Wire-Cell Toolkit Point Cloud

Brett Viren

September 26, 2022

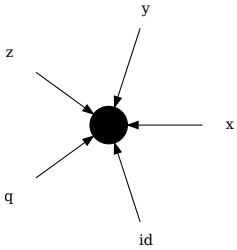
Topics

- Points and point data
- Point cloud, point data array and dataset
- k-d tree operations
- Data representation conversions
- WIP: extending point-cloud to point-graph

Point

An abstract entity, no intrinsic meaning.

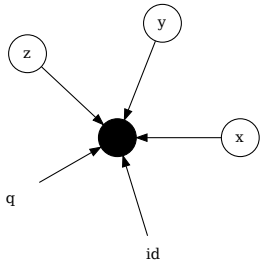
Point data



We may *associate* information with a point.

- shapes: scalar, vector, matrix, tensor
- numeric types: integer or floating point
  - homotypic if non-scalar

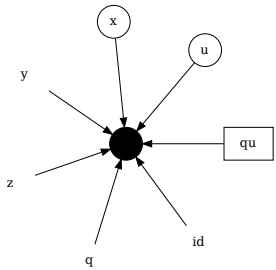
Data interpretation, *eg* coordinates



We may *interpret* specific *point data* in some way.

- An ordered set of n *coordinates* may provide a *position* in an n dimensional Cartesian space.
- Interpretation are *extrinsic* to the point and the associated data.

Shared interpretations



Different interpretations of subsets of point data.

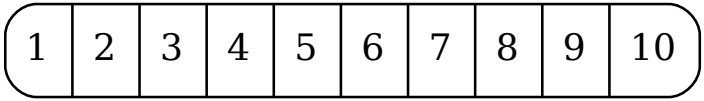
- The “x” point-data interpreted as part of a 3D position may also be used as part of a 2D position (projected x-u wire view).
- A charge, “qu” may be found with the projected 2D position and then later used along with 3D positions.

Point cloud

An abstract, **ordered** collection of N *points*.

- Well defined ordering of points (but may be arbitrary).
- An extrinsic **point index** reflects the ordering.

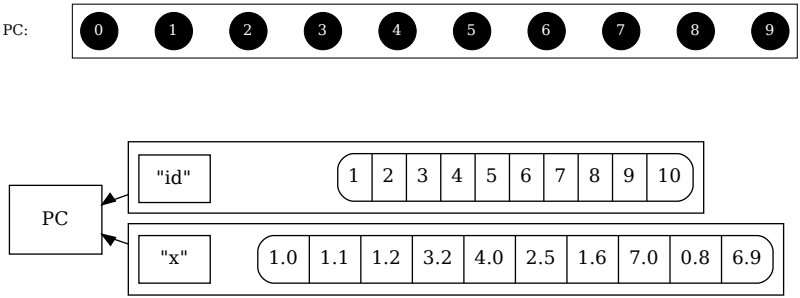
Point-data array



Collect all of one type of point data for the points in a point cloud into an array.

- The *point-cloud index* also identifies associated point data in the array.
- Array elements have common data type and shape.
  - (here, scalar integers one larger than point index)

Point-cloud dataset



Associate multiple *point-data arrays* to a point cloud.

- Each array is identified by a “name” in the context of the dataset.
- Heterogeneous type and shape across the arrays, but common length.

WireCell::PointCloud

`Array`	model of a *point-data array*
`Dataset`	model of a *point-cloud dataset*

PointCloud::Array

- Provide *type-erased* array data wrapper.
  - Required to form a heterotypic collection.
- Read-only, zero-copy sharing or read-write copy of user array.
- Read-only, zero-copy and typed array data access wrappers:
  - `span<T>` a flat `vector<T>` like view of underlying array
  - `boost::multi_array<T,NDim>` full featured multi-dimensional array operations
- **Minimal** but efficient set of array operations.
  - Essentially only: `append(Array)` which assures type/shape constraints.

PointCloud::Dataset

- Access an `Array` by its associated name.
- Assure array length constraints.
- Implement `append(Dataset)`.
  - Assure completeness, shape, type constraints of appended tail dataset.
- Call user-provided *callback hooks* on successful `append()`.
  - Needed for dynamic k-d tree support (comming up).
- Retrieve collection of references to `Array`'s via list-of-names.

