CMTool
Simple Tool for Clustering of Clusters

Kazuhiro Terao, kazuhiro@nevis.columbia.edu

June 3, 2014

Abstract

CMTool is a simple framework for clustering of clusters. In particular, it supports two kinds of clustering: (1) merging of multiple clusters in one plane and (2) pairing of a cluster from each wire plane. It makes a use of CPAN from ClusterRecoUtil toolkit to represent a 2D cluster. There is an interface to produce CPAN from a cluster data product that is analysis framework specific. This allows CMTool to be (almost) independent of analysis framework, and makes it available in both LArSoft and LArLight in particular. CMTool supports iterative approach for cluster merging and matching of clusters. The modulated design also decouples algorithm implementation from the rest of merging tasks such as looping over cluster pairs and keeping track of the process. This also makes it easy for anyone to develop his/her own algorithm easily and efficiently. This documentation currently only describes about (1) as (2) is still underdevelopment.

Contents

Chapter 1

`CMTool` Framework Basics

Despite its extreme simplicity let us call `CMTool` a framework, or FMWK. In this chapter we discuss the building blocks of `CMTool`.

1.1 External Dependency

The `CMTool` development effort is originated from EM shower reconstruction for MicroBooNE experiment, and it is an extension of existing tool set. The following is the brief list of packages or `C++` classes on which `CMTool` depends.

`ClusterRecoUtil`

- larreco/RecoAlg/ClusterRecoUtil in `LArSoft`
- UserDev/ClusterRecoUtil in `LArLight`

`GeometryUtilities`

- lardata/Utilities in `LArSoft`
- core/LArUtil in `LArLight`

`PxUtil`

- lardata/Utilities in `LArSoft`
- core/LArUtil in `LArLight`

These items are (or should be if not yet) described in a separate document. The rest of this section briefly covers what they are without going into a technical detail.

1.1.1 `ClusterRecoUtil`

Reconstruction parameters for EM shower 2D cluster involves a lot more variety of parameters than those held in a 2D cluster data product defined by a framework (`LArSoft` or `LArLight`). `ClusterRecoUtil` is a package that defines code for both calculation of and storage for those parameters. In particular, `CMTool` uses following `C++` classes from `ClusterRecoUtil`:

- `ClusterParamsAlg ...  (CPAN)`
  - `CPAN` is designed as both parameter computation algorithm and storage class, and is used along this design principle in `CMTool`. In particular, `CPAN` is created one per input 2D cluster.
- `cluster_param`
  - `cluster_param` is a utility storage class for important parameters computed by `CPAN`. A const reference can be retrieved from `CPAN` instance through `GetParams()` function call. Note, however, one should keep `CPAN` instance rather than `cluster_params` as a cluster parameter data set (see `CPAN` description above).
- `CRUException`
  - This is a very simple exception class derived from `std::exception`. It is used instead of `cet::exception` to avoid framework dependency. Only additional feature to the base class is that the argument string to the constructor is sent to standard output streem by overriden `std::exception::what()` function.

For details of all classes listed above, you should see Ref.[1].

1.1.2 `PxUtil`

Both `ClusterRecoUtil` and `CMTool` are designed to be independent of an analysis framework of user's choice (in particular `LArLight` and `LArSoft`). As mentioned in Sec.1.1.1, output of `CPAN` (i.e. cluster parameters) is stored with `cluster_params` instance. `PxUtil`, on the other hand, is used to define input data struct to `CPAN` (and hence `CMTool` and `ClusterRecoUtil` in general). `PxUtil` defines three data objects:

- `PxPoint`
  - represents 2D point and contains wire [cm], time [cm], and plane ID (unsigned char)
- `PxHit`
  - inhertis from `PxPoint` with an additional attribute to hold charge [ADC], representing 2D hit
- `PxLine`
  - represents 2D line and contains start and end coordinates, (wire [cm], time [cm]), with a plane ID (unsigned char)

It is a FMWK interface module's respnsibility to translate input data product into these format. In particular, this module must convert a vector of FMWK hit data product (= cluster) into a vector of `PxHit`. This is discussed in FMWK interface chapter (see Ch.2).

1.1.3 `GeometryUtilities`

`C++` class `GeometryUtilities` is a utility class that helps to retrieve detector geometry related information such as converting 3D point coordinate into a projected 2D coordinate on a given plane. `CMTool` uses `GeometryUtilities` to interpret geometry related variables in full extent. For development of any non-existing function that treats detector geometry information, it is encouraged to develop in `GeometryUtilities` (and certainly not in `CMTool`). Please see Ref.[2] for more detailed documentation.

