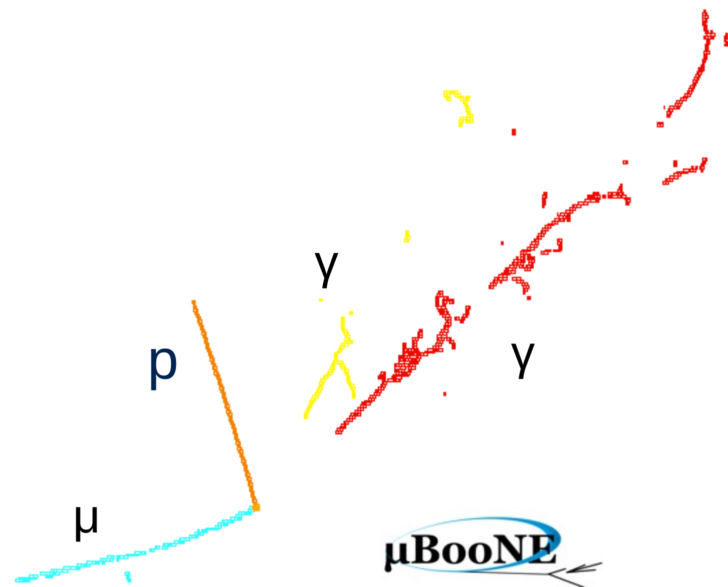


Pandora Talk 6: Vertex Reconstruction

J. de Vries for the Pandora Team

MicroBooNE Pandora Workshop

July 11-14th 2016, Cambridge

Pandora Vertex Reconstruction

- The Pandora vertex reconstruction process, as per Pandora's design philosophy, consists of multiple algorithms with clearly separated goals
- Despite this, the vertex reconstruction is less stratified than most of Pandora, consisting of two main parts:
  - The vertex candidate creation algorithm
  - The vertex selection algorithm
- The first algorithm creates multiple vertex candidates and the second algorithm aims to select the most likely representation of the neutrino interaction vertex
- This presentation will discuss both algorithms in detail, to give you a clear overview of the vertex reconstruction procedure inside Pandora
- This procedure is currently detector agnostic, and hence applies to MicroBooNE, DUNE, ProtoDUNE, etc.

Vertex Candidate Creation

μBooNE

- The main idea behind the vertex candidate creation algorithm is to identify all the main feature points of the event in 3D, so that the true neutrino interaction vertex should almost always be included
- The algorithm considers all the reconstructed 2D clusters and tries to match the positions of their 2D endpoints between views
- Using 2D cluster endpoint positions from two views allows one to predict the most likely endpoint position in 3D
- We then place a 3D vertex candidate there for later analysis

Next: illustrate this process step by step

Vertex Candidates

µBooNE

- This is what all the vertex candidates look like in a typical event (indicated in blue in the figure on the right)
- The event in the figure is a CC quasi-elastic ν_μ event with one muon and one proton
- As you can see, there is a large redundancy in the vertex candidates
- As mentioned before: this leads us to assume that, in the vertex reconstruction, the true neutrino vertex is present as a candidate at all times!

