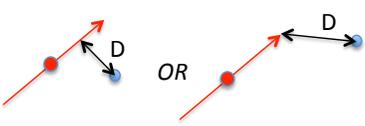
Method	Return type	Parameters	Sketch
GetLength	double	-	
GetDistance	double	Seed const& AnotherSeed	
GetAngle	double	Seed const& AnotherSeed	
GetProjDiscrepancy	double	Seed const& AnotherSeed	
GetProjAngleDiscrepancy	double	Seed const& AnotherSeed	
GetVectorBetween	double* xyz	Seed const& AnotherSeed	
GetPointingSign	int (always +/- 1)	Seed const& AnotherSeed	
Reverse	Seed	-	
GetDistanceFrom	double	SpacePoint	

Table 2: Geometrical methods of the `recob::Seed` object (The seed from which the method is called is shown in red)

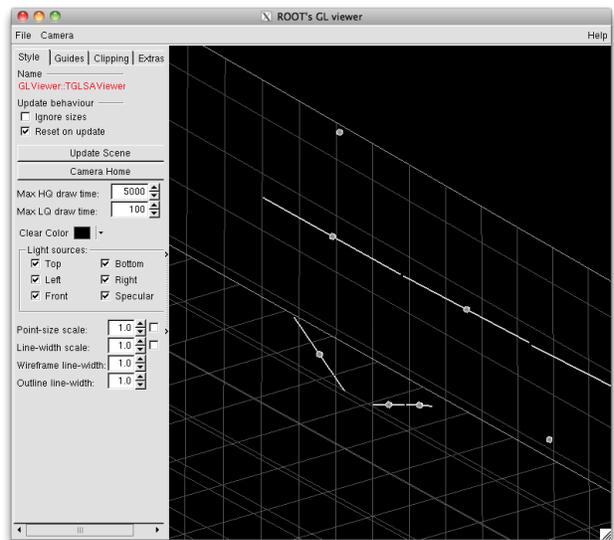
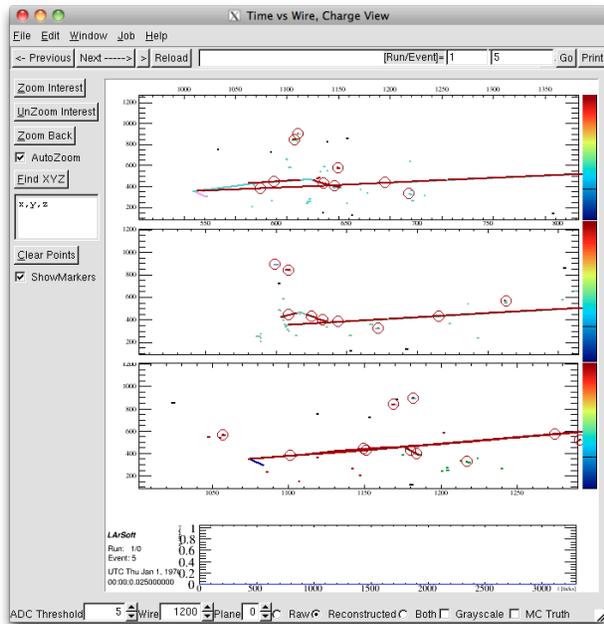
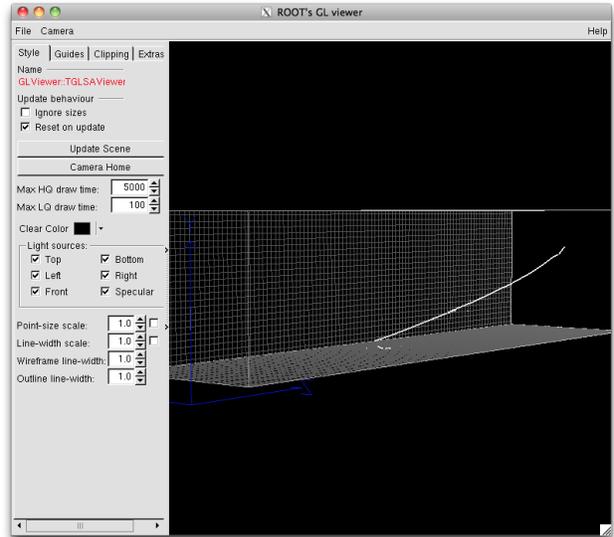
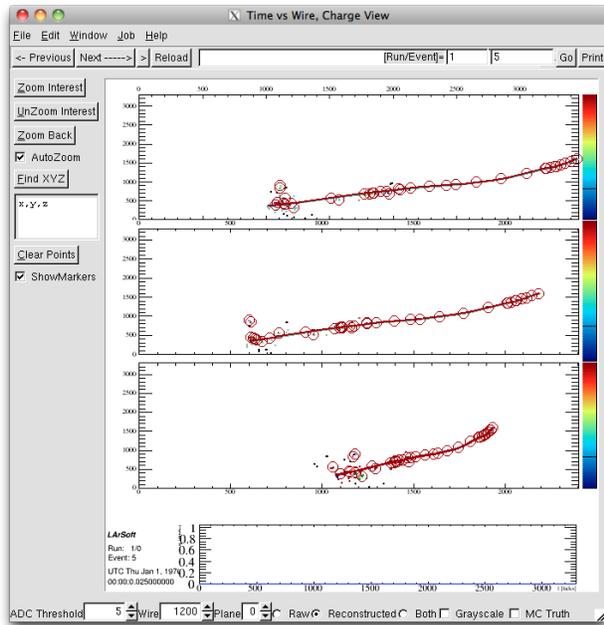


Figure 3: Seeds displayed in the 2D (left) and 3D (right) event display views, showing the full event (top) and zoomed into the vertex region (bottom)