1.2 FMWK Bulding Blocks

In this section we discuss two core buliding block classes: (1) `CBoolAlgoBase` and (2) `CBookKeeper`. Though we discuss these basic `C++` classes first, it might be easier to understand these classes after you read about `CMergeManager`, described in Sec.1.3, because there we discuss how `CMergeManager` works with practical usage of `CBoolAlgoBase` and `CBookKeeper`.

1.2.1 Book Keeping: `CBookKeeper`

`CBookKeeper` inherits from `std::vector<unsigned short>`, and hence it is a vector of integers by itself. The length of vector can be specified using a constructor argument.

This class is used to track how input clusters are merged/matched into an output. Here we focus on merging of input clusters for the simplicity sake. The design assumes one to perform a merging attempt of 2 clusters at a time. When one has many clusters to make such attempt on all possible pairs of clusters, it could be troublesome to perform book-keeping for which clusters are merged together. `CBookKeeper` is designed such that a user can just log two to-be-merged clusters' ID at each attempt, and it holds a summary log of all merged cluster pairs.

Here are technical details. The index of vector corresponds to the initial cluster ID numbers. If you have 8 clusters to be subject for merging, `CBookKeeper` instance should be initialized as length 8. Initially the values stored in the vector is same as their index numbers as shown in the top of Fig.1.2.1. One can re-set the state of `CBookKeeper` instance by calling the following function.

```
void Reset(unsigned short n=0)
```

When one finds 2 clusters to be merged together, (s)he can call `CBookKeeper` attribute function:

```
void CBookKeeper::Merge(unsigned short, unsigned short)
```

with corresponding clusters' IDs (i.e. index numbers). Figure 1.2.1 shows how the process works for calling `CBookKeeper::Merge()` function twice with arguments (1,2) and (2,5). As a result, unique output cluster IDs are reduced to 6 because 3 clusters are combined together.

You can see `CBookKeeper` instance allows to retrieve which output cluster (stored value) an input cluster (vector index) ended up with in an intuitive way: simply use `std::vector::at()` method. However, to create an output cluster, it is probably easier to ask the other way around: given output cluster ID, which input cluster(s) does it contain? `CBookKeeper` has a simple function to answer this question:

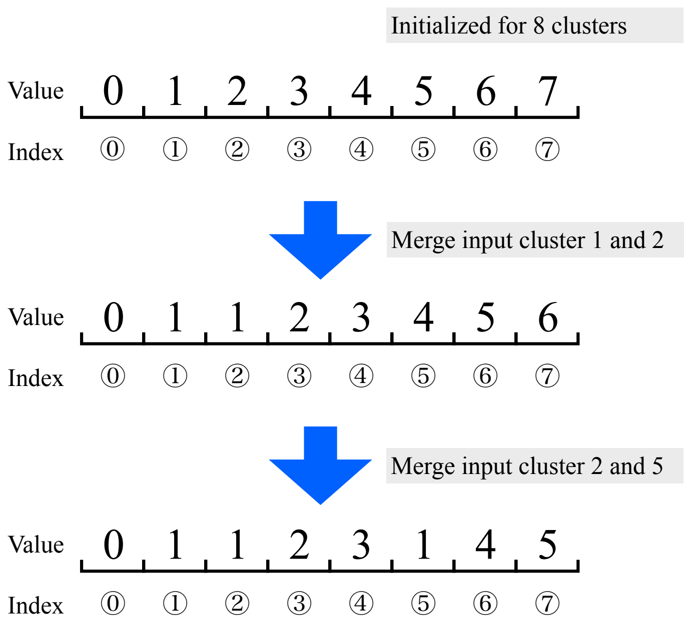

Figure 1.1: Example diagram showing how `CBookKeeper` keeps track of input and output cluster IDs. It shows `CBookKeeper` vector index (input cluster ID) and stored values (output cluster ID) for 8 input clusters (top), and how the stored values change as merging takes place. The middle shows the result of merging input cluster 1 and 2, and the bottom is for further merging 2 and 5.

```
std::vector<std::vector<unsigned short> > GetResult() const
```

which returns a vector with length same as number of output (i.e. merged) clusters. The value of this vector is `std::vector<unsigned short>` to contain possibly multiple input cluster indexes. For instance, the return from `CBookKeeper` shown in the bottom of Fig.1.2.1 would be a vector of length 6, and the index1 contains a vector of input cluster ID 1, 2, and 5 as they have been merged into output cluster ID = 1. Equivalent result can be obtained also by using

```
void PassResult(std::vector<std::vector<unsigned short> >&)
```

if that suits one's usage.

If you are interested in checking which input cluster(s) included in a particular output (merged) cluster, you can use:

```
std::vector<unsigned short> GetMergedSet(unsigned short)
```

which takes an output cluster index as an argument and returns a vector that contains a list of input clusters that consist the output.

