Proposal for a new recob::Vertex

G. Cerati

LArSoft Coordination Meeting

Dec. 5, 2017

Introduction

- Current recob::Vertex not suitable to capture vertex fit information
  - just an array of doubles for the position, and an integer for an ID
  - no room for covariance matrix, chi2, etc.
- This is an update of the presentation given at the end of September:
  - see presentation and minutes at: https://indico.fnal.gov/event/15361/
  - diagrams of the algorithm also in backup
- Got useful feedback, now it’s time for a concrete proposal

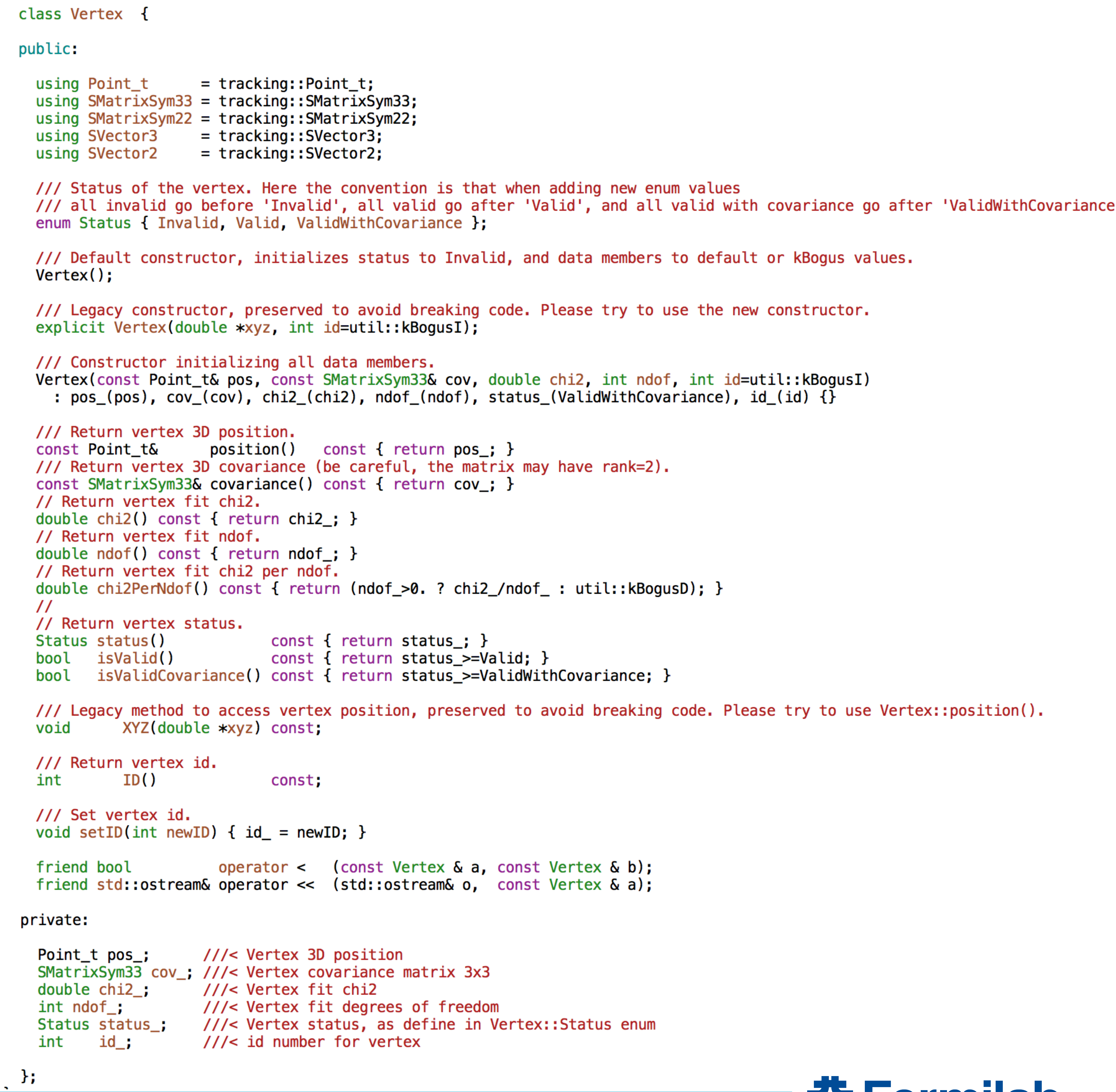
Overall strategy

- Extend recob::Vertex to store information about the vertex itself
  - only extension of the interface, ensure backward compatibility (at least for now)
  - update private data members: need schema evolution rules

