Pattern Recognition Porting Preparation

Brett Viren

August 21, 2025

Overview

WCT `clus/` updates for prepare pattern recognition (PR) related porting.

- Starts July 22nd after "steiner" and "UbooneClusterSource to remake all live slices"
- Removed `patrec/` sub-package, some docs now in `clus/docs/porting/`
- Add skeleton for `ToyFiducial` porting
- Initiate `WireCell::Clus::`PR namespace
  - Define "trajectory graph" of PR`::Vertex` node property and PR`::Segment` edge property
  - `Tranjectory` and `TranjectoryView` graph "wrapper" classes
- Linear interpolation, KS-like test metric
- Along the way: Doxygen cleanups, compiler warnings/errors, add support for `clangd` for Eglot.

New in `util/`

`util`: new `Flagged` base class give object up to **64 boolean "flags"**

Define set of flags in an **enum class** for each subclass of `Flagged`

```
enum class SegmentFlags {
    kUndefined = 0, kShowerTrajectory = 1<<1,
    kShowerTopology = 1<<2, kAvoidMuonCheck = 1<<3, kFit = 1<<4,
};
```

`Flagged` is a **mixin base class** used by `PR::Segment` and `PR::Vertex` (so far)

```
class Segment : public Flagged<SegmentFlags> { ... };
class Vertex : public Flagged<VertexFlags> { ... };

// Example usage:
vtx.set_flag(VertexFlags::kNeutrinoVertex);
if (! seg.flags_any(SegmentFlags::kAvoidMuonCheck)) { do_muon_check(seg); }
```

Flags are a **bit index** and we may use **bit-wise** operators.

```
seg.set_flag(SegmentFlags::kFit); // kFit is 1<<4, so bit index is 4
assert(seg.flag_at(4))

auto either = SegmentFlags::kShowerTrajectory | SegmentFlags::kShowerTopology;
if (seg.flags_any(either)) { do_either_shower_thing(seg); }
```

New in `clus/`

File overview of the new `WireCell::Clus::PR` namespace

All **file names** are like `PR*.{h,cxx}`:

- `PRCommon`
- `PRGraphType`
- `PRVertex`
- `PRSegment`
- `PRGraph`
- `PRTrajectoryView`
- `PRTrajectory`
- `PRShower`
- `PR*Functions`

The `PRXxx` files either define a class of same name `PR::Xxx` or they hold conceptually connected set of functions or classes, organized by type dependencies.

I'll next go through the list of files to describe contents and concepts.

PRCommon

Mixins: `HasCluster` and `HasDPCs`

Base mixin classes for a class to hold a `Cluster` or a named map of `DynamicPointCloud`.

`WCPoint` struct and vector

A 3D `Point` and 3 wire indices, similar to WCP struct of same name.

`Fit` struct and vector

A 3D `Point` and fit parameters (dQ, dx, etc).

Point-related functions

- `to_point()` eg, convert `vector<WCPoint>` to `vector<Point>`
- `closest_point()` find point in collection closest to given test point.
- `walk_length()` length to step between points in a collection

PRGraphType defines PR::Graph a "trajectory graph"

Graph type

A setS/setS graph, robust descriptors but unordered iterations.

- The graph maintains an **index** in node/edge properties which can provide order.
- ordered_nodes(g) gives ordered vector of node descriptors

Nodes and edges

- Node property has shared_ptr<PR::Vertex> vertex;
- Edge property has shared_ptr<PR::Segment> segment;

Usage

Do not use "bare" boost:: graph functions to build or use this graph.

- Need care to keep shared_ptr and **index** consistent.
- PRGraph.h provides bookkeeping equivalents to boost:: graph functions.