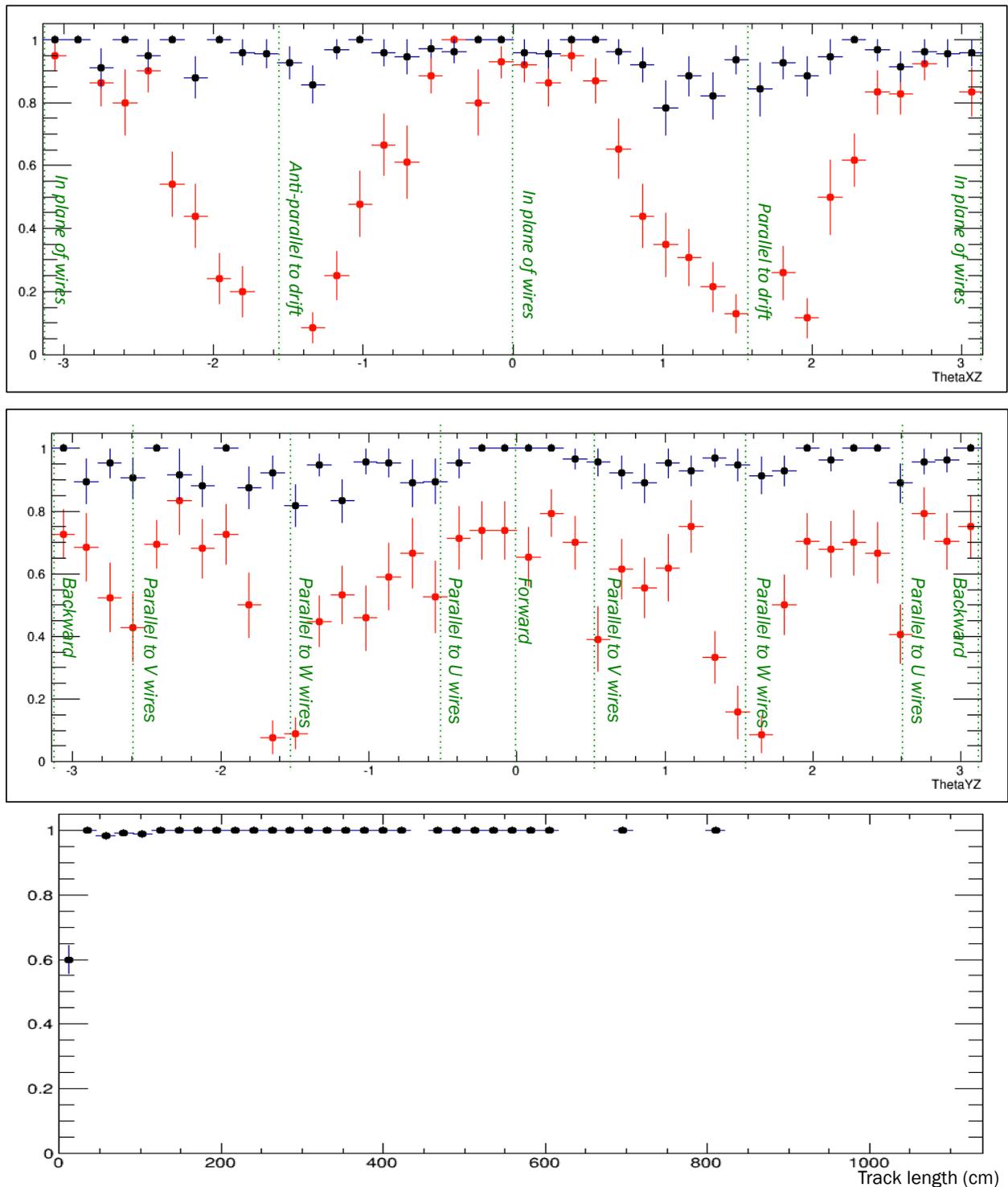


Figure 4: Efficiency plots for the SeedFinderModule, evaluated on a low energy isotropic muon sample. Black shows the performance of the latest version of the algorithm (described in this note), compared to red points which show the previous version of the SeedFinderAlgorithm (depreciated in October 2013).

Name	Type	Default	Description
InitSeedLength	double	2.	Distance to look around high Z point for spacepoints (cm)
MinPointsInSeed	int	4	Minimum number of spacepoints in collection to look for seeds
Refits	int	5	Number of times to extend and refit seed
HitResolution	double	1.	Number of hit widths to assume for hit extent
OccupancyCut	double	0.8	Fraction of channels in each view which must have activity in seed
MaxViewRMS	vector<double>	[2.,2.,2.]	Max allowable RMS of hits around seed
ExtendSeeds	bool	true	Flag determines whether seeds are extendable or fixed length
LengthCut	double	0	Minimum seed length cut after consolidation
SpacePointAlg	pset	N/A	Parameterset for the spacepoint algorithm

Table 3: Configurable parameters of the SeedFinderAlgorithm

Name	Structure	Purpose
OrgHits	OrgHits[View][Channel] = {ihit_1, ihit_1...}	Catalogue all hits by view and channel
HitsPerSpacePoint	HitsPerSpacePoint[ispt] = {ihit_1, ihit_2...}	Which hits go with which spacepoint
SpacePointsPerHit	SpacePointsPerHit[ispt] = {ispt_1, ispt_2...}	Which spacepoints go with which hit
HitStatus	HitStatus[ihit] = 0,1,2	What is the present status of each hit
PointStatus	PointStatus[ispt] = 0,1,2,3	What is the present status of each spt

Table 4: Lookup and status tables used for SeedFinder internal book keeping. “ihit” are integers giving positions in the HitsFlat vector of hits. “ispt” are integers giving positions in the spts vector of spacepoints.

set to true to ensure spacepoints are supplied sorted in Z, and all three views are enabled (EnableU = true, EnableV = true, EnableW = true). Best results have been obtained with unfiltered spacepoints (Filter = false) and requiring 3 view coincidences only (MinViews = 3).

3.5.3 Cluster overlap checking

The FindSeeds method accepts a flat collection of hits, in a PtrVector called HitsFlat. This vector is passed by reference to maximize efficiency.

The first operation performed by the FindSeeds method is the production of spacepoints based on the provided hit collection using an instance of SpacePointAlg, configured with the provided parameterset. If there are less than MinPointsInSeed spacepoints produced, the seed finding for this hit collection is stopped and an empty vector is returned. This ensures that seeds are only searched for in cluster combinations with some spatial overlap.

3.5.4 Internal book keeping tables

Next, several book-keeping lookup tables are filled, which enable the various hit, spacepoint and seed comparisons which follow to be performed efficiently. These are described below and summarized in table 4.