- Capture information about tracks via art Assn including meta data
  - tracks used in the vertex fit, or anyway associated to the vertex
  - plain association (no meta data) was already possible
  - meta data contains information about the track-vertex association

New recob::Vertex object

- Keep legacy constructor
- Keep int for ID
- Move to Point_t for position
  - ROOT::Math::PositionVector3D<ROOT::Math::Cartesian3D<Coord_t>>
- Add covariance matrix
- Add chi2 and ndof
- Add status enum
  - Invalid, Valid, ValidWithCovariance
  - can be extended following conventions
- Add new constructor initializing all data members
- Add various getters



```
class Vertex  {

public:

  using Point_t      = tracking::Point_t;
  using SMatrixSym33 = tracking::SMatrixSym33;
  using SMatrixSym22 = tracking::SMatrixSym22;
  using SVector3     = tracking::SVector3;
  using SVector2     = tracking::SVector2;

  /// Status of the vertex. Here the convention is that when adding new enum values
  /// all invalid go before 'Invalid', all valid go after 'Valid', and all valid with covariance go after 'ValidWithCovariance
  enum Status { Invalid, Valid, ValidWithCovariance };

  /// Default constructor, initializes status to Invalid, and data members to default or kBogus values.
  Vertex();

  /// Legacy constructor, preserved to avoid breaking code. Please try to use the new constructor.
  explicit Vertex(double *xyz, int id=util::kBogusI);

  /// Constructor initializing all data members.
  Vertex(const Point_t& pos, const SMatrixSym33& cov, double chi2, int ndof, int id=util::kBogusI)
    : pos_(pos), cov_(cov), chi2_(chi2), ndof_(ndof), status_(ValidWithCovariance), id_(id) {}

  /// Return vertex 3D position.
  const Point_t&      position()   const { return pos_; }
  /// Return vertex 3D covariance (be careful, the matrix may have rank=2).
  const SMatrixSym33& covariance() const { return cov_; }
  // Return vertex fit chi2.
  double chi2() const { return chi2_; }
  // Return vertex fit ndof.
  double ndof() const { return ndof_; }
  // Return vertex fit chi2 per ndof.
  double chi2PerNdof() const { return (ndof_>0. ? chi2_/ndof_ : util::kBogusD); }
  //
  // Return vertex status.
  Status status()             const { return status_; }
  bool   isValid()            const { return status_>=Valid; }
  bool   isValidCovariance()  const { return status_>=ValidWithCovariance; }

  /// Legacy method to access vertex position, preserved to avoid breaking code. Please try to use Vertex::position().
  void      XYZ(double *xyz) const;

  /// Return vertex id.
  int       ID()             const;

  /// Set vertex id.
  void setID(int newID) { id_ = newID; }

  friend bool          operator <   (const Vertex & a, const Vertex & b);
  friend std::ostream& operator <<  (std::ostream& o,   const Vertex & a);

private:

  Point_t pos_;      ///< Vertex 3D position
  SMatrixSym33 cov_; ///< Vertex covariance matrix 3x3
  double chi2_;      ///< Vertex fit chi2
  int ndof_;         ///< Vertex fit degrees of freedom
  Status status_;    ///< Vertex status, as define in Vertex::Status enum
  int    id_;        ///< id number for vertex

};
```

Schema evolution rules

- Rather straightforward:
  - convert array of double into Point_t
  - update ID variable name
  - default constructor takes care of initializing the other variables to dummy values (util::kBogus or 0)

```
<!-- recob::Vertex: schema evolution rules -->
    <!-- version 13 -->
      <!-- * position -->
<ioread
  version="[-12]"
  sourceClass="recob::Vertex"
  source="double fXYZ[3];"
  targetClass="recob::Vertex"
  target="pos_"
  include="lardataobj/RecoBase/Vertex.h">
  <![CDATA[
       pos_ = recob::tracking::Point_t(onfile.fXYZ[0],onfile.fXYZ[1],onfile.fXYZ[2]);
  ]]>
</ioread>
      <!-- * id -->
] <ioread
  version="[-12]"
  sourceClass="recob::Vertex"
  source="int fID;"
  targetClass="recob::Vertex"
  target="id_"
  include="lardataobj/RecoBase/Vertex.h">
  <![CDATA[
       id_ = onfile.fID;
  ]]>
</ioread>
```

