WCT Point Cloud Support

Brett Viren

September 23, 2022

1 Definition of terms:

point an abstract entity to which we may associate information

point data a unit of information associated to a point

coordinate point data associated with a location along some dimension

position a set of n coordinates in an n dimensional space associated with a point

point cloud (PC) an abstract, ordered collection of *points*

point index (or just *index*) a number identifying a point in a PC

point array the units of a common type of point data across the points in a PC ordered in step with the PC

coordinate array a point array eith each element interpreted a coordinate

position array an ordered set of n coordinate arrays

dataset a collection of identified point arrays of common *major axis* size.

k-d tree a data structure relating a number N of k dimensional positions of points in a PC to their *index*.

distance metric a function returning a scalar value given two positions and which represents some notion of distance. A common metric is L2 which is the square of the Cartesian distance.

kNN search locating k (different k than in "k-d tree") indices of points in a PC which have positions that are the nearest neighbors to a given position.

radius search locate all indices of points in a PC which have positions within a fixed distance metric from a given position. Note the radius is measured against a distance metric (thus using L2 the radius has units `[(length)^2]`).

2 Comments

- A point is not a position. Rather, one or more positions, possibly of different dimension, may be associated with a point.

- Point arrays in a dataset may have different shape as long as they all have identical sizes of *major axis*. The major axis is the first index when written in the usual C++/Python manner.
- A dataset identifies its arrays by name.
- Each coordinate array is independent. Thus, one may have a 3D position array composed of the coordinate arrays with names ("x","y","z") and a 2D position array representing some projection into "w", a linear combination of "y" and "z" with the set of coordinate arrays ("x","w").

3 WCT Implementation

The wire-cell toolkit provides support for *point clouds* via three inter-operating APIs.

- `PointCloud` provides `Array` and `Dataset` classes for the abstractions described above.
- `KDTree` provides *knn* and *radius search* operations that operate on `PointCloud` objects.
- `TensorTools` provides conversion between `Array` and `ITensor` and between `Dataset` and `ITensorSet`.

The remainder of this note provides some detail of these API.

4 `PointCloud` API

Quick taste of some code:

```
#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);
```

See `util/test/test_pointcloud.cxx` for more examples. The rest of this section describes `Array` and `Dataset`.

4.1 `Array`

The `PointCloud::Array` is a simple container of array like data. It's primary purpose:

- Be held by `Dataset`.
- Erase the C++ numeric type and shape information of the underlying array to enable `Dataset` to be heterogeneous.

- Allow option to share user array data or to maintain an internal copy.
- Support appending along the major axis.
- Supply array views in useful types (flat vector as `boost::span<T>` or shaed `boost::multi_array<T,NDIM>`).

Here are more creation examples:

```
// some user data
std::vector<int> user(10, 0);

// shape: (2 rows, 5 columns), user data is shared
Array arr1(user, {2,5}, true);

// special, shape: (10,) not shared
Array arr2(user);

// copy/move constructor/assignment all supported
arr1 = arr2;
```

The underlying data can be retrieved in a number of constant, read-only forms that are more useful for custom code operations. These forms include:

```

// Return a boost::span (same as C++20 std::span and that is
// std::vector like), of the flattened array.
auto flat = arr1.elements<int>();

// as properly shapped boost::multi_array
auto ma = arr1.indexed<int, 2>();

// fixme: is an Eigen3 array interface useful?
```

These formats do not impose a copy and they will throw a `ValueError` if the requested type or shape is not compatible with the underlying array content.

