The Point Tree Example

Brett Viren

November 17, 2023

Contents

1 Overview **1**

2 Meta **2**

3 Preamble **2**

4 What is a (n-ary) tree **2**

5 What is a node **3**

6 Interlude, running a test **4**

7 A value with a node **4**

8 A larger but simple tree **5**

9 Do simple things with this simple tree **5**

10 Scope **6**

11 Scoped view **8**

12 Scoped nodes **9**

13 Scoped point cloud **9**

14 Interlude: disjoint range **10**

15 Scoped k-d tree **12**

16 From k-d tree to n-ary tree **13**

1 Overview

A **point tree** refers to an n-ary tree with **nodes** that hold a set of "local" **point clouds**. The caller may ask any node for a local point cloud by a given **name** (`pcname`). The caller may also ask for a **view** on a subset of these local point clouds in a given **scope**. A **scope** defines a subset of tree nodes and view collects nodes and point clouds in the scope and offers a **k-d tree** spanning

the point clouds. A variety of **iterators** are available to navigate the tree, the collection of point clouds in a scope and to represent the results of k-d tree queries

2 Meta

This document is meant to be read linearly but one may also jump around to find specific examples. It describes the following data structures: point clouds, selections, point trees and k-d trees and related types. Also provided are several C++ code examples in the form of doctest test cases.

The source for this document is in org markup. You may read that form directly or you may use Emacs to *export* the org content to HTML with `C-c C-e h h` or to PDF with `C-c C-e l p`. The org content may also be *tangled* to produce the `doctest_pointtree_example.cxx` source file with `C-c C-v t`. Depending on where you are reading this document you may find these links provide it in the different formats: HTML, PDF, C++.

3 Preamble

The main entry point is the API in `PointTree.h`. Additional headers are for the doctest framework.

```
#include "WireCellUtil/PointTree.h"

#include "WireCellUtil/PointTesting.h"
#include "WireCellUtil/doctest.h"
#include "WireCellUtil/Logging.h"

using namespace WireCell;
using namespace WireCell::PointCloud;
using namespace WireCell::PointCloud::Tree; // for "Points"
using namespace WireCell::PointTesting;     // make_janky_track()
using namespace spdlog;                     // for debug() etc.
```

4 What is a (n-ary) tree

The above are all things that the point tree can provide. But, what is meant by a "tree" here. Specifically what is an **n-ary tree**? Most generally speaking an n-ary tree is a directed acyclic graph that allows for n-way splits but joins are not allowed. This means that there is a unique **path** from the singular **root node** to each of the other nodes in the graph (tree).

In the implementation described here, there is no actual "tree" object per se. Rather, we work only with nodes. A node may have a **parent** node a node may have zero or more **children** nodes. A node with no parent is called the **root** node and this node may be considered as being a representative of a **tree**.

Children nodes are owned by a parent node in an ordered list of nodes. Ownership is expressed by holding a child node in a `std::unique_ptr`. Since the `unique_ptr<>` is important, lets define a type for it. The template argument will be described below.

```
using node_ptr = std::unique_ptr<Points::node_t>;
```

Children in this list are called **sibling** nodes. By default, the order of siblings in this list is determined by the order in which the child node was inserted to the list. The ordering implies that

a child may have a **left sibling** (an "older child") and may have a **right sibling** (a "younger child"). This ordering is reflected in the order that nodes are seen in a depth-first descent of the tree.

A child may be inserted to or removed from a parent. When removed, the caller is given ownership though the `std::unique_ptr`. If the caller drops the removed child, it and any of its children will be destructed. In either case (insertion or removal) the parent node, its parent, etc up to the root node, can receive notification of the change. The ordering of any of the child list may be changed by the user but this will not (currently) trigger notification.

5 What is a node

Generically a node here is of type `WireCell::NaryTree::Node<ValueType>`. The two types are closely integrated. The "node" knows about its "value" and vice versa. This allows generic tree algorithms to operate on the "node side" while application-specific functionality can be provided by the "value side".