WireCell::PointCloud code snippet

```
#include "WireCellUtil/PointCloud.h"
using namespace WireCell::PointCloud;

Dataset d;
// Add an integer array named "one" of shape (5,)
d.add("one", Array({1,2,3,4,5}));
// Add a double array named "two" of shape (5,)
d.add("two", Array({1.1,2.2,3.3,4.4,5.5}));

auto sel = d.selection({"two","one"});
const Array& one = sel[1];
assert(sel[0].get().num_elements() == 5);

const auto& one = d.get("one");
```

Many other ways to make `Array` and add them to `Dataset`.

Array:: and Dataset::metadata()

```
using metadata_t = Configuration;

metadata_t& metadata();
const metadata_t& metadata() const;
```

- Type is `WireCell::Configuration`,
  - aka JsonCPP's `Json::Value`.
- Merely carried and not directly utilized by `Array/Dataset`.
  - Utilized in I/O related conversions (coming up).
- Users are free to stash their own structured data.

Point-cloud position queries

We may interpret certain arrays in a dataset as holding coordinate point data.

- Each array represents a location in a given Cartesian dimension.
- Must be scalar valued but may be homotypic integer or floating point.

Position queries

knn	the k'th nearest neighbors to query position.
radius	all point positions within some *metric distance* to a query position.

Results in:

index	an array of point indices into the original dataset.
distance	the *metric distance* between point and query positions.

Metric distance

A *distance* between two positions in a space requires a *metric*.

L2	the usual, **but squared** Cartesian distance
L1	sum of steps, each strictly taken in one dimension
SO2	2D angular distance
SO3	3D angular distance

The query *radius* and returned *distances* are expressed in this metric.

- *eg*, units are $[length]^2$ for choice of the L2 metric.

`WireCell::KDTree` for position queries

- Utilizes a `Dataset`
- Provides a thin wrapper around `nanoflann`
  - Simplifies and regularizes `nanoflann` API.
  - Converts complex `nanoflann` templated types to simple variable values.
- Common result set type for both `knn` and `radius` searches.

`WireCell::KDTree` code snippet

```
#include "WireCellUtil/KDTree.h"
using namespace WireCell::KDTree;
using namespace WireCell::PointCloud;
void func() {
    Dataset d = ...;
    std::vector<double> query_pos = {1,2,3};

    auto qptr = query_double(d, {"x","y","z"});

    size_t k = 3;
    auto knn = qptr->knn(k, query_pos);
    const size_t nfound = knn.index.size();
    for (size_t ifound=0; ifound<nfound; ++ifound) {
        cerr << ifound << ":" << " index=" << knn.index[ifound]
             << " distance=" << knn.distance[ifound] << "\n";
    }
    double rad = 5*units::cm;
    auto radn = qptr->radius(rad*rad, query_pos);
    // use radn just like knn....
```

WireCell::KDTree::query_TYPE()

For TYPE $\in$ {int, float, double}

```
std::unique_ptr<Query<TYPE>>
query_TYPE(PointCloud::Dataset& dataset,
           const PointCloud::name_list_t& selection,
           bool dynamic = false,
           Metric mtype = Metric::l2simple);
```

- The TYPE in the function name needed to hide nanoflann templates.
- The `selection` names the arrays in `dataset` to use as coordinates.
- The `dynamic` enables `Dataset::append()` callback to update k-d tree.
- A `unique_ptr` needed, wrapped nanoflann objects are not copyable.

Dataset I/O with TensorTools.h API

PointCloud::Array $\longleftrightarrow$ ITensor
PointCloud::Dataset $\longleftrightarrow$ ITensorSet

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

ITensor::pointer as_itensor(const PointCloud::Array&);
PointCloud::Array as_array(const ITensor::pointer&, bool);
ITensorSet::pointer as_itensorset(const PointCloud::Dataset&);
PointCloud::Dataset as_dataset(const ITensorSet::pointer&, bool);
```

- If `bool` is `true`, utilize zero-copy data sharing, requires programmer care. Default is `false`
- The `ITensor::ident()` mapped to `Dataset::metadata()["ident"]`.
- `ITensorSet::metadata()["_dataset_arrays"]` holds list of `Array` names known in the `Dataset`.

WIP: extending point-cloud to point-graph

Extend `Dataset` to `boost::graph`:

Graph vertex:

- Simple node type, perhaps holding only an array *index*.
- May then rely on `Dataset` to hold rich node data.

Graph edge:

- Extend `Dataset` to store edges as $2 \times N_{edge}$ array
  - requires `Dataset::add()` to relax check on N.
  - requires `Dataset::append()` to relax check on array completeness.
  - "edge data arrays" may be accommodated.
- Or, new `WireCell::Graph` with vertex and edge `Dataset`'s?
- `Dataset` $\longleftrightarrow$ `boost::graph` converters. *ibid* `TensorTools`.