The contents of an `Array` may be fully replaced which drops and does not modify previous contents:

```
// Just as when we initially constructed arr1 above.
arr1.assign(user.data(), {2,5}, true);

// array of shape (10,) not shared
arr1.assign(user);
```

One of the few array-like operations supported is to *append* to an `Array`. If the underlying data is shared, the `append()` methods will first copy that data to an internal buffer and drop the original shared array (copy-on-write semantics).

```

// Reuse user data
Array tail(user, {2,5}, true);

```

```
// append, causes internal copy so "user" not modified.
arr1.append(user);
```

The array begin appended (`tail`) must be of the same numeric type as the current array and the sizes of all non-major axes of `tail` must match those same axes of the current array. The following represents shapes which are legal to use in an append and their resulting shape:

```
(N,n2,n3,...) + (M,n2,n3,...) = (N+M,n2,n3,...)
```

And `Array` also carries but does not in any way use a *metadata object* implemented as `WireCell::Configuration` (a JsonCPP object).

```
Array a;
auto& md = a.metadata();
md["foo"] = "bar";
```

4.2 Dataset

A `PointCloud::Dataset` provides a collection of `Array` with minimal operations. It's primary purposes:

- Hold instances of `Array`, each identified by a "name" of type `std::string`.
- Assure all accepted `Array` have common sized major axes.
- Implement the appending of another `Dataset` while keeping assurances.
- Accept user-provided callback hooks to be called on successful append.
- Provide various means of access to the collection of the arrays.

Here shows the successful adding of two arrays followed by a rejection of a misshapen third.

```
Dataset d;
d.add("one", Array({1,2,3,4,5}));
d.add("two", Array({1.1,2.2,3.3,4.4,5.5}));
d.add("broken", Array({1}));    // throws ValueError
```

One may `add()` higher dimension arrays as long as the number of their "elements" match. In general, for a point data of shape `(v1, ...)` the point data array must be of shape `(nele, v1, ...)` to be added to a dataset with `num_elements()` returning `nele`.

The names of the held arrays can be retrieved:

```
for (const auto& name : d.keys()) {
    // ...
}
```

One `Dataset` ("tail") may be appended to another `Dataset` ("head"). This operation merely appends each `Array` in "tail" to the array of the same name in "head". A `ValueError` is thrown if any names in "head" are not in "tail" or if any in the set of matching arrays from "tail" have differing number of elements.

```
Dataset d2 = ...;
d.append(d2);
```

A `Dataset` may be copy/move assigned/constructed.

```
Dataset d2 = d;
Dataset d3(d);
```

Like `Array`, `Dataset` carries but does not use a metadata object.

```
Dataset d;
auto& md = d.metadata();
md["foo"] = "baz";
```

5 `TensorTools` API

The `TensorTools` API provides conversion functions between `Array` and `ITensor` and between `Dataset` and `ITensorSet`. This allows point cloud information to be sent between nodes of the WCT data flow graph and to be sent through I/O with files. The `ITensor` representation is first described and the conversion functions.

5.1 `ITensor` representation

A `Dataset` is mapped to an `ITensorSet` and each of its named `Array` items to an `ITensor` in the set. The two representations are close, but not exact.

- On creation, the `ItensorSet` metadata will be given a special attribute `"_dataset_arrays"` holding an array of strings representing the names of the `Array` items and in the same order as the `ITensor` vector which the set holds.
- When read, if this special attribute is missing then a `"name"` attribute from the metadata of each `ITensor` is used to provide the `Array` name in the `Dataset`. If this attribute is missing a name is generated as `"array%d"` which will be interpolated with the index of the corresponding `ITensor`.
- The `ident` value of the `ITensorSet` may be provided by an `"ident"` attribute of the `Dataset` metadata, else zero is stored. In the reverse direction, the `Dataset` will be given an `"ident"` metadata attribute holding the value from the `ITensorSet`.
- Otherwise, all `Array` and `Dataet` metadata objects are pass unchanged to/from those of `ITensor` and `ITensorSet`.