For a **point tree**, the value type is the class `WireCell::PointCloud::Tree::Points`. This class provides the following application-specific features:

- node-local point clouds
- scoped point clouds
- scoped k-d trees
- callback methods to react to tree changes

These features are described more below.

Let's start by making a "node-free" `Points` instance with a single point cloud named "3d" that consists of three arrays named "x", "y" and "z".

```
TEST_CASE("point tree example nodeless points")
{
    Points p( { {"3d", Dataset({
            {"x", Array({1.0, 1.0, 1.0})},
            {"y", Array({2.0, 1.0, 3.0})},
            {"z", Array({1.0, 4.0, 1.0})})}) } });

    // Normally, we can get the node that holds this value
    // but in this example that node does not exist.
    CHECK( p.node() == nullptr );

    // But we can access the local point clouds.
    auto& lpcs = p.local_pcs();
    CHECK( lpcs.size() == 1 );
    CHECK( lpcs.find("3d") != lpcs.end() );
}
```

6 Interlude, running a test

After tangling, the above test can be run with these commands:

```
$ waf --target=wcdoctest-util
$ ./build/util/wcdoctest-util -tc='point tree example nodeless points'
[doctest] doctest version is "2.4.11"
[doctest] run with "--help" for options
===============================================================================
[doctest] test cases: 1 | 1 passed | 0 failed | 59 skipped
[doctest] assertions: 3 | 3 passed | 0 failed |
[doctest] Status: SUCCESS!
```

All tests made from this file can be run with a wildcard:

```
$ ./build/util/wcdoctest-util -tc='point tree example*'
```

To see logging output, assign an environment variable `SPDLOG_LEVEL=debug`:

```
$ SPDLOG_LEVEL=debug ./build/util/wcdoctest-util -tc='point tree example*'
```

7 A value with a node

We can extend the previous example to put the value into a node, and thus make our first if somewhat boring tree. Since the value knows the node type (and vice versa) and since we can construct the node with a value, this extension is simple and similar to the above example.

```
TEST_CASE("point tree example single node")
{
    // Normally, we would not create a node on the stack
    // as it could never be a child because as a child
    // must be held by unique_ptr.
    Points::node_t n(Points( { {"3d", Dataset({
            {"x", Array({1.0, 1.0, 1.0})},
            {"y", Array({2.0, 1.0, 3.0})},
            {"z", Array({1.0, 4.0, 1.0})}}) } }));

    // The node directly exposes its value as a data member
    Points& p = n.value;

    // And we can go full circle to get a pointer to the value's node
       .
    Points::node_t* nptr = p.node();

    // This time, that node must exist because we just made it
    CHECK( nptr == &n );
}
```

8 A larger but simple tree

So far, so boring. Let us now make a larger but still fairly simple tree. We will put this into a local function that returns a `unique_ptr<>` (`node_ptr`). It makes use of a function `make_janky_track()` provided by `PointTesting.h` to generate some point cloud.

```
static
Points::node_ptr make_simple_pctree()
{
    // We will return this unique pointer to node
    Points::node_ptr root = std::make_unique<Points::node_t>();

    // Insert first child with a set of named points clouds
    // containing one point cloud build from a track.
    auto* n1 = root->insert(Points({ {"3d", make_janky_track()} }));
    REQUIRE(n1 != nullptr);

    // Insert a second child with a point cloud
    // from a different track.
    auto* n2 = root->insert(Points({ {"3d", make_janky_track(
                        Ray(Point(-1, 2, 3), Point(1, -2, -3)))} }));
    REQUIRE(n2 != nullptr);

    return root;
}
```

Note, we construct the `Points` value as part of the insert. If possible, this will take `std::move()` semantics, otherwise a copy is suffered. Internally, a new node is created and held in a `std::unique_ptr`. However, the **bare** pointer to this new node is returned. This is not transferring ownership and the caller is free to ignore the pointer. At some later time, a caller may also retrieve the child pointer with methods provided on the node side.