The OrgHits table indexes all hits in the input vector by their channel and view. The lowest level elements of the table are integers which give the indices of hits in the HitsFlat input vector, and these are sorted into collections corresponding to each channel and view. This allows an efficient lookup of all hits within one specified channel and view, which is an operation performed many times within the seed finding algorithm.

The SpacePointsPerHit table has one entry per spacepoint, with each entry being a vector of integers which correspond to indices in the HitsFlat vector, indicating which hits are associated with every spacepoint found.

The HitsPerSpacePoint vector is the inverse lookup table, with one vector per hit which stores integers corresponding to indices in the spts spacepoint vector. Since the recommended mode of operation of the SpacePointAlg within SeedFinderAlgorithm is to use unfiltered spacepoints, in general each hit in all views can feature in multiple spacepoints.

The `HitStatus` vector has one element for each element in the `HitsFlat` vector, and tracks the status of each hit. A status of 0 indicates that the hit is available for forming a seed. A status of 1 indicates that the hit has already been used in a seed. A status of 2 indicates that this hit is being considered for the present work-in-progress seed. Each hit may appear in only one seed per cluster combination. However, if a cluster `a` in view `A` has overlaps with clusters `b1` and `b2` in view `B`, each hit in cluster `a` may be used once in combination `{a b1}`, and once in combination `{a b2}`, for example.

The `PointStatus` table has one element for each spacepoint and tracks the status of each spacepoint. A value of 0 indicates that this spacepoint is unused and available for seed finding. A value of 1 indicates that this spacepoint has been used in a seed already. A value of 2 indicates that the seed was considered for seeding but discarded and should not be used again. A value of 3 is a flag which can be given to the first element of the `PointStatus` vector, and which indicates that the final spacepoint has been used and that seed finding should cease.

3.5.5 The Seed finding loop

Seed finding is an iterative process which continues until either enough seeds have been found (if the `StopAfter` argument is nonzero), or when all spacepoints have been exhausted. This subsection gives an overview of the procedure, and subsequent subsections treat each step in more detail.

In each round of the seed finding loop, we search for a seed around the remaining spacepoint with the highest `z` coordinate using the `FindSeedAtEnd` method.

If we find a seed there, we call the `ConsolidateSeed` method which fills in any hits which were missed in the first pass, and makes some seed quality checks. If the `ExtendSeeds` parameter is set to true, the `ConsolidateSeed` method will also attempt to extend the seed in both directions. If the `Refits` parameter has a value larger than 0, we may refit the seed to the hits we collected during consolidation and try extending again.

Once the seed is consolidated, all hits which were collected for this seed, having been tagged with a `HitStatus` of 2, are given a status 1 and removed from further consideration for future seed finding. All spacepoints corresponding to these hits are also removed from consideration by setting their entry in the `PointStatus` table to 1. If we did not find a valid seed around the highest-`z` spacepoint, this single point is removed from consideration by setting its entry in the `PointStatus` table to 2. In subsequent iterations we consider only hits and spacepoints with status 0 which are still available for seeding. If the `FindSeedAtEnd` method cannot find any valid spacepoints, it will set the first entry in the `PointStatus` table to 3, indicating that the spacepoint vector has been exhausted and seed finding for this cluster combination is complete.

3.5.6 Determining seed centers and directions

There is no single definitive way of evaluating the 3D direction implied by a set of spacepoints or a set of hits. The `SeedFinderAlgorithm` has gone through several versions using different methods to evaluate seed coordinates. The most powerful method so far involves making a linear least-squares regression on the hits which have been collected in each view, and using the two most populated views to produce a 3D center and direction hypothesis. This code is implemented in the `GetCenterAndDirection` method of the `SeedFinderAlgorithm`.

Methods using spacepoint coordinates to establish seed directions tend to fail because degenerate spacepoints lead to directional biases along the wire directions. Principal component analysis of hit coordinates gave promising results, but cannot weight the hit pulls according to the widths of hits, and so suffers in situations where hits are particularly wide, as in high angle tracks. A weighted linear least squares approach applied to the hits in wire/time coordinates can both incorporate hit widths and minimize degeneracy based biases.

```
GetCenterAndDirection(
    art::PtrVector<recob::Hit> const& HitsFlat,
    std::vector<int>& HitsToUse,
    TVector3& Center,
```

```
TVector3& Direction,  
std::vector<double>& ViewRMS)
```

This method takes as inputs a reference to the vector of all hits (`HitsFlat`), and a list of integers (`HitsToUse`) indicating which elements of this vector should be considered in the least squares minimization. It returns by reference a `TVector3` representing the 3D seed `Center`, a `TVector3` representing the 3D seed `Direction`, and a vector of doubles giving the RMS of the hits from the seed spine in each view (`ViewRMS`). Seeds for which the RMS does not pass the cut defined by the `MaxViewRMS` parameter will later be discarded.

3.5.7 Consolidating and extending seeds

The `ConsolidateSeed` method transforms both ends of the seed under consideration into wire / time coordinates in each view. All hits in the cluster combination which are in the relevant channel range are looked up using the `OrgHits` table, and checked to see if they lie within `HitResolution` times the hit width from the seed spine in the time direction. In this way, any hits which were not incorporated in the initial fit are included. The seed is shortened such that it only extends to the furthest hits in each direction, since even though the seed has strong directionality over the `InitLength` distance, it may only have hits populating a sub-section of that length before consolidation and need to be contracted. If the seed does not have hit activity in a large enough fraction of its channels in each view (defined by the `OccupancyCut` parameter), it is discarded.

If the `ExtendSeeds` parameter is set to true, the seed is extended linearly in each direction, channel-by-channel in each view. If an available (`HitStatus=0`) hit exists with a time which is within `HitResolution` times the hit width from the seed spine in the relevant channel then the seed extension continues. If a channel is met with no hit consistent with the seed, this limits the extension possible in this view. The final extension at each end of the seed is limited by the most constraining view in each case.

If the `Refits` parameter is greater than zero, the seed center and direction are then recalculated based on the total set of hits now associated with the seed, and consolidation and extension is repeated a number of times determined by the value of the `Refits` parameter.