CCQEL_MU_P

W

μ

Vertex candidates: blue

U

p

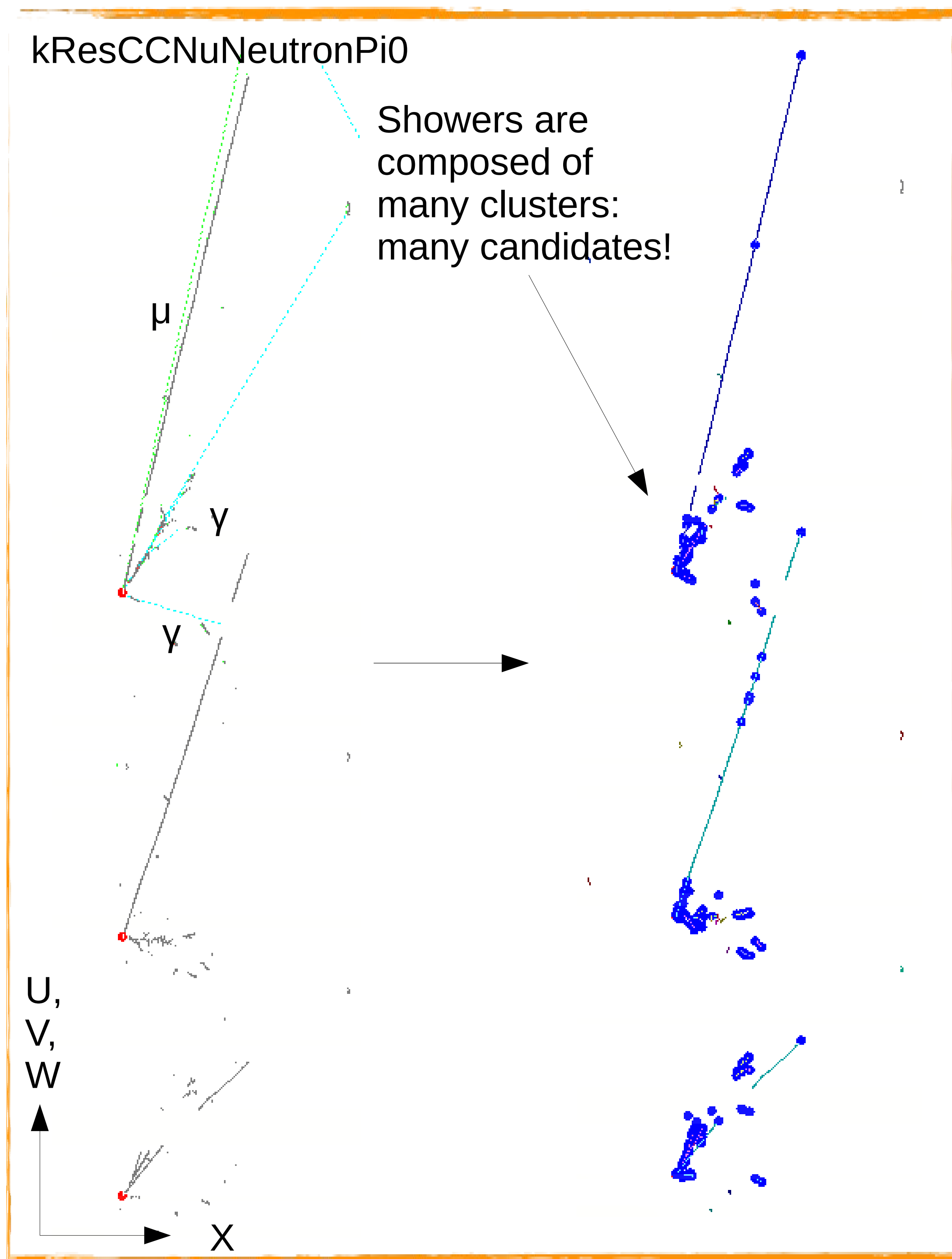
V

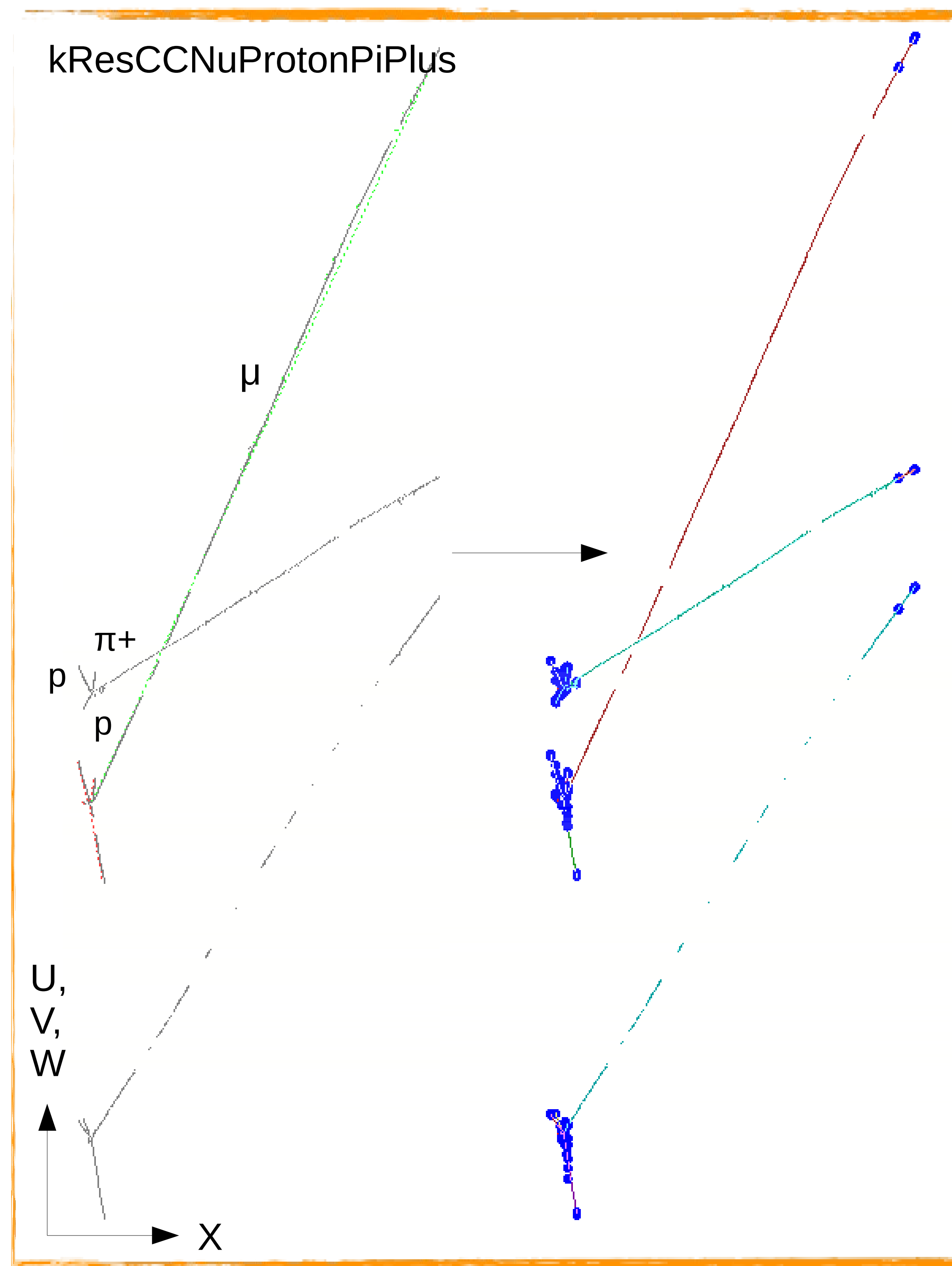
U, V, W

True vertex projection: red

X

Vertex Candidate Creation





2D Sliding Linear Fits

- I have mentioned that we want to look at the cluster endpoints
- 2D cluster endpoint positions are typically extracted by performing a 2D sliding linear fit on each cluster in the event
- A 2D sliding linear fit is an internal Pandora object that can track along ‘bendy’ objects and consists of a fixed number of segments
- The 2D sliding fit object has a lot of built-in machinery, such as producing the positions of the first and last segment in the fit
- Works just as well for 'transverse' clusters as for 'longitudinal' clusters

Cache the 2D sliding linear fits for all the clusters so we only have to create them once (for efficiency)

```
const float slidingFitPitch(LArGeometryHelper::GetWireZPitch(this->GetPandora()));
const TwoDSlidingFitResult slidingFitResult(pCluster, m_slidingFitWindow, slidingFitPitch);
```

layers for sliding fit: parameter configurable through XML settings file

Then we retrieve the relevant sliding fit from cache and request the minimum and maximum layer positions (in the relevant view)

```
const TwoDSlidingFitResult &fitResult1(this->GetCachedSlidingFitResult(pCluster1));
const CartesianVector minLayerPosition1(fitResult1.GetGlobalMinLayerPosition());
const CartesianVector maxLayerPosition1(fitResult1.GetGlobalMaxLayerPosition());
```

Pandora makes this very easy!