9 Do simple things with this simple tree

With this simple tree we may show off some of the functionality.

```
// Loop over children and dump info about them.
static void dump_children(const node_ptr& node)
{
    // Loop over children node "Points" values
    for (auto& cval : node->child_values()) {
        // The named point clouds held by this node value.
        const auto& pcs = cval.local_pcs();

        debug("child node at {} with {} local point clouds",
              (void*)cval.node(), pcs.size());

        // loop over the set of point clouds
        for (const auto& [name, pc] : pcs) {
```

```
            debug("\tchild has pc named \"{}\" with {} points",
                  name, pc.size_major());
        }
    }
}

TEST_CASE("point tree example simple tree operations")
{
    auto root = make_simple_pctree();
    dump_children(root);

    // Find iterator to first child in child list
    CHECK( root->children().size() == 2 );
    auto cit = root->children().begin();

    // Remove it as a child and we get ownership as unique_ptr.
    auto cuptr = root->remove(cit);
    CHECK( root->children().size() == 1 );

    // We can add that orphaned child back.  This transfers ownership
       .
    auto* cptr = root->insert(std::move(cuptr));
    CHECK( root->children().size() == 2 );

    // But, we caught the return and now have a loaned bare pointer
    CHECK( cptr );

    // We should now see the reverse order as above dump.
    dump_children(root);
}
```

Running this test will produce output similar to the following which has some extraneous parts suppressed:

```
$ SPDLOG_LEVEL=debug ./build/util/wcdoctest-util -tc="point tree example simple tree operations"
child node at 0x1f9f0a0 with 1 local point clouds
child has pc named "3d" with 74 points
child node at 0x1fa00a0 with 1 local point clouds
child has pc named "3d" with 74 points
child node at 0x1fa00a0 with 1 local point clouds
child has pc named "3d" with 74 points
child node at 0x1f9f0a0 with 1 local point clouds
child has pc named "3d" with 74 points
```

10 Scope

Before we may explore the `Points` API further, let's take a break and talk about **scopes**. A **scope** defines the extent of a "view" on the point tree. The scope extends from a given node and covers

a **subset of arrays** in a **subset of point clouds** held in a **subset of nodes**. A scope is then defined by the triplet of constraint values that define these subsets:

1. a local point cloud name (`pcname`).
2. a tree descent depth number (`depth`).
3. a list array names (`coords`).

The scope then will cover all local point clouds with the given name that can be found from a depth-first search starting at the given node and limited by the descent depth number and that provide the list of array names. The depth number gives the number of levels to descend. A depth of 1 means only the starting node is considered, 2 includes the starting node and its children, 3 includes also the grand children, etc. The special depth number of 0 means that there is no limit to the depth.

A scope is represented by the `Scope` class. Here are some ways that it may be used.

```
TEST_CASE("point tree example scope")
{
    // A default scope can be constructed.
    // It is a null scope as it would match no local
    // point cloud arrays despite having unlimited depth.
    Scope s;
    CHECK( s.pcname == "" );
    CHECK( s.coords.empty() );
    CHECK( s.depth == 0 );

    // Some non-empty scopes.  Note the case sensitivity.
    Scope s0{ "pcname", {"x","y","z"}, 0};
    Scope s1{ "pcname", {"x","y","z"}, 1};
    Scope s2{ "PCNAME", {"x","y","z"}, 0};
    Scope s3{ "pcname", {"X","Y","Z"}, 0};
    Scope sc{ "pcname", {"x","y","z"}, 0};

    // Scopes can be compared for equality
    CHECK( s0 == s0 );
    CHECK( s0 == sc );
    CHECK( s0 != s1 );
    CHECK( s0 != s2 );
    CHECK( s0 != s3 );

    // A scope has a std::hash().
    CHECK( s0.hash() == sc.hash() );
    CHECK( s0.hash() != s1.hash() );
    CHECK( s0.hash() != s2.hash() );
    CHECK( s0.hash() != s3.hash() );

    // So, it may be used as a key.
    std::unordered_map<Scope, size_t> m;
```

```
    m[s1] = 1;
    m[s2] = 2;
    CHECK( m[s0] == 0 );          // size_t default value
    CHECK( m[s1] == 1 );
    CHECK( m[s2] == 2 );

    // One can also print a scope
    debug("Here is a scope: {}", s0);
}
```