3.5.8 Final attempt for unseeded combinations

If we met the cluster overlap condition but seeding failed, the algorithm makes one final attempt to find a strong directionality using all hits in all views, rather than a subset. This can help to find seeds in collections which are too short to successfully produce seeds with the iterative method, or which are inclined highly relative to the wireplanes so have widely spaced but still very collinear hits. If the RMS displacement of hits in any view from the seed spine obtained with this method is too large (determined by the `MaxViewRMS` parameter) then the seed will be discarded.

4 Bezier Tracking

4.1 Using Bezier Tracks in LArSoft

4.1.1 Producing Tracks with BezierTrackerModule

BezierTracks are produced by the LArSoft EDProducer `BezierTrackerModule`, and stored into the event. This producer takes clusters as an input, from a producer specified by the `ClusterModuleLabel` parameter. The module also accepts a parameterset for the `BezierTrackerAlgorithm` which contains all the parameters relating to bezier track finding and are described in section 4.5.1.

As well as producing a collection of tracks, the `BezierTrackerModule` also produces 3D vertices based on extrapolation of tracks found, associations between tracks and these vertices, and associations from tracks to the hits from which they were constructed.

4.1.2 Retrieving BezierTracks from the Event

The data object stored in the event for each bezier track is a `recob::Track`. This object should **NOT** be used interchangeably with a track as produced by any other tracking algorithm. Instead, after retrieving the track collection from the event, the `recob::Track` must be converted back into `BezierTrack` objects in order to obtain the correct bezier trajectory. An example is shown below.

```
std::vector<trkf::BezierTrack> BTracks;
art::Handle< std::vector<recob::Track> > btbcol;
evt.getByLabel(ModuleLabel, btbcol);
for(unsigned int i = 0; i < btbcol->size(); ++i){
    art::Ptr<recob::Track> btb(btbcol, i);
    BTracks.push_back(trkf::BezierTrack(*btb));
}
```

After running this code snippet, the vector of `trkf::BezierTrack` objects called `BTracks` can be used for analysis. Using the stored `recob::Track` object in analyzers designed for tracks proceed by other algorithms will lead to inaccurate and unpredictable results. This slightly confusing design scheme was forced upon the `BezierTracker` by LArSoft policy early in the algorithms development, despite the best efforts of the author to implement a more appropriate class structure.

4.2 The `trkf::BezierTrack` Analysis Object

The `BezierTrack` object contains many powerful geometrical methods. The most important are the `GetTrackPoint` and `GetTrackDirection` methods, which gives the track trajectory points and directions as a function of the position `S` along the track (see section 2.2). The `GetTrackPointV` and `GetTrackDirectionV` methods are wrappers for the `GetTrackPoint` and `GetTrackDirection` methods which return a `TVector3` rather than an array of doubles, for coding convenience.

The `GetProjectedPointUVWX` and `GetProjectedPointUVWT` methods provide the track trajectory as a function of `S` as projected into the TPC coordinate system for each view (for tpc `t` and cryostat `c`, as per LArSoft conventions). Each method returns the coordinates of the point in each wireplane. The coordinate perpendicular to the wires can be given either as a position, as in `GetProjectedPointUVWX`, or as a drift time in ticks, as in `GetProjectedPointUVWT`.

The `GetCurvature` method returns the local rate of change of the track direction at point `s`. The `GetRMSCurvature` method calculates the average curvature along the track and may find application in the measurement of track momentum through multiple scattering.

The `GetLength` method returns the total length of the bezier track, evaluated by finding the distance between many evenly space points along the trajectory.

There are a suite of `GetClosestApproach` methods, which give the minimum 3D distance between the track and a hit, spacepoint, seed, vector or wire/ time coordinate respectively. There are also two `GetClosestApproaches` methods which accept a vector of hits, and are optimized for finding the closest approaches between the track trajectory and many hits at once.

There are several calorimetry methods which should presently be considered work-in-progress, and can return the pitch-corrected dQ/dx along the length of the track based on a set of hits in the collection view.

4.2.1 Internal Structure and Organization of the Bezier Track

The `BezierTrack` object stores its trajectory information in two complementary ways. Firstly, as a set of trajectory points and directions, as the data members of the `recob::Track` base class. Secondly, as a vector of `recob::Seeds`, the private `fSeedCollection`, which are used for interpolation calculations.

In order to ensure the track is interpolated to the very end of the outermost seeds, two extra seeds of very short length are padded onto the end of the `SeedCollection` stored in `BezierTrack` during construction from a seed vector. They are removed when the seed collection is accessed from the public `GetSeedVector` method, so the returned collection represents the same seed vector which was used to construct the bezier track.

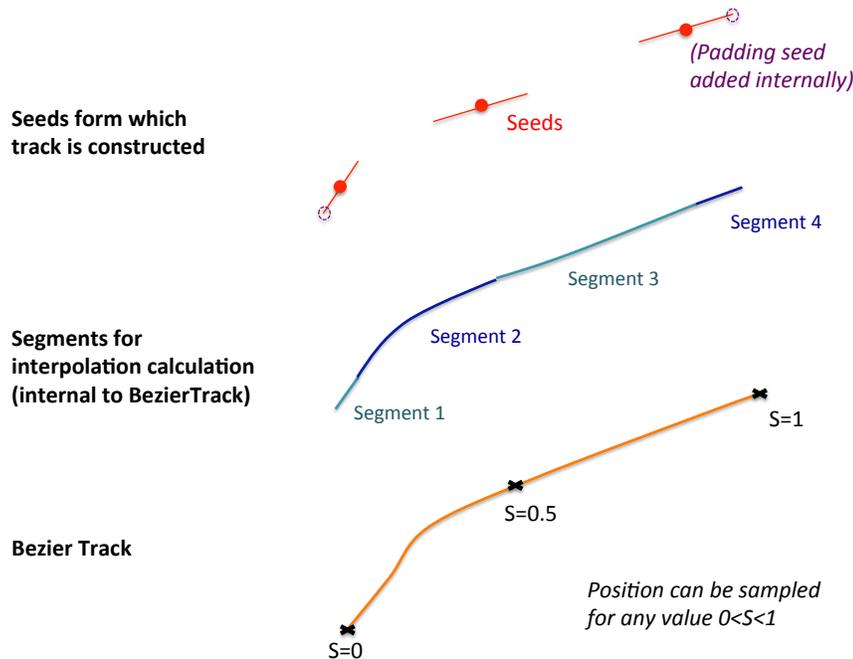


Figure 5: Sketch showing the seeds, internal segments and parameterized curve of a BezierTrack

When constructed from a set of seeds, after filling the seed collection, the BezierTrack constructor calls the `CalculateSegments` method. This method establishes the length of each interpolated segment and hence apportions regions of S along the length of the track. When a track point at position S is requested via the `GetTrackPoint` method, the segment in which this point lies is determined using the pre-calculated cumulative lengths. Then the local S value within this segment is determined, and the bezier trajectory point is calculated along the curve connecting the two relevant seeds. Some of these concepts are shown in the cartoon sketch of figure 5.