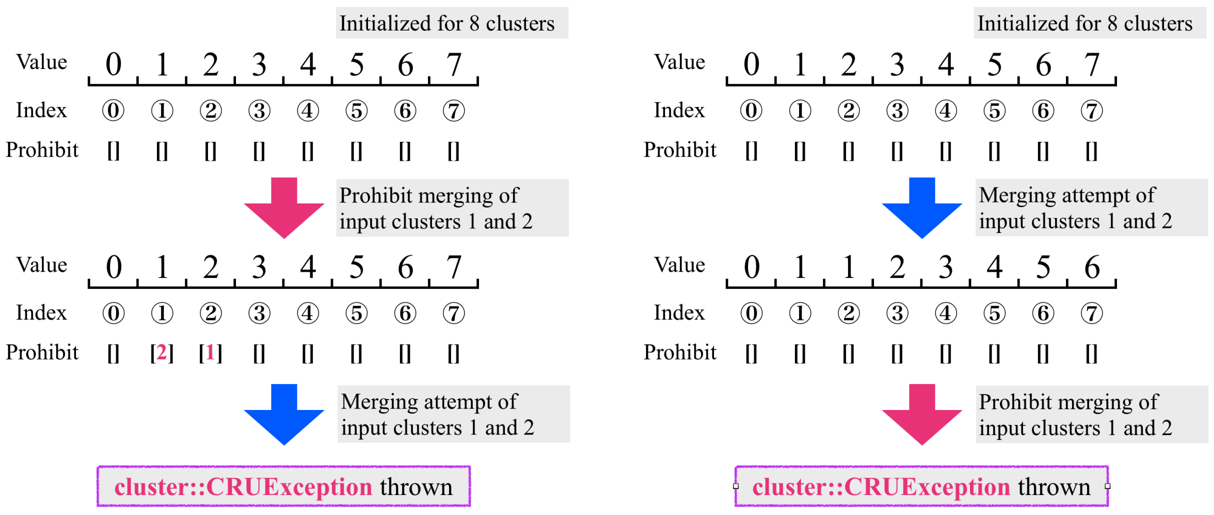
Initialized for 8 clusters
Value 0 1 2 3 4 5 6 7
Index ⓪ ① ② ③ ④ ⑤ ⑥ ⑦
Prohibit [] [] [] [] [] [] [] []
Prohibit merging of input clusters 1 and 2
Value 0 1 2 3 4 5 6 7
Index ⓪ ① ② ③ ④ ⑤ ⑥ ⑦
Prohibit [] [2] [1] [] [] [] [] []
Merging attempt of input clusters 1 and 2
cluster::CRUException thrown
Initialized for 8 clusters
Value 0 1 2 3 4 5 6 7
Index ⓪ ① ② ③ ④ ⑤ ⑥ ⑦
Prohibit [] [] [] [] [] [] [] []
Merging attempt of input clusters 1 and 2
Value 0 1 1 2 3 4 5 6
Index ⓪ ① ② ③ ④ ⑤ ⑥ ⑦
Prohibit [] [] [] [] [] [] [] []
Prohibit merging of input clusters 1 and 2
cluster::CRUException thrown

Figure 1.2: Example diagrams showing how `CBookKeeper` reacts to conflict `Merge()` and `ProhibitMerge()` function calls on the input cluster IDs. The left diagram shows `CRUException` thrown due to calling `Merge()` on a cluster pair that is registered as merge-prohibit pair. The right diagram shows a result from calling `ProhibitMerge()` on a cluster pair that is already merged.

Finally, `CMergeManager` can also prohibit merging of specific two cluster pair(s), and this is incoorporated by `CBookKeeper` as well. This is, again, to support user to log merging in a mindless manner. A user may specify specific 2 cluster pair should not be merged. Then (s)he can use the following function to register these 2 clusters for merge-prohibit list.

```
void ProhibitMerge(unsigned short, unsigned short)
```

The above function takes input cluster IDs (i.e. index numbers). Once 2 clusters are registered as merge-prohibit pair, attempting a function call `Merge()` will result in `CRUException` thrown. This is illustrated in the left of Fig.1.2. Another possible case that results in `CRUException` thrown would be calling `ProhibitMerge()` function on two input cluster IDs that are already merged. This is shown in the right hand side of Fig.1.2.

To avoid `CRUException`, one should register all cluster pairs to prohibit-merge list before trying any merge attempt. Upon attempting to call `Merge()`, then, one can use the function:

```
bool MergeAllowed(unsigned short, unsigned short)
```

to check if a cluster pair is registered to merge-prohibit list. Calling this function prior to evaluation of merging (i.e. running some computation algorithm to decide whether to merge or not) may save some computation time and help the overall speed of the program (this is implemented in `CMergeManager`).

