

Wire-Cell Toolkit Point Cloud

Brett Viren

October 12, 2022

Topics

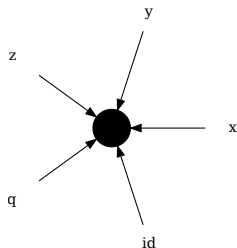
- Points and point data
- Point cloud, point data array and dataset
- k-d tree operations
- Data representation conversions
- WIP: extending point-cloud to point-graph

Point



An abstract entity, no intrinsic meaning.

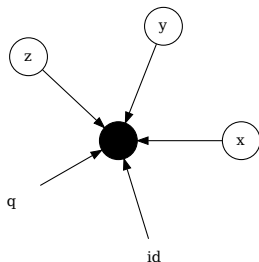
Point data



We may *associate* information with a point.

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

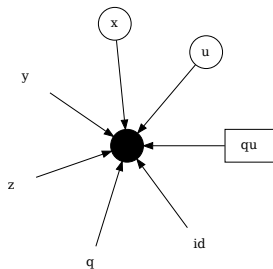
Data interpretation, eg coordinates



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

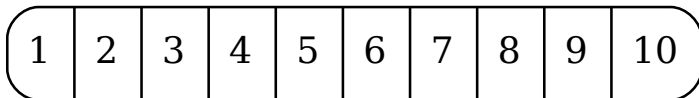


An abstract, **ordered** collection of N *points*.

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

Point-data array

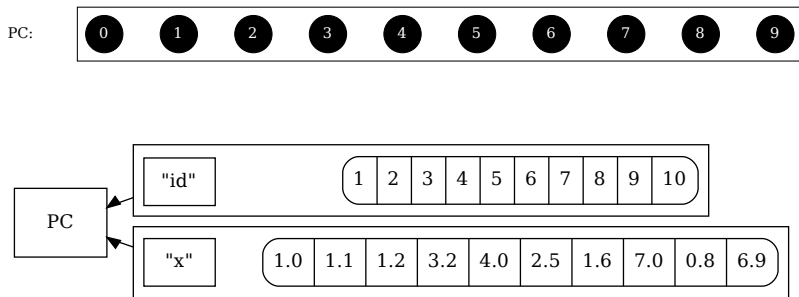
PC:



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.

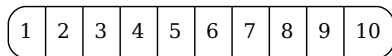
- Each array is identified by a “name” in the context of the dataset.
- Heterogeneous type and shape across the arrays, but common length.

WireCell::PointCloud

Array model of a *point-data array*

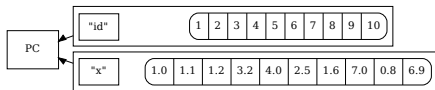
Dataset model of a *point-cloud dataset*

PointCloud::Array



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

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.
 - ▶ eg "x" array of X-coordinates.
- Any set of scalar and common numeric type arrays may provide coordinates.

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 two arrays:

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

- Uses a `Dataset`
- Provides a thin wrapper around `nanoflann`
 - ▶ Simplifies and regularizes `nanoflann` API.
 - ▶ Converts complex `nanoflann` templated types to option variables.
- 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 $\text{TYPE} \in \{\text{int}, \text{float}, \text{double}\}$

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

- The TYPE is coordinate numeric type.
- 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_itor(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`.

Related ongoing I/O work

Get round trip I/O working for:

- Frame \longleftrightarrow “Frame file”
- Cluster \longleftrightarrow “Cluster file”

Wish to deprecate these “direct I/O” patterns and instead standardized on intermediate Tensor representation.

- Dataset \longleftrightarrow Tensor Set \longleftrightarrow “Tensor file”
- Frame \longleftrightarrow Tensor Set \longleftrightarrow “Tensor file”
- Cluster \longleftrightarrow Tensor Set \longleftrightarrow “Tensor file”

A WCT “tensor file” is JSON+Numpy files in Zip/Tar streams. Essentially follows HDF5 schema. So, expect it easy to add:

- Tensor Set \longleftrightarrow HDF5

WIP: extending point-cloud to point-graph

Essential idea: use a “node” and an “edge” Dataset.

node Exactly a point-cloud Dataset.

edge A second Dataset with "tail" and "head" arrays holding point indices in to the **node** dataset. May have additional arrays to hold *edge features*.

Benefits:

- Leverage existing converters to Tensor Set representations and file I/O.
- Easy to use alongside `boost::graph` representations.