The underlying `recob::Track`, which contains all the information necessary to reproduce the track, can be acquired using the `GetBaseTrack` method. This method is called to produce `recob::Tracks` to store in the event. The original BezierTrack can be recreated from the `recob::Track` by using the constructor `BezierTrack(recob::Track)`.

The track can be reversed (swapping the order of the seeds and reversing each one) using the `Reverse` method. A partial bezier track between limits of S can be obtained with the `GetPartialTrack` method. Note that the outermost segments of the partial track are approximated with straight lines, and if both the low and high S values are within the same segment, the resulting partial track will have no curvature. A track trajectory represented as N spacepoints can be acquired with the `GetSpacePointTrajectory(N)` method.

4.2.2 Bezier Interpolation and the BezierCurveHelper

The calculation of bezier trajectory points is performed by a helper object in the RecoObjects package called BezierCurveHelper. Given two seeds the BezierCurveHelper can be used to extrapolate to any point along the bezier segment connecting them. A bezier curve is a polynomial curve which interpolates a series of vector points in terms of a single parameter. The curve passes through the two end points, but in general does not pass directly through the mid points although these determine the trajectory. Two bezier curve definitions have been tested : a cubic curve using the seed centers, and two intermediate points from direction vectors projected one third of the way along the line dividing the seed centers, and a quartic curve formed from the seed centers and three intermediate points, corresponding to the ends of the seed and their projected

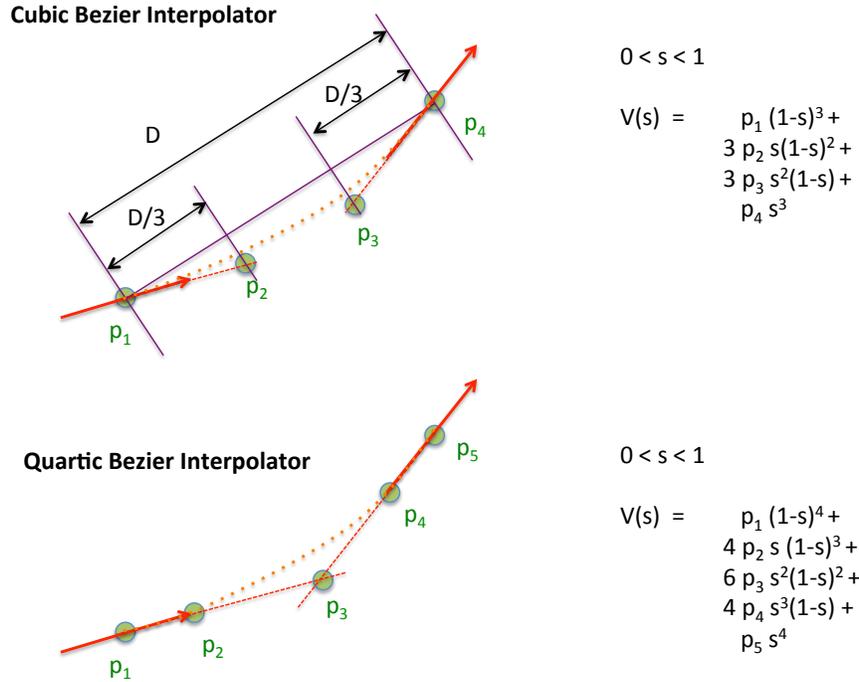


Figure 6: The two BezierCurveHelper interpolation schemes. Quartic interpolation is currently favored for bezier tracking.

point of closest approach. These two interpolation schemes are illustrated in figure 6. The quartic definition gives a good approximation to tracks with significant curvature, as the track is forced to run along the seed vectors for the first and last sections. However, this interpolation can behave badly when the seeds are almost parallel and the projected crossing point is distant. For this reason, the cubic interpolator is favored for Bezier tracking, although both methods are available in the BezierCurveHelper object.

4.3 Bezier Tracks in the Event Display

The event display can draw BezierTracks in both 2D and 3D, by setting `DrawBezierTracks=true` and specifying a `BezierTrackModuleLabel`. In both 2D and 3D the track is drawn as a smooth curve through 100 trajectory points evenly spaced along the length of the track. Bezier vertices are drawn as yellow dots at the 3D vertex point in the 3D event display. A sample bezier tracked event is shown in figure 7.

4.4 Bezier Tracking Performance

This section will be filled in following MCC 2.2

4.5 Technical Description of the BezierTrackerAlgorithm

The BezierTrackerModule runs the BezierTrackerAlgorithm in several stages. First, clusters are read in from the event and hits are sorted into an appropriate data structure for seed finding and bezier tracking.

The main BezierTracking method, `MakeTracks`, is called with this data structure passed by reference as an input parameter. This method performs seed finding and then bezier tracking.