VertexAssnMeta

- Contains useful information about track-vertex association:
  - they are supposed to be computed wrt an ‘unbiased’ vertex, i.e. the vertex fitted without using that track
  - propagation distance from start point to closest approach to vertex
  - impact parameter (with error)
  - chi2
  - status enum
- In principle, nothing is specific to tracks and could be used for associating the vertex to other objects, e.g. showers

```
class VertexAssnMeta {
public:
  enum VertexAssnStatus { Undefined, NotUsedInFit, RejectedByFit, IncludedInFit };
  VertexAssnMeta() { status_ = Undefined; }
  VertexAssnMeta(float pD, float iP, float iPErr, float c2, VertexAssnStatus st)
    : propDist_(pD), impactParam_(iP), impactParamErr_(iPErr), chi2_(c2), status_(st)
  float propDist()          const { return propDist_; }
  float impactParam()       const { return impactParam_; }
  float impactParamErr()    const { return impactParamErr_; }
  float impactParamSig()    const { return impactParam_/impactParamErr_; }
  float chi2()              const { return chi2_; }
  VertexAssnStatus status() const { return status_; }
  void updateStatus(const VertexAssnStatus& newstatus) { status_ = newstatus; }
private:
  float propDist_;
  float impactParam_;
  float impactParamErr_;
  float chi2_;
  VertexAssnStatus status_;
};
```

Updates to GeometricVertexFitter

- 3D vertex fitter based on the geometric properties (start position, direction, covariance) of the tracks, described in previous talk
- Added methods to obtain the (unbiased) propagation distance, impact parameter, impact parameter error, impact parameter significance, and chi2 of a track with respect to the vertex.
- Inputs are: a set of tracks; interface is provided allowing these to be passed directly of through a PFParticle hierarchy.
- Outputs are: a VertexWrapper, containing the vertex and the reference to the tracks actually used in the fit; methods to produce recob::VertexAssnMeta are provided.
- Tracks are included if the significance of the impact parameter with respect to the vertex is < cut (cut=3. by fcl default)

VertexWrapper

- Wrapper class to facilitate vertex production.
- It stores the recob::Vertex being built and the references to the tracks being used in the vertex fit.
- Tracks are stored in a vector of std::reference_wrapper<const recob::Track>, so the wrapper does not own the pointer to the original track object.

```
// use reference_wrapper instead of pointers: we do not want ownership of the tracks
typedef std::vector<std::reference_wrapper<const recob::Track> > TrackRefVec;

class VertexWrapper {

public:

  VertexWrapper() { vtx_ = recob::Vertex(); }
  VertexWrapper(const recob::Vertex& vtx) : vtx_(vtx) {}
  VertexWrapper(const recob::tracking::Point_t& pos, const recob::tracking::SMatrixSym33& cov, double chi2, int ndof)
  //
  const recob::Vertex& vertex() const { return vtx_; }
  bool isValid() const {return vtx_.isValid();}
  const recob::tracking::Point_t& position() const { return vtx_.position(); }
  const recob::tracking::SMatrixSym33& covariance() const { return vtx_.covariance(); }
  //
  void setVertexId(int newID) { vtx_.setID(newID); }
  //
  void addTrack(const recob::Track& tk) { vtxtks_.push_back(tk); }
  void addTrackAndUpdateVertex(const recob::tracking::Point_t& pos, const recob::tracking::SMatrixSym33& cov,
                               double chi2, int ndof, const recob::Track& tk) {
    vtx_ = recob::Vertex(pos, cov, vtx_.chi2()+chi2, vtx_.ndof()+ndof);
    addTrack(tk);
  }
  //
  size_t findTrack(const recob::Track& tk) const {
    for (size_t it = 0; it!=vtxtks_.size(); ++it) {
      if (&tk==&vtxtks_[it].get()) return it;
    }
    return vtxtks_.size();
  }
  //
  size_t tracksSize() const { return vtxtks_.size(); }
  const TrackRefVec& tracks() const { return vtxtks_; }
  TrackRefVec tracksWithoutElement(size_t element) const {
    TrackRefVec tks = vtxtks_;
    tks.erase(tks.begin()+element);
    return tks;
  }

private:
  recob::Vertex vtx_;
  TrackRefVec vtxtks_;
};
```