PRVertex gives PR::Vertex and PR::VertexFlags

Analog to WCP's ProtoVertex

PR::Vertex **IsA** Flagged and **IsA** HasCluster and holds:

- A WCPoint giving "initial point"
- A Fit for fit point and parameters

Implements "chainable setters".

```
WCPoint p;
Cluster* c = ...;
vtx.wcpt(p).cluster(c);
```

Has "getters" named same as setter, but no args.

```
WCPoint p = vtx.wcpt();
```

PRSegment gives PR::Segment and PR::SegmentFlags

Analog to WCP's ProtoSegment

PR::Segment **IsA** Flagged, HasCluster, and HasDPCs and holds:

- vector of WCPoint
- vector of Fit

HasDPCs gives string map to DynamicPointCloud

```
auto dpc_steiner = seg.dpcloud("steiner"); // get
seg.dpcloud("default", dpc_default);       // put
```

PRGraph gives main API for using “trajectory graphs”

- PRGraph.h provides functions in terms of the graph and PR::Vertex and PR::Segment types.
- These PR:: types are always held in shared_ptr.

- make_vertex(graph) returns a new vertex added to the graph.
- add_vertex(graph, vtx) add existing vertex.
- remove_vertex(graph, vtx) remove vertex from graph.
- make_segment() make a segment not (yet) in a graph.
- make_segment(graph, vtx1, vtx2) make a segment as an edge between two vertices.
- add_segment(graph, seg, vtx1, vtx2) add a segment as an edge between two vertices.
- remove_segment(graph, seg) remove segment’s edge from graph.
- find_enpoints(graph, seg) return pair of vertices that are endpoints of segment’s edge.

PR::Graph bookkeeping

The Graphed base class gives vertex/segment a graph descriptor

```
auto edge = seg->get_descriptor();
if (vtx->descriptor_valid()) {
    cout << "Vertex is in the graph!\n";
}
```

Guidance

Always use PRGraph.h functions to assure the bookkeeping is correct.

- PR::add_vertex(graph,vtx) is **good!**

Never use "bare" boost:: functions on "trajectory graphs".

- ~~boost::add_vertex()~~ is **bad!**

PRTrajectoryView provides `PR::TrajectoryView` class

Wrapper around a trajectory graph providing **filtered graph**

```
PR::Graph graph; // user keeps alive
TrajectoryView tv(graph);
// Add objects already in the graph to the filter
tv.add_vertex(vtx);
tv.add_segment(seg);
```

Using `PR::TrajectoryView`

- Can create them as needed but `PR::Trajectory` gives convenient “collection of views”.
- A `PR::TrajectoryView` is a useful base class, eg `PR::Shower` uses it.

PRTrajectory provides PR::Trajectory class

Wrapper around trajectory graph which can make views

```
PR::Graph graph; // user keeps alive
Trajectory t(graph);

auto tv = t.make_view<TrajectoryView>();
auto shwr = t.make_view<Shower>();

cout << "has " << t.size() << " views (2)\n";
for (auto view : t.views()) {
    cout << "has node? " << view->has_node(vtx->get_descriptor()
}
```

Details

- Views are created in shared pointers.
- User must keep the graph alive while using the `Trajectory` (and any views).

PRShower provides PR::Shower

Analog of WCP's WCShower

IsA TrajectoryView with extra:

- start vertex and segment

More needed.

This class may need more data.

PRSegmentFunction.{h,cxx}

Intention is to hold function versions of WCP ProtoSegment methods

WCT implementations can be **much** simpler due to use of graph, holding segment fit as vector<Fit> and dedicated functions factoring common algorithm primitives.

Implemented but none are tested yet

- break_segment(graph, seg, point)
- segment_track_length(seg)
- segment_median_dQ_dx(seg)
- segment_track_length_threshold(seg, thresh)
- segment_geometric_length(seg)

PRShowerFunctions.{h,cxx}

Intention is to hold function versions of WCP WCShower methods.

None are yet implemented, just place-holder declarations

- shower_kinematics(shower)
- update_particle_type(shower)
- longmuon_kinematics(shower)