Permutations, permutations…

- Now the positions of the endpoints of the clusters as well as the 2D sliding linear fit objects themselves are available
- Next, compare the cluster endpoint positions for every permutation of the U, V and W views
- Do this by taking the endpoints of the 2D sliding fit object in one view, together with the sliding fit object in a second view
- Do this for every endpoint and every sliding fit: we compare all permutations of cluster endpoints
- If a set of criteria are met (explained on the next slide) then create a 3D vertex at this position
- Many possible permutations: 4 candidates for every pair of 2D clusters. This is why there are so many vertex candidates!

The method works by comparing cluster endpoints for all three permutations of the three views

```
this->ClusterEndPointComparison(clusterListU, clusterListV);
this->ClusterEndPointComparison(clusterListU, clusterListW);
this->ClusterEndPointComparison(clusterListV, clusterListW);
```

U, V
U, W
V, W

The way this is done is by comparing all possible permutations of the 2D sliding fit endpoints

```
this->CreateVertex(maxLayerPosition1, hitType1, fitResult2);
this->CreateVertex(minLayerPosition1, hitType1, fitResult2);
this->CreateVertex(maxLayerPosition2, hitType2, fitResult1);
this->CreateVertex(minLayerPosition2, hitType2, fitResult1);
```

Cluster Endpoint Projection Example

µBooNE

W view

CCQEL_MU_P

Cluster endpoint projection indicated by dotted line

µ

Vertex candidates in yellow

p

U view

V view

True vertex projection in red

U,V, W

X

Vertex Creation Logic

- Now compare a cluster endpoint in one view to a 2D sliding fit in a second view
- The 'create vertex' logic checks whether the match makes sense by demanding that:
  - The endpoint X position projects onto the sliding fit length (X span) in the second view
- If it doesn't, then it must be within some minimum distance from the cluster in X
- If two suitable cluster endpoints are found, use the in-built Pandora functionality to project this position into 3D
- If this projected 3D position has an acceptable level of uncertainty, we create a vertex candidate there

Check whether the endpoint projects onto the 2D sliding fit, or lies within a minimum distance

```
if ((((position1.GetX() < minLayerPosition2.GetX()) && (position1.GetX() < maxLayerPosition2.GetX())) ||
    ((position1.GetX() > minLayerPosition2.GetX()) && (position1.GetX() > maxLayerPosition2.GetX()))) &&
    (std::fabs(position1.GetX() - minLayerPosition2.GetX()) > m_maxClusterXDiscrepancy) &&
    (std::fabs(position1.GetX() - maxLayerPosition2.GetX()) > m_maxClusterXDiscrepancy))
{
    return;
}
```

Project the two positions into 3D

```
CartesianVector position3D(0.f, 0.f, 0.f);
LArGeometryHelper::MergeTwoPositions3D(this->GetPandora(), hitType1, hitType2, position1, position2, position3D, chiSquared);
```

Finally create the vertex using the standard Pandora object creation pattern

```
PandoraContentApi::Vertex::Parameters parameters;
parameters.m_position = position3D;
parameters.m_vertexLabel = VERTEX_INTERACTION;
parameters.m_vertexType = VERTEX_3D;

const Vertex *pVertex(NULL);
PANDORA_THROW_RESULT_IF(STATUS_CODE_SUCCESS, !=, PandoraContentApi::Vertex::Create(*this, parameters, pVertex));
```

Vertex Selection Algorithm

- Many vertex candidates have now been created
- The task that remains is selecting the vertex candidate that is most likely to be the true neutrino vertex
- This is the far harder part of the vertex reconstruction because of the great redundancy in the candidates, as well as the existence of many complicated topologies in the various interaction channels
- The main approach of the vertex selection algorithm is to give each vertex candidate a score, indicating how ‘good’ it is