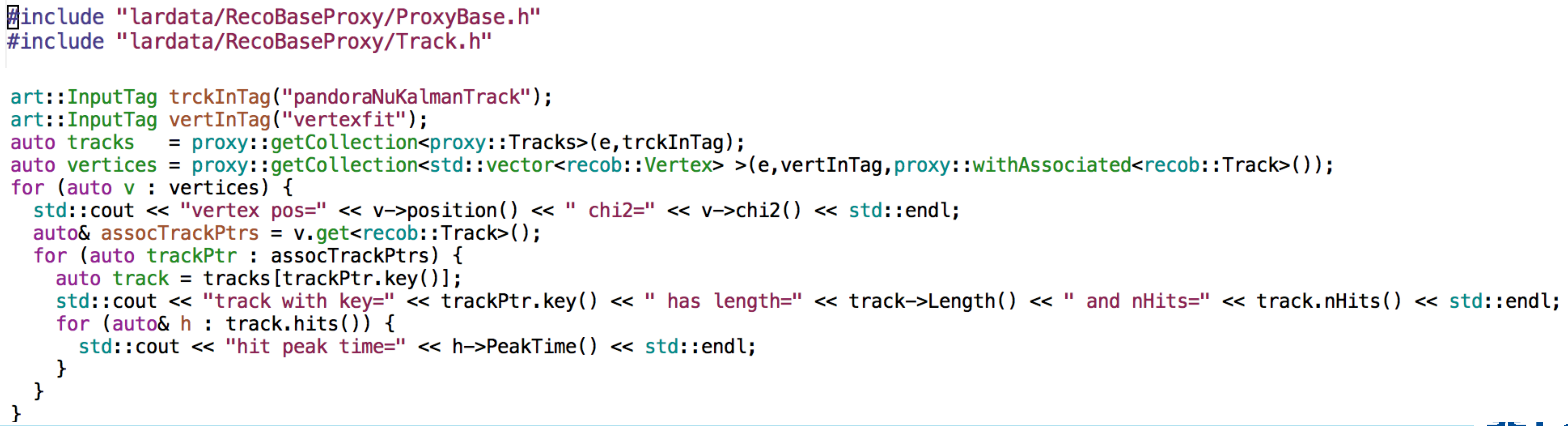
Fermilab

VertexFitter_module

- Currently taking as input tracks linked to PFParticles that are daughters of the neutrino PFParticle
- Produces:
  - std::vector<recob::Vertex>
  - art::Assns<recob::PFParticle, recob::Vertex>
  - art::Assns<recob::Vertex, recob::Track, recob::VertexAssnMeta>

Accessing Vertices and Tracks with proxy

- Spoiler: this is an advertisement!
- Proxy recently developed to make it easier to access associated data
  - see Gianluca’s talk: https://indico.fnal.gov/event/15455/contribution/3/material/slides/0.pdf
- Here is a simple example to access the tracks associated to the vertex, via the track proxy (which in turn gives easy access to e.g. hits)
  - access to meta data not supported yet, hopefully coming soon



```
#include "lardata/RecoBaseProxy/ProxyBase.h"
#include "lardata/RecoBaseProxy/Track.h"

art::InputTag trckInTag("pandoraNuKalmanTrack");
art::InputTag vertInTag("vertexfit");
auto tracks   = proxy::getCollection<proxy::Tracks>(e,trckInTag);
auto vertices = proxy::getCollection<std::vector<recob::Vertex> >(e,vertInTag,proxy::withAssociated<recob::Track>());
for (auto v : vertices) {
  std::cout << "vertex pos=" << v->position() << " chi2=" << v->chi2() << std::endl;
  auto& assocTrackPtrs = v.get<recob::Track>();
  for (auto trackPtr : assocTrackPtrs) {
    auto track = tracks[trackPtr.key()];
    std::cout << "track with key=" << trackPtr.key() << " has length=" << track->Length() << " and nHits=" << track.nHits() << std::endl;
    for (auto& h : track.hits()) {
      std::cout << "hit peak time=" << h->PeakTime() << std::endl;
    }
  }
}
```