Imperfect clustering upstream of bezier tracking introduces some artifacts into the Bezier Track collection which are removed by the next three stages. First, `FilterOverlapTracks` remove short track segments which

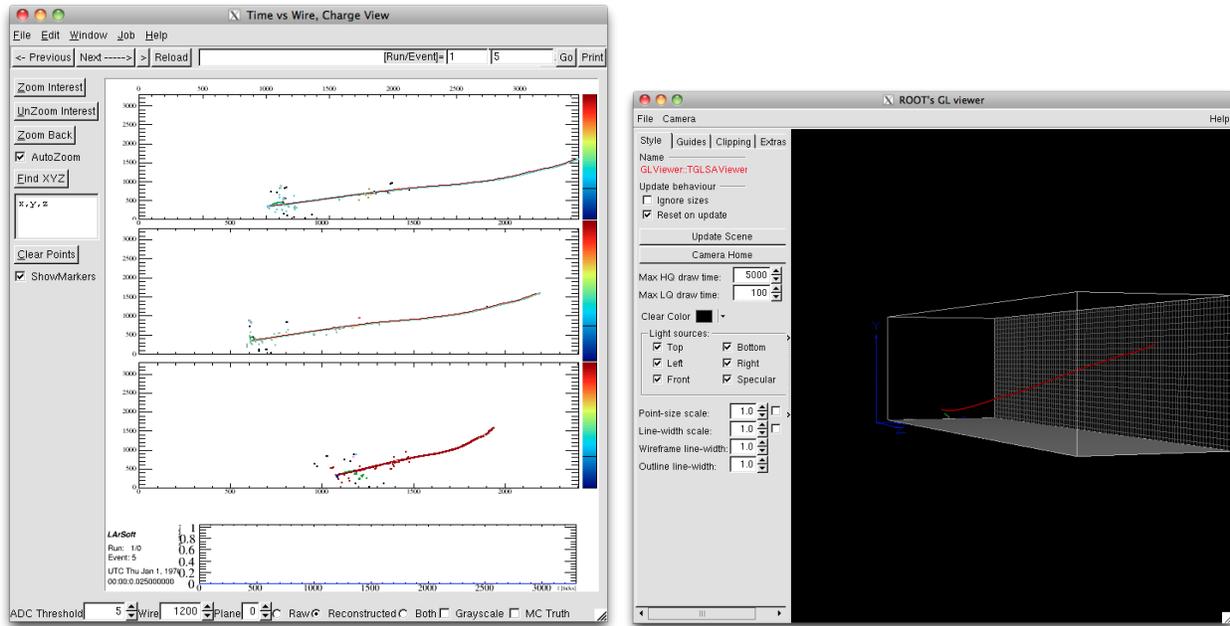


Figure 7: Example of a Bezier Tracked event drawn in the LArSoft event display

Parameter	Type	Default	Description
TrackJoinAngle	double	0.5	Max angle for seed to join track (rad)
OccupancyThresh	double	0.8	Fraction of channels which must have activity for extrap to be valid
OverlapCut	double	0.2	Maximum allowed overlap of hits in seed with used hits
DirectJoinDistance	double	5.0	Max distance between track segments to allow a join (cm)
VertexImpactThreshold	double	2.0	Max impact param between tracks for vertexing (cm)
VertexExtrapDistance	double	5.0	Max distance allowed for track extrapolation to a vertex (cm)
TrackResolution	double	2.0	Distance from track spine where hits can be collected
SeedFinder	pset	-	Parameterset for the SeedFinderAlgorithm

Table 5: List of the parameters of the BezierTrackerAlgorithm, with their default values and an approximate description of each

have been reconstructed directly on top of other, longer tracks. These segments arise when the ClusterFinder does not include a short section of the activity in one plane into the track, for some reason. Then, two Joiner functions, MakeOverlapJoins and MakeDirectJoins attempt to connect collinear tracks which were constructed in several parts. Next, Bezier vertexing (MakeVertexJoins) is performed. Finally the collection of tracks, vertices, calorimetry objects and relevant associations are stored into the event.

4.5.1 Parameters of the BezierTrackerAlgorithm

Table 5 gives the configurable parameters of the BezierTrackerAlgorithm and their default values, as well as an approximate description of each. Included in the parameter list is a parameterset for the SeedFinderAlgorithm. This should be similar to the default parameterset described in section 4.5.1. Better results have been obtained by removing very short seeds, by setting the LengthCut parameter of the SeedFinderAlgorithm to 2 cm.

4.5.2 Producing the organized hit collection

The `MakeTracks` method accepts a sorted hit collection and returns a vector of bezier tracks and, by reference, a vector of `PtrVectors` to hits which associate to each of those tracks.

```
std::vector<trkf::BezierTrack> MakeTracks(  
    std::map<geo::View_t,  
    std::vector<art::PtrVector<recob::Hit> > >& SortedHits,  
    std::vector<art::PtrVector<recob::Hit> >& HitAssocs)
```

The `SortedHits` collection provided to `BezierTracker` has the same structure as the collection provided to the `SeedFinderAlgorithm::GetSeedsFromSortedHits` method, as was described in section 3.1.2. The key to the highest level map in `SortedHits` is a `geo::View_t`, which for `MicroBooNE` can be either `geo::kU`, `geo::kV` or `geo::kW`. This splits the collection at the highest level into hits from each view. The next level of organization is a vector of `PtrVectors`, with each element representing one cluster in the given view. Finally, each cluster contains many hits, which are stored as the elements of the lowest level `PtrVector`. The method `GetHitsFromClusters` of `BezierTrackerModule` is responsible for retrieving clusters from the event and forming the structured `SortedHits` object.

4.5.3 Cluster Overlap Checking and Hit Filtering

One of the most time computationally intensive steps in the Bezier Tracking process is the formation of spacepoints during seed finding. Completely non-overlapping clusters, and extraneous hits included in one cluster with no overlap in another view can greatly increase the spacepoint finding time. As such, a significant speed increase (on the order of a factor of 10) was obtained by pre-filtering the clusters and hits to remove a) non-overlapping cluster combinations, and b) hits outside of the overlap region.

During an initial loop through all sets of hits in each view, the minimum and maximum time and wire in each cluster is extracted. The times are corrected for the plane-to-plane time-offsets which are obtained from the `DetectorProperties` service, to ensure that these times are comparable between planes. In the subsequent steps, we only consider cluster combinations which have some overlap in time - that is, if a cluster in view A exists completely outside of the time region of a cluster B, there can be no overlap and there is no need to attempt to find spacepoints. This is illustrated in figure 8. We conservatively also add a small buffer around the region of interest corresponding to the spacepoint algorithm timing resolution. We can obtain a similar constraint in the wire coordinate. The crossing point of the highest U and V wires, and the crossing point of the lowest U and V wires, determines the upper and lower limit in W where spacepoints can be found. If the cluster in the W view has no hits within this region, this cluster combination has no overlap, and processing does not continue.