5.2 Conversion functions

Each of the four possible conversions has a corresponding function. Their interface is very simple and the `WireCellAux/TensorTools.h` header file gives the essentials:

```
// PointCloud support

/// Convert Array to ITensor.  Additional md may be provided
```

```
ITensor::pointer as_itensor(const PointCloud::Array& array);

/// Convert a Dataset to an ITensorSet.  The dataset metadata is
/// checked for an "ident" attribute to set on the ITensorSet and
/// if not found, 0 is used.
ITensorSet::pointer as_itensorset(const PointCloud::Dataset& dataset)
   ;

/// Convert an ITensor to an Array.  A default array is returned
/// if the element type of the tensor is not supported.  Setting
/// the share as true is a means of optimizing memory usage and
/// must be used with care.  It will allow the memory held by the
/// ITensor to be directly shared with the Array (no copy).  The
/// user must keep the ITensor alive or call
/// Array::assure_mutable() in order for this memory to remain
/// valid.
PointCloud::Array as_array(const ITensor::pointer& ten, bool share=
    false);

/// Convert an ITensorSet to a Dataset.  The "ident" of the
/// ITensorSet will be stored as the "ident" attribute of the
/// Dataset metadata.  If a "_dataset_arrays" key is found in the
/// ITensorSet metadata it shall provide an array of string giving
/// names matching order and size of the array of ITensors.
/// Otherwise, each ITensor metadata shall provide an "name"
/// attribute.  Otherwise, an array name if invented.  The share
/// argument is passed to as_array().
PointCloud::Dataset as_dataset(const ITensorSet::pointer& itsptr,
    bool share=false);
```

Note that `share` may be passed as `true` to enable the optimization of zero-copy with optional copy-on-write. It is `false` by default as the user must assure that the `ITensorSet` or `ITensor` remains alive.

The `aux/test/test_tensor_tools.cxx` unit test contains more examples.

6 `KDTree` API

The `KDTree` API provides a binding between a `Dataset` and the `nanoflan` library whic provides k-d tree data structure and *knn* and *radius search* operations. The goals of `KDTree` is to simplify, regularize and hide the `nanoflan` API. See the appendix below for notes that attempt to document `nanoflann`.

Here is a simple example:

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

void func() {
    Dataset d = ...;             // some Dataset

    // Unique pointer to a KDTree::Query
    auto qptr = query_double(d, {"x","y","z"});

    // Some point in (x,y,z) space
    std::vector<double> query_pos = {1,2,3};

    // do a knn search
    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";
    }
    // do a radius search
    double rad = 5*units::cm;
    auto radn = qptr->radius(rad*rad, query_pos);
    // ... etc similar type of iteration as above
}
```

Notes,

- A family of functions, `KDTree::query_TYPE()` are provided with `TYPE` being `int`, `float` or `double`.
- These labels must match the types of the point coordinate arrays. The coordinate arrays are named with a vector `{"x","y","z"}`.
- The `qptr` is a `unique_ptr` and so will take care of destruction of its `KDTree::Query` when it falls out of scope.
- The `radius()` search function expects a radius in the same units as the distance metric, which default to L2 and so a value with units that of squared length is passed.

Two optional argument may be provided to the `query_TYPE()` functions:

```
auto qptr = query_double(d, names, dynamic, metric);
```

The `dynamic` argument is a `bool` defaulting to `false`. If `true` the `KDTree::Query` will register a callback with the passed `Dataset` so that if the user performs a successful `Dataset::append()` the underlying k-d tree will be updated.

The `metric` is an enum in the `KDTree::Metric::` namespace and may be any in: `{ l2simple, l1, l2, so2, so3 }` which directly map to the metrics implemented by `nanoflann`. The default is `l2simple` which is an L2 (squared Cartesian distance) appropriate for low dimension space.

The example above prints the indices of the points in the point cloud for which the associated positions matched the search criteria. It also prints the metric distance from the point positions to the query point position. The user likely wishes to access other point data arrays held in the `Dataset` which are associated with the found points.

Assuming `knn` and the context from the above example, here are some ways to do just that.

```
// Get a 1D float array by name as a std::vector-like "span"
auto my_1d = d.get("my_1d").elements<float>();

// Get a 2D int array as a boost::multi_array
auto my_2d = d.get("my_2d").indexed<int, 2>();

for (size_t ifound=0; ifound<nfound; ++ifound) {
    size_t index = knn.index[ifound];
    cerr << "1d value: " << my_1d[index]
         << " 2d values: " << my_2d[index][0] << "," << my_2d[index
            ][1]
         << "\n";
}
```

7 Appendix: nanoflann

This appendix provides some developer documentation for `nanoflann` as what it provides is rather slim. What info is available is at the repo readme and in the doxygen docs. This information is not required to use WCT PCs.

7.1 Dataset

A `nanoflann` *dataset adaptor* must be provided for each unique type of PC data. One that makes use of Eigen3 arrays is provided by `nanoflann`. WCT provides one for `boost::multi_array` that comes from an `ITensor` as described above. A dataset adaptor must provide two methods and may provide a third, optional method. As copied from comments in the source and embelished these are:

```
template<...>
class MyDatasetAdaptor {
public:
    using point_type = ...;

    // Must return the number of data poins
    inline size_t kdtree_get_point_count() const
    {
        // return npoints
    }

    // Must return the dim'th component of the idx'th point in the
    // class:
    inline point_type kdtree_get_pt(const size_t idx, const size_t
       dim) const
    {
        // return a point
    }

    // Optional bounding-box computation: return false to default to
       a
    // standard bbox computation loop.  Return true if the BBOX was
    // already computed by the class and returned in "bb" so it can
       be
    // avoided to redo it again.  Look at bb.size() to find out the
    // expected dimensionality (e.g. 2 or 3
    template <class BBOX>
    bool kdtree_get_bbox(BBOX& bb) const
    {
        bb[0].low = ...; bb[0].high = ...;  // 0th dimension limits
        bb[1].low = ...; bb[1].high = ...;  // 1st dimension limits
        // ...
        return true;
    }
};
```

7.2 Distance

A distance metric must be provided to measure separation of points in the space. L1, L2 and some other metrics are provided. Note, the units of "distance" depend on the metric. For example, when the L2 metric is used the distance is in units of length-squared.

7.3 Index

A `nanoflann` *index adaptor* provides the k-d tree query interface. `KDTreeSingleIndexAdaptor` provides one implementation. It is templated in terms of a *distance* type, a *dataset adaptor*, the dimensionality and the index type. It operates assuming the adapted dataset remains unchanged.

The `KDTreeSingleIndexDynamicAdaptor` is another which allows for points to be added or removed from the k-d tree. Points may be removed from the k-d tree without removing them from the dataset. When adding points, they are first added to the dataset and the adaptor is notified about the range of indices which are added.

The `nanoflann` library also defines an `KDTreeEigenMatrixAdaptor`. This class shears across both dataset adaptor and index adaptor design layers. It provides a dataset adaptor for `Eigen::Matrix` and then internally creates an maintains a `KDTreeSingleIndexAdaptor` using itself as the dataset adaptor.

7.4 Query methods

There are two primary query methods provided by `KDTree*Index*Adaptor` classes (and via the exposed `KDTreeEigenMatrixAdaptor::index` pointer to a `KDTreeSingleIndexAdaptor`).

```
Size knnSearch(
    const ElementType* query_point, const Size num_closest,
    AccessorType* out_indices, DistanceType* out_distances_sq) const;
```

```
Size radiusSearch(
    const ElementType* query_point, const DistanceType& radius,
    std::vector<std::pair<AccessorType, DistanceType>>& IndicesDists,
    const SearchParams&                                 searchParams)
        const;
```

The `knnSearch()` finds the k-nearest neighbors (`num_closest`) of the query point and `radiusSearch()` finds neighbors withing the given distance.

The `ElementType* query_point` provides a point in the space as a C array. The type `ElementType` is the type of the point coordinates. The `DistanceType` is the type in which distances between points is expressed. These are set through the distance metric type and by default they are the same type.

The `AccessorType` is that of the index of a point in the point cloud. The `SearchParams` are a small bundle, presumably to tune the k-d tree.

Both of these rely on `findNeighbors()` method.

```
template <typename RESULTSET>
bool findNeighbors(
    RESULTSET& result, const ElementType* query_point,
    const SearchParams& searchParams) const;
```

The `RESULTSET` passed to `findNeighbors()` may be a `KNNResultSet<DistanceType>` for `knnSearch()` or a `RadiusResultSet<DistanceType>` for `radiusSearch()`. The only reason to call this directly instead of through the high level methods is if the result set provides additional required information that they do not.