1.2.2 Algorithm Base: `CBoolAlgoBase`

`CBoolAlgoBase` is a simple `C++` template class and also a base class for an algorithm used for cluster merging and matching by `CMergeManager` and `CMatchManager` application classes. In this and later sections, we use the term `CMAlgo` to refer to an algorithm class derived from `CBoolAlgoBase`.

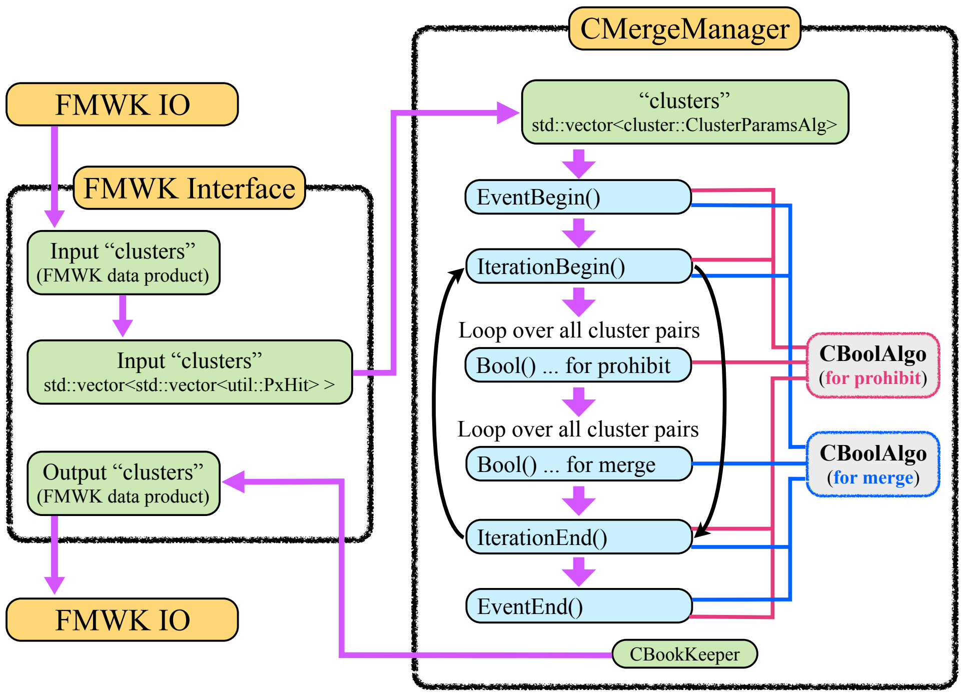
CMergeManager
FMWK IO
FMWK Interface
Input "clusters" (FMWK data product)
Input "clusters" std::vector<std::vector<util::PxHit> >
Output "clusters" (FMWK data product)
FMWK IO
"clusters" std::vector<cluster::ClusterParamsAlg>
EventBegin()
IterationBegin()
Loop over all cluster pairs
Bool() ... for prohibit
Loop over all cluster pairs
Bool() ... for merge
IterationEnd()
EventEnd()
CBookKeeper
CBoolAlgo (for prohibit)
CBoolAlgo (for merge)

Figure 1.3: The data processing stream of `CMergeManager` framework. The purple allow shows the ordered process flow. It requires an analysis framework interface to convert framework dependent cluster data produdct into a vector of `PxHit` and hand over to `CMergeManager` application. Internally `CMergeManager` calls various members of `CBoolAlgoBase` instances: one for merging and another for prohibiting. There is a configuration in `CMergeManager` to perform iterative approach, which repeats internal loop over possible cluster pairs until no more newly merged cluster is found in the output. The merged cluster groups (i.e. output) can be obtained through `CBookKeeper` instance by an analysis framework interface to form an appropriate output data product.

The idea is very simple: given two clusters, it is expected to return a boolean for decision making from the following function.

```
virtual bool Bool(const cluster::ClusterParamsAlg &, const cluster::ClusterParam
```

As you can see, "clusters" are handed over in terms of CPAN instance. In CMergeManager, the cluster merging application class, for instance, this is used for making merge decision as well as prohibit-merge decision (i.e. CMergeManager holds two CBoolAlgoBase inherit class instance, one for merging and another for prohibiting). An algorithm developpers should write his/her own C++ algorithm class by inheriting from CBoolAlgoBase. In the derived class, the Bool() function implementation should be over-riden. There is a script to generate a template (i.e. empty) CBoolAlgoBase inherit class and everyone is encouraged to use it for starting a new algorithm class. These are:

- For LArSoft

  ```
  > python $MRB_TOP/srcs/larreco/RecoAlg/CMTool/mac/gen_cmalgo.py NAME
  ```

- For LArLight

  ```
  > python $MAKE_TOP_DIR/UserDev/ClusterStudy/bin/gen_cmalgo NAME
  ```

where NAME should be replaced by the new algorithm's name. Running above command results in creating CMAlgoNAME.h and CMAlgoNAME.cxx in the appropriate directory. Then you can proceed to build following the framework's guidance.