Where cluster overlap is found, only the hits which exist within the 3-view overlap time window, determined by the time coordinates of the most constraining views and corrected for the plane-to-plane time offsets, need be used for seed finding (since only these hits will produce spacepoints). Hence the hit collection which is passed to the `SeedFinder` is produced only from hits within the time overlap region from these three clusters. The wire overlap region condition could also be used, but in practice for most tracks the majority of extraneous hits are already removed by the time constraint. This can be understood as the principal that, if the track extends beyond the region of interest in a particular view, it likely leaves the region both in the time and wire coordinates, since most tracks are either overall upward-going or overall downward-going rather than horizontal. In fact, the additional checking of wire coordinate as well as time coordinate was found to slightly detriment rather than improve the performance of the algorithm over large samples, due to the additional per-hit calculation required, so it is not included in the present version.

4.5.4 Making Bezier Tracks

The Bezier tracking routine is run for every overlapping combination of one U cluster, one V cluster and one W cluster. The first stage is running the `SeedFinderAlgorithm` on the sorted hit collection. This returns a vector of seed vectors, one for each combination, some of which may be empty.

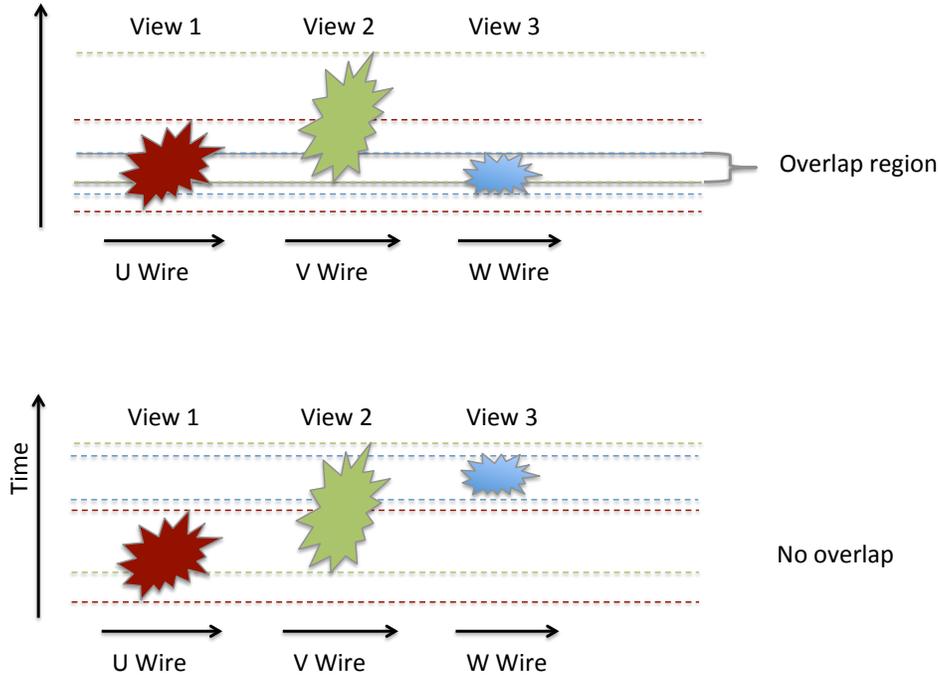


Figure 8: Illustration of time-overlap condition for cluster and hit filtering. A similar condition also exists in wire-space (although it is less easily drawn)

Before the loop over cluster combinations begins, for each cluster in each view, a lookup table is produced which indexes the hits in each cluster by channel and view, similar in structure to the `OrgHits` table used in the `SeedFinderAlgorithm`, which was described in 3.5.4. By passing pointers into this collection to the track finding method, we avoid re-making the lookup table each time the same cluster is used. The `OrgHits` tables used by `SeedFinder` and `BezierTracker` do not refer to similarly structured objects so separate tables are maintained for each. There is room for further optimization here, but major code restructuring may be required to bring both hit tables into the same format.

We then loop over all combinations for which at least one seed was found, and call the `OrganizeSeedsIntoTracks` method for each.

```
std::vector<std::vector< recob::Seed > > OrganizeSeedsIntoTracks(
    std::vector<recob::Seed > & AllSeeds,
    std::vector<art::PtrVector<recob::Hit>*> & AllHits,
    std::vector<art::PtrVector<recob::Hit> > & WhichHitsPerSeed,
    std::vector<std::vector< std::vector<int> >*> & OrgHits,
    std::vector<std::vector<std::vector<int> > > & HitsWithTracks )
```

This method returns a vector of seed vectors, one for each proposed track. The collection of hits which are associated with each proposed track are returned in the `HitsWithTracks` structure. This three level vector is indexed first by the track ID (the index of the entry in the returned vector of seed vectors), then by view, and at the lowest level is a vector of integers, each of which indicates a position of a hit in the supplied `AllHits` `PtrVector`. To minimize repeated computational work we also supply the method with the list of hits which are associated to each seed as produced by `SeedFinder`, `WhichHitsPerSeed`, and pointers to the hit lookup table for each view, compiled into a single vector with 3 entries, `OrgHits`.

Within the `OrganizeSeedsIntoTracks` method, the overall approach will be to loop until we have exhausted the seed collection, starting with the seed which is at one of end of the remaining collection and attempting

to chain other seeds which can make a consistent bezier track end to end. After each seed is added, both the hits associated with that seed and those used in the interpolation from the previous seed are incorporated into the track hit collection. After a seed is added to the track, each remaining seed is checked to see if it has a significant fraction (defined by the `OverlapCut` parameter) of its hits already incorporated into a track. Any seeds which meet this condition will be removed from consideration for tracking. After a track is found, all of its hits become unavailable for future interpolations.

The requirements for a seed to be joined onto a growing track are (refer to table 2 for seed geometry definitions):

- `SeedAngle` between this seed and the last seed must be less than the `TrackJoinAngle` parameter
- `ProjAngleDiscrepancy` between this seed and the last seed must be less than the `TrackJoinAngle` parameter
- `PointingSign` must be consistent between the last three seeds
- The occupancy of the extrapolation (see below) between the seeds must be larger than the `OccupancyThresh` parameter in each view

If all these conditions are satisfied, the seed is added to the track and will be removed from future consideration. If one or more is not satisfied, the seed is temporarily set aside but will be considered for the next track.