That `debug()` will show a line like:

```
Here is a scope: <Scope "pcname" L0 x,y,z>
```

11 Scoped view

The point tree will provide to the caller a **scoped view** object (`ScopedView`) that collects the information that from the point tree that is in a given scope.

```
TEST_CASE("point tree example scoped point cloud")
{
    auto root = make_simple_pctree();

    // Specify point cloud name and the arrays to select.
    // We take the default depth number of 0.
    Scope scope{ "3d", {"x","y","z"} };

    // The corresponding scoped view.
    auto const & sv = root->value.scoped_view(scope);

    CHECK(sv.npoints() > 0);
}
```

The `scoped_view()` method returns a `ScopedView<double>` by default. The numeric template type must match the underlying type of the coordinates given by the scope.

A it is a **view**, the `sv` object lifetime can not be longer than that of the tree (`root` node). If a node is **inserted** to a viewed scope, that view will be **mutated** by appending to the view's contents. On the other hand, if a node is **removed** from a viewed scope then the view is **invalidated**. The caller must retrieve the scoped view to get a valid view.

We now look at the major parts of the scoped view.

- vector of nodes
- vector of local PCs of the scope's `pcname`
- vector of selections on these PCs with arrays named as given by the scope's `coords`
- a k-d tree built on these selections

The three vectors have the same size and order.

12 Scoped nodes

The **scoped nodes** are presented as a vector of pointers to nodes. The vector is nominally in the order in which the nodes were encountered by the scoped depth-first descent. In the case that additional in-scope nodes were added to a tree after the view was created they will appear appended to the scoped node vector.

```
TEST_CASE("point tree example scoped nodes")
{
    auto root = make_simple_pctree();

    Scope scope{ "3d", {"x","y","z"} };
    auto const & sv = root->value.scoped_view(scope);

    // Vector of pointers to n-ary tree nodes
    auto const& snodes = sv.nodes();
    CHECK(snodes.size() > 0);
    for (const auto& node : snodes) {
        const auto& lpcs = node->value.local_pcs();
        REQUIRE(node);
        CHECK(lpcs.find("3d") != lpcs.end());
    }
}
```

13 Scoped point cloud

The **scoped point cloud** collects the **local point cloud** from each in-scope node that is named by the scope's `pcname`. The scoped point cloud is presented as a vector of references to the original point cloud and in the same order as the scoped node vector. As the scoped point cloud keeps the individual local point clouds distinct it has the structure of an "array-of-array-of-points". A `disjoint_range` may be used to iterate over the points in a scoped point cloud as if it were a flat collection of points (this type is described more below).

Let's now make a scoped point cloud using the same simple tree introduced above.

```
TEST_CASE("point tree example scoped nodes")
{
    auto root = make_simple_pctree();

    Scope scope{ "3d", {"x","y","z"} };
    auto const & sv = root->value.scoped_view(scope);

    auto const& snodes = sv.nodes();
    auto const& spcs = sv.pcs();
    CHECK(spcs.size() == snodes.size());

    // A scoped point cloud is a vector of
    // references to node-local point clouds
    for (const Dataset& pc : spcs) {
```

```
            debug("pc {} arrays and {} points",
                  pc.size(), pc.size_major());
        }
}
```

14 Interlude: disjoint range

A scoped point cloud is a vector of (references to) individual node-local point clouds. We have some contradictory requirements in how we wish to use a scoped point cloud. On the one hand, we wish to keep the individual node-local point clouds distinct as each node represents something (eg a "blob"). On the other hand, we wish to avoid making copies as point clouds can be rather large. On the gripping hand, we wish to use a scoped point cloud as if it were contiguous.