Scoring Vertex Candidates

µBooNE

- There are multiple different ways of scoring the vertex candidates
- **Topology score**: this score essentially tries to document how many clusters are leaving the vertex point by plotting the neighbouring hit positions in R and phi
  - The main idea behind the topology score is that a good vertex candidate will have multiple straight tracks emerging from it: this means that in a binned R and phi distribution, many hits will fall into the same bin
- **Beam score**: this score tries to take into account how far down the beamline the vertex candidate is, as it is more likely that the true neutrino vertex will be upstream
  - This beam score can be switched on and off via the XML settings file
- **Other scores**: we are constantly looking for ways in which to improve the vertex reconstruction, as it has large impact on the quality of the subsequent reconstruction
  - An example is such a score could be be a score that implements calorimetry information when it is available

Vertex Filtering Procedure

- Initial vertex filtering stage, where one checks whether the vertex projection lies acceptably close to a hit
- This filtering is just to make the reconstruction faster by removing bad candidates at an early stage
- This filtering treatment seeks to disregard obviously infeasible candidates, but also allows for a view to be empty or a vertex to lie within a gap:

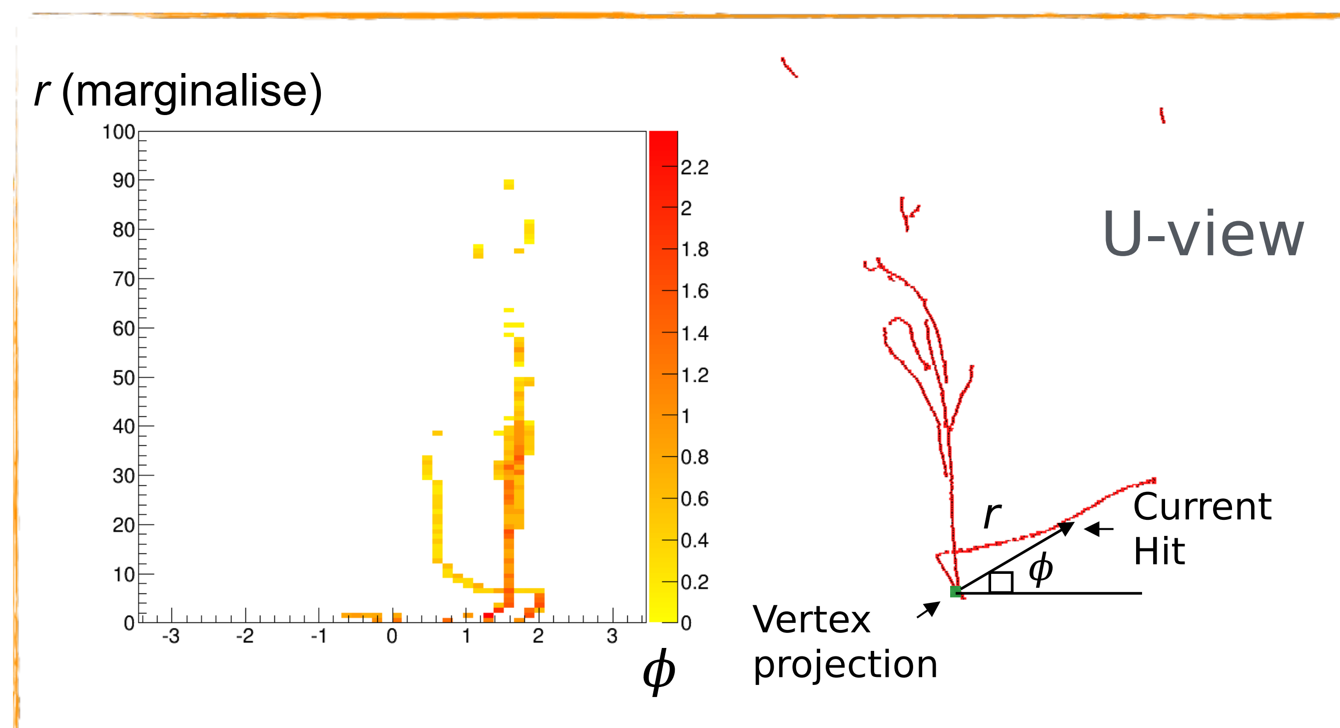
The vertex filtering procedure allows 2 view only reconstruction:

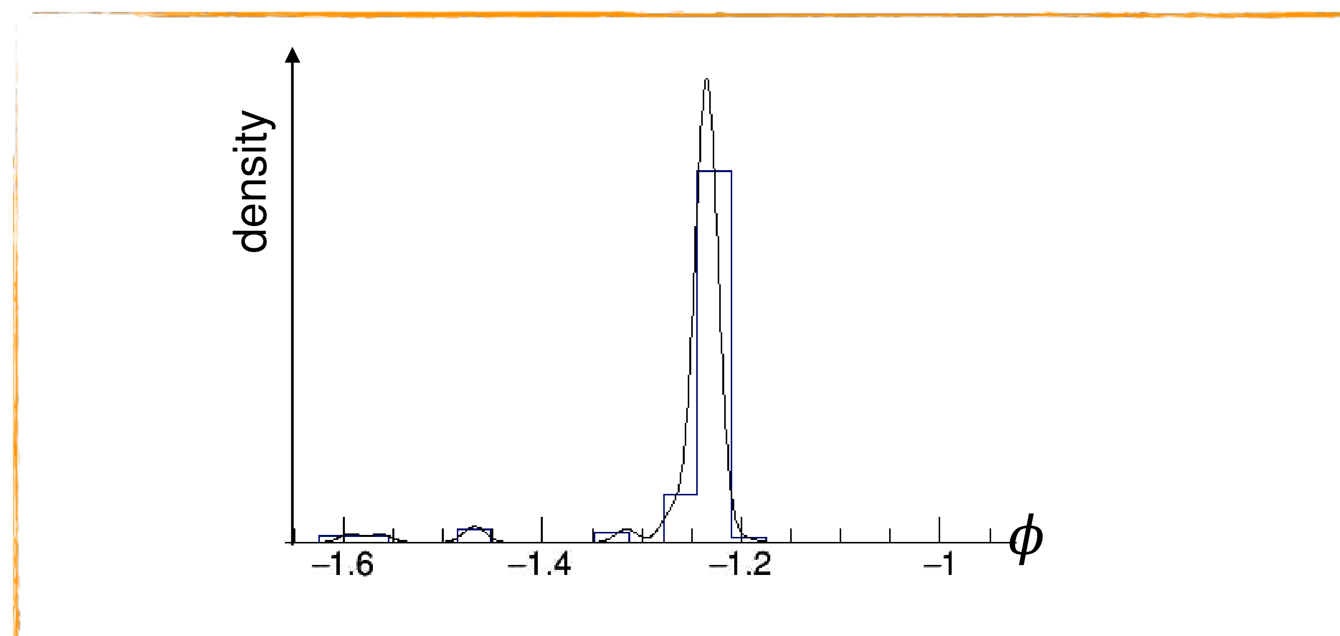
Empty view? Vertex on hit? Vertex in gap?

```
if ((m_isEmptyViewAcceptable && kdTreeU.empty()) || this->IsVertexOnHit(pVertex, TPC_VIEW_U, kdTreeU) || this->IsVertexInGap(pVertex, TPC_VIEW_U))
    ++nAcceptableViews;

if ((m_isEmptyViewAcceptable && kdTreeV.empty()) || this->IsVertexOnHit(pVertex, TPC_VIEW_V, kdTreeV) || this->IsVertexInGap(pVertex, TPC_VIEW_V))
    ++nAcceptableViews;

if ((m_isEmptyViewAcceptable && kdTreeW.empty()) || this->IsVertexOnHit(pVertex, TPC_VIEW_W, kdTreeW) || this->IsVertexInGap(pVertex, TPC_VIEW_W))
    ++nAcceptableViews;
```

Vertex Topology Scoring

- The algorithm will now assign a score to each vertex candidate in the filtered vertex list. This process proceeds in three steps:
  - 1D Histogram (R, phi) approach
  - Kernel density estimation
  - Folding method
- For every vertex candidate, the nearby hits (in 2D) are plotted as a function of (R, ϕ): see figure
- Every hit gets a weight and are deweighted by their distance from the vertex candidate
- The hit ϕ distribution is binned, and the bin entries are squared and summed to give a score
- The idea is that for a good vertex candidate, many straight clusters will leave that point, and we will see many hits in a narrow range of (R, ϕ): a high peak