As mentioned, CBoolAlgoBase is operated by CMergeManager and CMatchManager applications. For a given event, there is an internal loop of calling Bool() on all possible cluster pairs. To provide flexibility in algorithm design, those applications also call other functions of CBoolAlgoBase in its processing.

The following is a list of such CBoolAlgoBase functions.

```
virtual void EventBegin(const std::vector<cluster::ClusterParamsAlg>&)
```

- Called at the beginning of an event, prior to looping over cluster pairs. The provided function argument is a vector of CPAN for all clusters subject for merging. This is where computation that requires the whole, initial cluster sets should be done.

```
virtual void IterationBegin(const std::vector<cluster::ClusterParamsAlg>&)
```

- Called at the beginning of an each loop over cluster pairs. Note CMergeManager and CMatchManager can have multiple iteration of looping over possible cluster pairs if configured to do so. For instance CMergeManager can take iterate over multiple loops till no more to-be-merged cluster pair is found (see Sec.1.3 and Sec.1.4 for details). The argument of this function is an output cluster sets, in terms of CPAN instance, from the previous loop. In case of the 1st loop, the provided cluster sets is identical to those provided in the argument of EventBegin() function.

```
virtual void IterationEnd()
```

- Called at the end of each merge or match attempt loop over all cluster pairs, to clean up or summarize information.

`virtual void EventEnd()`

- Called at the end of each event, after all iterations over cluster pairs are done.

Further, there are two functions that are not yet implemented in `CMergeManager` nor `CMatchManager` but ideas exist. Your input on how these should be implemented is welcome and very much appreciated.

`virtual void Reset()`

- Meant to clear some member variables. But one can do that within `IterationEnd()` and/or `EventEnd()`, so it is not clear where this should be called. Possibly at the end of the whole event processing.

`virtual void Report()`

- Meant to support some standardized `std::cout` and `std::cerr` report from an algorithm instance. `CMergeManager` and `CMatchManager` have ability to control such text output for different levels, and there is a hope this `Report()` can be also included there.

Tips for writing `CMAlgo` First of all, you are encouraged to use `gen_cmalgo` script as described earlier. Another thing to note is to reduce a dependency on analysis framework. This makes it easier to port your code between different frameworks.

1.3 `CMergeManager`: Cluster Merging Tool

`CMergeManager` is a `C++` class that performs merging of 2D clusters, and is an application of `CMAlgo` algorithm classes that are inherit from `CBoolAlgoBase`, the algorithm base class. In a few sentences, `CMergeManager` perform merging by looping over all possible 2D cluster pairs among the given set of clusters. For each pair, `Bool()` function of both merging and prohibit `CMAlgo` are called to inspect possible 2D cluster merging. It uses `CBookKeeper` to keep track of which pair of clusters are merged. For details of `CBoolAlgoBase` and `CBookKeeper`, see Sec.1.2.2 and Sec.1.2.1 respectively. The overall process stream diagram is shown in Fig.1.3 which we keep referring in this section.

1.3.1 Why `CMergeManager`?

Why using `CMergeManager`? Why not just writing a program that does everything from scratch? Be my guest and write your code! That might inspire a better design or strategy toward merging. That being said, there are a few reasons to use `CMergeManager`.

Task Outside Merging Decision: `CMergeManager` The core part of merging is to compare 2 clusters and make a decision to merge or not. Outside this process, however, there is a non-negligible overburden of running such algorithm such as (1) keeping track of which cluster pairs are merged and (2) optimizing the loop process of cluster pair comparison. `CMergeManager` does those tedious jobs while out-sourcing the decision making to an independent `C++` algorithm class that inherit from `CBoolAlgoBase`.

Easy Development via Modulated Design: Because `CMergeManager` outsource an independent algorithm class to perform merging of a cluster pair, it supports modulated code design. Accordingly each `CMAlgo` class becomes a pure algorithm class. This makes a code development very easy because a new algorithm can be written without re-writing the I/O or book-keeping part. Also it makes life easy to compare different algorithms: you can simple change which algorithm to be attached to `CMergeManager` at run-time, and this avoids re-compilation or re-writing of code.

X-Framework Implementation `CMergeManager` does not use data product type specific to an analysis framework (for instance `LArLight` or `LArSoft`), so it can be ported easily among different frameworks. That means `CMAlgo` developed by one person in one analysis framework can be shared among other frameworks (assuming an algorithm developer follows x-framework design pattern).