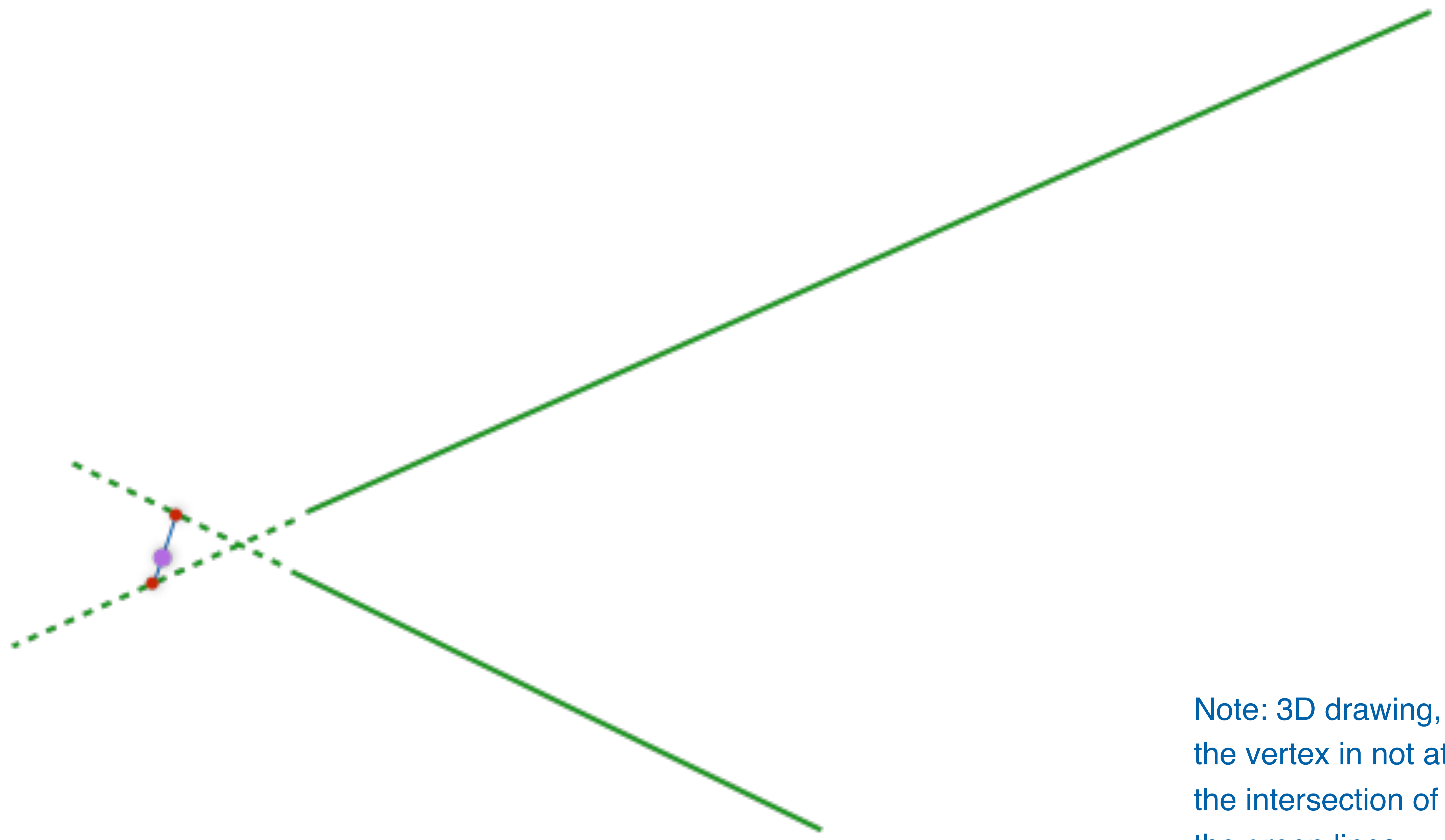
Conclusions

- Code is ready
  - lardataobj feature/cerati_new-recob-vertex-and-fitter
  - larreco feature/cerati_new-recob-vertex-and-fitter
- Not a breaking change, would like to push for this week's release

Backup

Two tracks vertex fit

- Consider the lines defined by the track start position and direction
- Find the two points along the lines with minimum distance
- Propagate the track uncertainties to the two points
- The vertex (position and uncertainty) is computed from the weighted average of the two points



Note: 3D drawing, the vertex in not at the intersection of the green lines

Vertices with >2 tracks

- In case a vertex has more than two tracks, tracks are sorted by number of hits
- The first two are fitted as before to get the 2-track vertex, the others are added as follows
- Consider the line defined by the 3rd track start position and direction
- Find the point along the line with minimum distance to the 2-track vertex
- Propagate the track uncertainties to the point
- The updated vertex (position and uncertainty) is computed from the weighted average of the 2-track vertex and point
- Repeat for 4th track (with 3-track vertex), etc.