The scoped point cloud structure satisfies the first two. To provide a contigous view of a scoped point cloud we combine the already introduced coordinate range with a **disjoint range** that we introduce here.

A disjoint range allows for a "vector of vectors of elements" to look like a flat "vector of elements". Below, will see it used internally in the interface to k-d trees, but here we will show how to use it directly.

```
#include "WireCellUtil/DisjointRange.h"
TEST_CASE("point tree example scoped point cloud disjoint")
{
    // As above.
    auto root = make_simple_pctree();
    auto spc = root->value.scoped_view(Scope{ "3d", {"x","y","z"} }).
       pcs();

    size_t npoints = 0;
    for (const Dataset& pc : spc) {
        npoints += pc.size_major();
    }

    // This part is a bit verbose, see text.
    // First we select a subset of arrays from each point cloud.
    std::vector<Dataset::selection_t> sels;
    for (Dataset& pc : spc) {
        sels.push_back(pc.selection({"x","y","z"}));
    }
    // Then we wrap them as coordinate points.
    using point_type = coordinate_point<double>;
    using point_range = coordinate_range<point_type>;
    std::vector<point_range> prs;
    for (auto& sel : sels) {
        prs.push_back(point_range(sel));
    }
    // Finally we join them together as a disjoint range
    using points_t = disjoint_range<point_range>;
```

```
    points_t points;
    for (auto& pr : prs) {
        points.append(pr.begin(), pr.end());
    }

    // Finally, we can perform a simple iteration
    // as if we had a single contiguous selection.
    CHECK( points.size() == npoints );

    for (auto& pt : points) {
        CHECK( pt.size() == 3);
    }

    // We know the scoped PC has two local PCs.
    // Pick an index in either
    const size_t in0 = 0.25*npoints; // in local PC at index 0
    const size_t in1 = 0.75*npoints; // in local PC at index 1

    // Iterator to the first point
    auto beg = points.begin();

    // Iterators to points inside the flat disjoint_range.
    // Eg, as we will see returned later from a k-d tree query.
    points_t::iterator mit0 = beg + in0;
    points_t::iterator mit1 = beg + in1;

    // Full circle, find that these points are provided
    // by the first and second major range.
    size_t maj0 = points.major_index(mit0);
    size_t maj1 = points.major_index(mit1);

    CHECK( 0 == maj0 );
    CHECK( 1 == maj1 );
}
```

The central part of this example is rather verbose as we have to implement a few layers of conversion. However, these layers are largely light-weight and involve iterators and do not copy any array data. When used in the interface to k-d trees, these layers are hidden from the caller.

A note on performance. One can iterate though a disjoint range at linear cost. However, take care that while `disjoint_range` provides a random-access API its cost is not $\mathcal{O}(1)$. True random access requires an ordered walk of a `std::map` to find the local point cloud followed by $\mathcal{O}(1)$ access of elements of its arrays. To mitigate the cost of finding the local point cloud the `disjoint_range` will cache its last location. This allows the walk to the next index to be shorter on average than if it were to start at the beginning of the map each time. When the caller must perform many random accesses it is advised to do so with pre-ordered indices so that the relative walk of the map is minimized. The absolute speed of this "random" access is greatly improved when compiling with `-O2`.

15 Scoped k-d tree

The scoped view also provides a "k-d tree" formed on the scoped point cloud (specifically its selection). The k-d tree "knows" about the disjoint nature of the scoped point cloud and it can grow as nodes are added in scope.

A k-nearest-neighbor (knn) and radius search is implemented by the k-d tree interface. Results of either query are in the form of a vector of a pair consisting of a disjoint iterator (see `disjoint_range` above) and the distance from the point referenced by the iterator and the query point. The iterator yields a vector-like point as well as a **major index** and a **minor index**. The major index counts into the vectors provided by the scoped view and the minor index counts to the point within the local point cloud at that index. These two indices allow the caller to navigate from each point in a k-d tree query result to the node that provided the point or to get point attributes in the point cloud that provided the point other than those that were used for k-d tree coordinates.