1.3.2 How Does It Work?

The core part of decision making is simple: provided 2 clusters, decide whether they should be merged or not. This is done through `CMAlgo`::Bool() function call:

```
virtual bool Bool( const cluster::ClusterParamsAlg &, const cluster::ClusterParamsAlg
```

If the return is `true`, two clusters are merged. Else not. This is ilustrated in Fig.1.4.

`CMergeManager` takes care of creating a loop over all possible pairs of 2D clusters and calling the above function of `CMAlgo` for merging. In the process, book-keeping of which pairs to be merged is recorded using `CBookKeeper`.

Let us consider a case of two merged pairs that shares the same input cluster ID. As described in Sec.1.2.1, `CBookKeeper` combines two pairs of 2D clusters in such case (see Fig.1.2.1). This could be problematic sometimes. For instance, consider an example `CMAlgo` that combines two clusters if their start points are very close and the angle between two clusters are within 40 degrees. Further assume we have 3 input clusters that share the start point but have different angles at 20, 40, and -20 degrees. For convention these have input cluster ID 0, 1, and 2 respectively. This example algorithm would conclude pairs (0,1) and (0,2) but not for (1,2). But because merged pairs share cluster ID 0, it ends up all 3 clusters being merged. This is shown in the right-top ilustration of Fig.1.5.

This could be a problem, or may not be: this can be only answered by the algorithm author. The way to avoid this in `CMergeManager` is to implement another `CMAlgo` as *merge-prohibit*, or simple *prohibit* algorithm. When prohibit algorithm is provided, `CMergeManager` first run

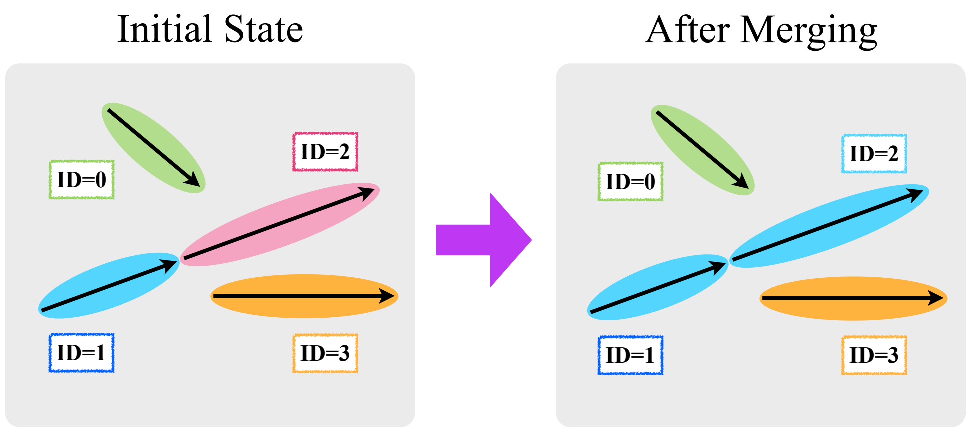

Figure 1.4: Example of 2D clusters in a plane before and after merging attempt. Each oval represents a cluster with cluster 2D axis shown in a black solid arrow. The direction of an arrow represents the direction of a cluster development. Each cluster is labeled with an input cluster ID number while the oval color represents distinct output cluster group. Left: initial state with 4 input clusters. Right: example output after running a merge loop over cluster pairs.

over all cluster pairs and execute `Bool()` function to find all cluster pairs that are prohibited to merge. Then `CMergeManager` run over the same set of cluster pairs except for those are prohibited to be merged. The right-bottom ilustration of Fig.1.5 shows how prohibit algorithm affects the result.

The prohibit rule is also inherited to merged cluster pairs. For instance, consider input clusters 0, 1 and 2. The pair (1,2) is prohibited to merge, and (0,1) is found to be merged. Then the merged cluster, (0,1) cannot be merged with 2. That means even if a merge algorithm finds (0,2) to be merged, it cannot be because the prohibit rule is inherited from cluster 1 when 0 and 1 aremerged. This feature brings up a question: what determines 0 to be merged with 1 rather than 2? If the merge algorithm is executed on the pair (0,2) first, then the result would have been different. So what determines a different outcome is an order of cluster pairs in the loop of merging inspection. There are different ordering options available in `CMergeManager`. Currently available options are defined as `enum` type in `CMergeManager` class, `CMergePriority_t` which include following values.