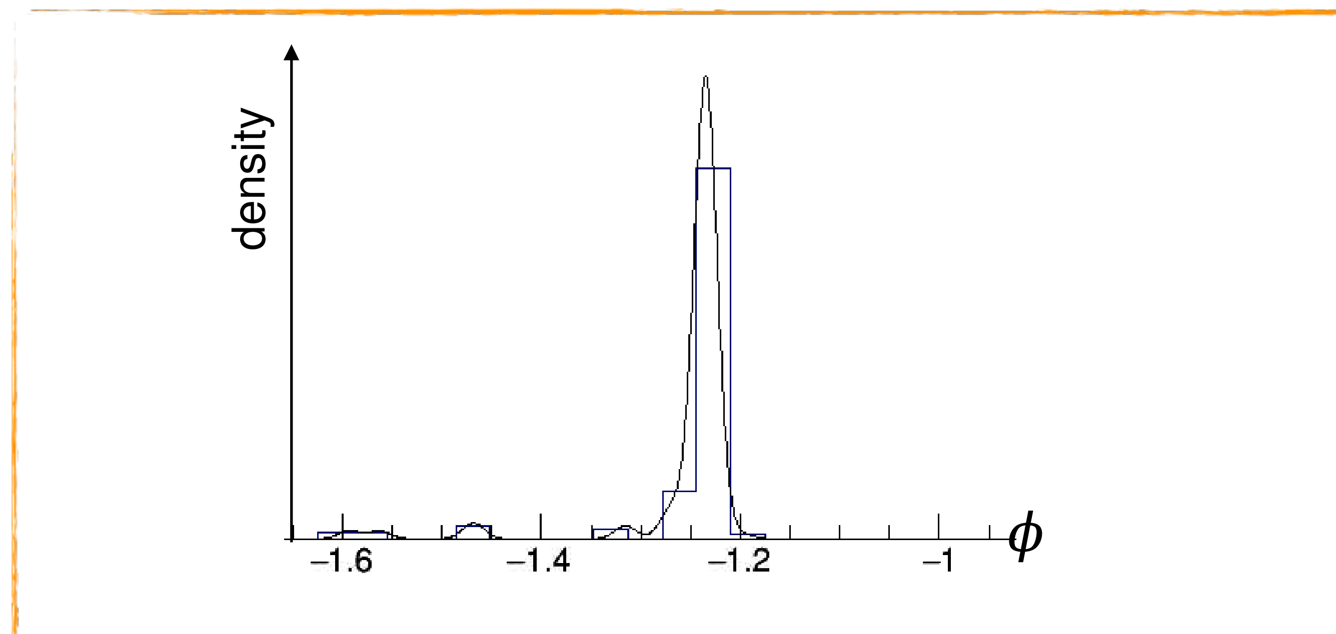


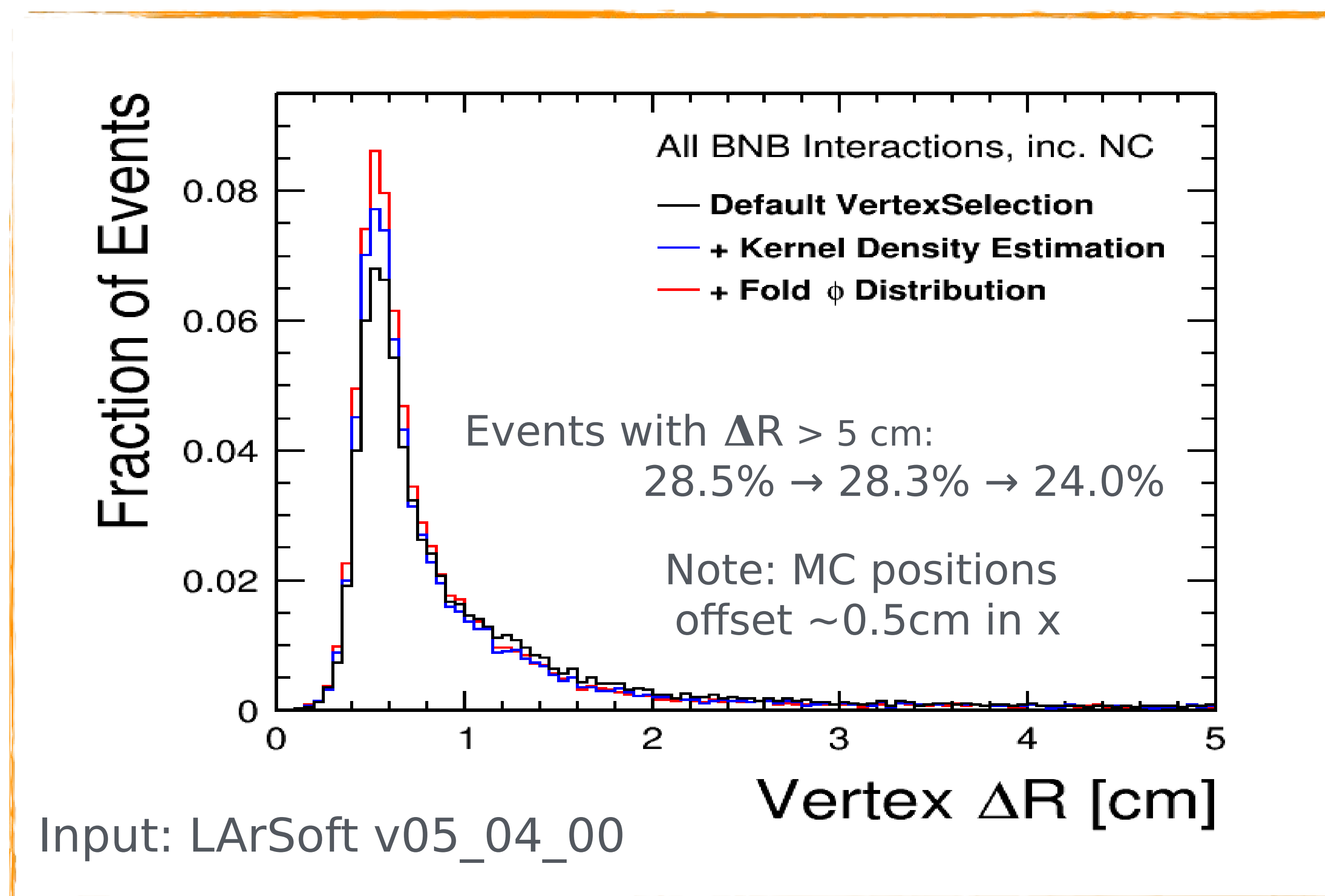
$$Score = \sum_{i}^{n} w_i f(R)$$

Kernel Density Estimation & “Folding”

µBooNE

- Kernel density estimation provides a non-parametric estimation of the hit ϕ distribution (r-deweighted)
- Use KDE to get rid of binning effects
- Failure mode: because of the way the candidates are generated, one often gets candidates in the middle of straight tracks as can be seen in previous event displays
- We deweight such otherwise appealing candidates by deweighting the positive ϕ range with the negative positive ϕ range
- Give the negative ϕ region and negative weight and 'fold' it over to the positive ϕ region: this deweights hits that are offset by 2 pi





Beam Deweighting & Final Vertex Selection

µBooNE

- Since the previous scoring approach is determined purely by the topology, we often refer to it as the 'topology score'
- The second element of the scoring procedure is to modify the score value based on how far down the beamline the vertex candidate is
- As was mentioned: it is simply more likely to find the true neutrino interaction vertex upstream, rather than down the beamline. This score modifier attempts to account for that fact.
- The beam deweighting is performed using 3D information
- First, we determine what the 'span' of the event is in Z using the vertex candidates that made it through the filtering step mentioned previously:

$$z_{span} = |z_{min} - z_{max}|$$

- The beam deweighting factor is then defined to be: $e^{-\lambda z}$
- Where the first argument can be modified via the Pandora xml settings file

$$\lambda \equiv nDecayLenghtsInZSpan / ZSpan$$

- Now that a score has been assigned to every vertex candidate, the neutrino interaction vertex is assumed to be the vertex candidate with the highest score
- The neutrino vertex is then used in the following stages of the reconstruction

Thank you for your attention!

Any Questions?