The extrapolation condition is evaluated by producing a bezier curve between the two seeds and checking every channel between the last occupied channel of seed 1 and the first occupied channel of seed 2 in each view to see whether a hit exists within a distance defined by the `TrackResolution` parameter. The fraction of channels which have activity within the track resolution of the extrapolated curve must be larger than the `OccupancyThresh` parameter. If the last channel of seed 1 and the first channel of seed 2 are adjacent, then the condition is automatically satisfied. First channel and last channel in this case refer to the channels at the closest ends of the seed pair. This must be established consistently in all views, and as such at the beginning of the run each seed has its first and last channels in each view identified and tabulated in the `SeedHighSChans` and `SeedLowSChans` vectors in each view. The pointing direction of the seed match dictates whether the first and last channels are sampled from the high S or low S end of each seed. After seed finding is complete, any seeds which are running from high to low S in the track are flipped, so that in the final bezier tracks the direction vector of each seed points in the same direction along the track.

After tracking is complete, tracks are sorted by length, and flipped such that all tracks point forward in the z direction, as a convention, by the `SortTracksByLength` method.

4.5.5 Overlap Filtering and Track Joining

There are three methods for removing artifacts of imperfect clustering, and tracks which have been produced as broken sections due to extraneous activity (eg delta rays) interfering with seed finding. The purpose of each function is shown graphically in table 6.

The `FilterOverlapTracks` method takes a long track A, and searches for a shorter track B which has both ends within `TrackResolution` of the A trajectory, and with both ends of the B having their closest approach to A in the range $0 < S_A < 1$. If such an overlap is found, track B discarded.

The `MakeOverlapJoins` method seeks to connect tracks which run directly into each other, as can happen when wide hits are improperly reconstructed giving two almost parallel track trajectories within a cluster. The condition for connecting the two tracks is that end $S_A = 1$ of track A must be within `TrackResolution` of track B in the range $0 < S_B < 1$, and end $S_B = 0$ of track B must be within `TrackResolution` of track A in the range $0 < S_A < 1$. If such a pair of tracks is found, they are combined into a single track, with the trajectory of A in the range $\{0, S_A\}$ followed by the trajectory of B in the range $\{S_B, 1\}$.

The `MakeDirectJoins` method connects tracks which can be extended over short untracked regions. The end directions must match to within `TrackJoinAngle`, and there is a distance cut of `DirectJoinDistance` applied. The ends to be joined must be the closest pair of ends between the two tracks and be collinear

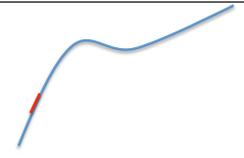
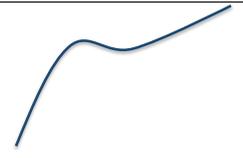
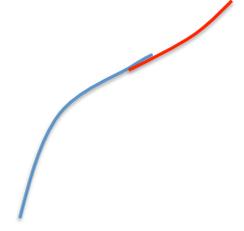
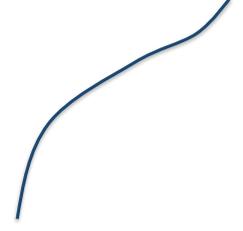
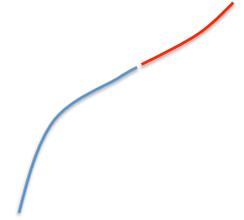
Method	Before	After
FilterOverlapTracks		
MakeOverlapJoins		
MakeDirectJoins		

Table 6: Purpose of Overlap Filtering and Track Joining Methods

to within `TrackResolution`. If all these conditions are satisfied the two tracks may be connected into one longer track.

4.5.6 Bezier Vertexing

After all tracks have been found, the `BezierTrackerModule` looks for 3D vertices which can be formed by extrapolating each track. All pairs of tracks are extrapolated to find their impact parameters with respect to one another from both ends. If any pair can be extrapolated less than `VertexExtrapDistance` and meet with an impact parameter of less than `VertexImpactThreshold` then a vertex is placed at this 3D location. Once a vertex is found, every other track is checked to see if it could originate from the same vertex. The final vertex position is the average of the position given by every pair of vertexed tracks.

4.5.7 Bezier Calorimetry

Once the track trajectory has been established and the hits along the track have been identified, it is a relatively simple operation to perform pitch-corrected calorimetry. The `EDProducer Calorimetry/BezierCalorimetry` loops through all tracks in the events and accesses their associated hits. These hits are processed via the `BezierTrack::GetCalorimetryObject` method, which finds the S position of each hit along the track, calculates the local track pitch and applies the appropriate pitch correction. This method has been exploited in previous MicroBooNE handscan exercises and its performance will be characterized in MCC 2.2.

5 Conclusions

This note has described the Seed Finding and Bezier Tracking algorithms. Seed finding is an effective way of finding unambiguous sections in complicated events, and is likely to be a part of the 3D reconstruction within LArSoft as a precursor to 3D tracking algorithms such as `Track3DKalmanHit` or `BezierTracker`. Bezier tracking is one method of establishing the trajectories of long, curved tracks, and it appears to be effective for

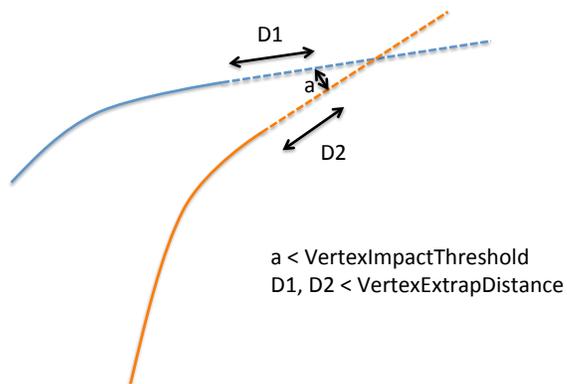


Figure 9: Bezier vertexing conditions

simple topologies such as cosmic ray muon tracks. It is likely that more complicated events will require more advanced algorithms, and in particular, Seed Finding and Bezier Tracking are unlikely to be the appropriate reconstruction technology for very low energy tracks, or complicated events such as electromagnetic showers.

Acknowledgements

Thanks to Herb Greenlee and Sowjanya Gollapini for the development of diagnostic tools to understand the performance of these algorithms, and to the whole LArSoft team for helping to develop the ideas and codebase upon which these algorithms are built.