`kIndex`

– From the smaller to larger input cluster ID, basically following input cluster set

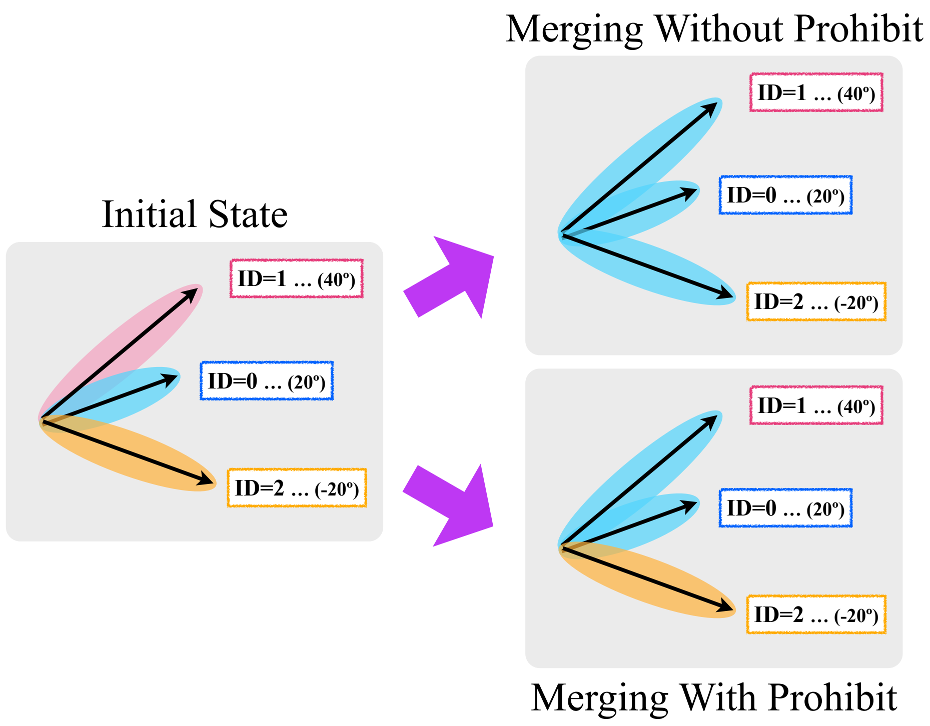

Figure 1.5: Example of how prohibit algorithm works. Each oval and solid arrow represents a 2D cluster with its direction while the color represents distinct output (merged) cluster. Input cluster index (ID) is shown in the label with the cluster 2D angle. Right-top shows an output of running `CMAlgo` that merge clusters with angle difference within 40 degrees. Right-bottom shows an output of the same `CMAlgo` for merging but with another `CMAlgo` that prohibits merging of clusters with angle difference larger than 40 degrees.

ordering

`kPolyArea`

- From the clusters with smaller polygon area to the larger

`kChargeSum`

- From the clusters with smaller charge sum to the larrger

`kNHits`

- From the clusters with smaller number of hits to the larger

The ordering parameters including all listed above are taken from `cluster_params` member attributes. If more useful parameter is found, one should make a request to make it available. It is very simple to implement in `CMergeManager`.

In summary, CMergeManager uses one CMAlgo instance for merging and another for merge-prohibit. A possible ambiguity of which cluster pairs to be merged in the presence of prohibited pairs is resolved by defining a priority for cluster pairs in the merging attempt.

1.3.3 Process Flow for Cluster Merging

An overall process flow diagram is already shown in Fig.1.3, so here we describe slightly more details in texts by breaking down into 5 blocks.

1. Event Preparation

   CMergeManager receives an input cluster in terms of a set of PxHit (see Sec.1.3.4). Then it creates CPAN instance for each cluster, and calls CMAlgo::EventBegin() for merging and prohibit algorithms.

2. Iterative Merge Operation

   A loop over cluster pairs for prohibit (if provided) and merge CMAlgo can be configured to repeat until no newly merged cluster is found. This is shown in Fig.1.6. If a user does not configure to choose iterative approach, iteration will not happen even if there is a newly merged cluster in the output.

3. Prohibit CMAlgo Loop Over Cluster Pairs

   See Fig.1.6. Only occurs if prohibit algorithm is provided by a user (optional). Records which cluster pair to be prohibited for merging using CBookKeeper.

4. Prohibit CMAlgo Loop Over Cluster Pairs

   See Fig.1.6. The algorithm's Bool() is not executed for a subject pair of clusters if (a) they are prohibited for merging or (b) already inspected in the previous process and not merged (i.e. result should not change). This is to save computation time.

5. Merge Clusters

   Based on the outcome of merge/prohibit CMAlgo algorithms, actual merging is performed at this step. It creates a new set of CPAN for merged clusters. For clusters that are not merged, CPAN instance is copied from input to save CPAN computation time.