We will see some of this navigation below but lets first just exercise a k-d tree query.

```
TEST_CASE("point tree example scoped k-d tree")
{
    auto root = make_simple_pctree();

    // Form a k-d tree query over a scoped point cloud.
    Scope scope = { "3d", {"x","y","z"} };
    const auto& sv = root->value.scoped_view(scope);
    const auto& skd = sv.kd();

    // Some query point.
    const std::vector<double> origin = {0,0,0};

    // Find three nearest neighbors.
    auto knn = skd.knn(3, origin);
    CHECK( knn.size() == 3 );

    // Get the underlying scoped PC.
    const auto& spc = sv.pcs();

    // Loop over results and refrence back to original
    // scoped PC at both major and minor range level.
    for (size_t pt_ind = 0; pt_ind<knn.size(); ++pt_ind) {
        auto& [pit,dist] = knn[pt_ind];

        const size_t maj_ind = skd.major_index(pit);
        const size_t min_ind = skd.minor_index(pit);
        debug("knn point {} at distance {} from query is in local
           point cloud {} at index {}",
              pt_ind, dist, maj_ind, min_ind);
        const Dataset& pc = spc[maj_ind];
        for (const auto& name : scope.coords) {
            debug("\t{} = {}", name, pc.get(name)->element<double>(
               min_ind));
```

```
        }
    }
}
```

Some of the logging output from this test would be like:

```
knn point 0 at distance 0 from query is in local point cloud 1 at index 37
x = 0
y = 0
z = 0
knn point 1 at distance 0 from query is in local point cloud 0 at index 37
x = 0
y = 0
z = 0
knn point 2 at distance 0 from query is in local point cloud 0 at index 36
x = 0
y = 0
z = 0
```

This example shows a k-nn query over a scope for the three points nearest the given query point (`origin`). It ends in a loop over the results. Each result provides an iterator and a distance (from point to query point). The iterator is over the `disjoint_range` that the k-d tree uses to hold a `coordinate_range` representation of each local point cloud. We convert the iterator to a corresponding major and minor index. We use the major index to locate the point cloud that provided the point. The minor index is used to index arrays in the point cloud. In this example, we merely access the same attributes that supplied the coordinates ("x", "y" and "z"). Had the point cloud provided additional arrays we could off course access their attributes with the minor index as well.

16 From k-d tree to n-ary tree

Very similarly, we may navigate from the k-d tree result iterator via the major index to get the node that provided the point cloud that provided the point.

```
// a little helper
static const Dataset& get_local_pc(const Points& pval, const std::
   string& pcname)
{
    const auto& pcs = pval.local_pcs();
    auto pcit = pcs.find(pcname);
    if (pcit == pcs.end()) {
        raise<KeyError>("no pc named " + pcname);
    }
    return pcit->second;
}

TEST_CASE("point tree example scoped k-d tree to n-ary nodes")
{
    auto root = make_simple_pctree();
```

```
    // Form a k-d tree query over a scoped point cloud.
    Scope scope = { "3d", {"x","y","z"} };


    // Get the scoped view parts
    const auto& sv = root->value.scoped_view(scope);
    const auto& skd = sv.kd();
    const auto& snodes = sv.nodes();

    // Some query point.
    const std::vector<double> origin = {0,0,0};

    // Find three nearest neighbors.
    auto knn = skd.knn(3, origin);
    CHECK( knn.size() == 3 );

    for (size_t pt_ind = 0; pt_ind<knn.size(); ++pt_ind) {
        auto& [pit,dist] = knn[pt_ind];

        // This time use major index to get node.
        const size_t maj_ind = skd.major_index(pit);
        const auto* node = snodes[maj_ind];

        debug("knn point {} at distance {} from query at node {} with
            {} children",
              pt_ind, dist, maj_ind, node->children().size());
    }
}
```

Notes:

- scope is more restrictive than `depth()`. Misalignment between `major_index` and depth iterator will occur if the `scope.pcname` limits which point clouds are in scope