1.3.4 Setting an Input 2D Clusters

Input to CMergeManager is a set of clusters where each cluster is represented as a vector of hits, or more precisely std::vector<util::PxHit>. This input can be set by the following function:

```
void SetClusters(const std::vector<std::vector<util::PxHit> > &)
```

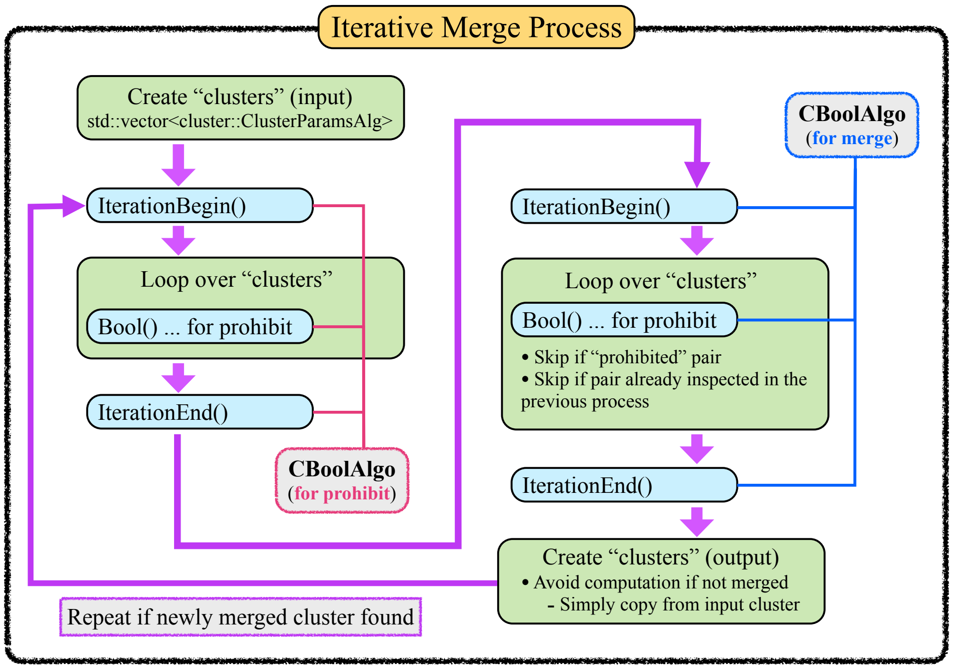
Iterative Merge Process
Create "clusters" (input)
std::vector<cluster::ClusterParamsAlg>
IterationBegin()
Loop over "clusters"
Bool() ... for prohibit
IterationEnd()
CBoolAlgo
(for prohibit)
Repeat if newly merged cluster found
CBoolAlgo
(for merge)
IterationBegin()
Loop over "clusters"
Bool() ... for prohibit
• Skip if "prohibited" pair
• Skip if pair already inspected in the previous process
IterationEnd()
Create "clusters" (output)
• Avoid computation if not merged
- Simply copy from input cluster

Figure 1.6: Illustration of how iterative merge approach is processed. Internally a loop over cluster pairs for prohibit and merge algorithm are repeated if (a) user configured to use an iterative approach and (b) newly merged cluster is found.

where the argument is a vector of 2D clusters. When this function is called, `CMergeManager` internally calls

```
void Reset()
```

which initialize previously stored input clusters, output clusters and `CBookKeeper` instance. Then it creates `CPAN` instance per input 2D cluster using a set of `PxHit`. This completes setting (or initiating) the input 2D cluster sets to be used for internal processing.

1.3.5 Setting Merge/Prohibit `CMAlgo` Instances

`CMAlgo` instance is necessary for merging and optional for prohibit. Call the function,

```
void AddMergeAlg(CBoolAlgoBase* algo)
```

to set `CMAlgo` for merging, and

```
void AddSeparateAlgo(CBoolAlgoBase* algo)
```

to set `CMAlgo` for merge-prohibit.

1.3.6 Setting Run Configuration

There are only 3 run options to configure for `CMergeManager`.

Verbosity: There are 5 verbosity levels a user can configure as listed below:

`kNone`

– No report from `CMergeManager`

`kPerEvent`

– Report made at the end of even processing

`kPerIteration`

– Report made per merge iteration (see Sec.1.3.3). This could be quite verbose.

`kPerMerging`

– Report made per `CMAlgo`::Bool() execution (see Sec.1.3.3). This is even more verbose than `kPerIteration`.

These are `enum` type, `CMergeMSGLevel_t`, defined under `CMergeManager` class. The verbosity level can be set by the function:

```
void DebugMode(CMergeMSGLevel_t level)
```

Iterative Approach: As described in Sec.1.3.3, `CMergeManager` can be configured to attempt iterative approach for merging. This can be enabled by the following function call.

```
void MergeTillConverge(bool)
```

Providing `true` as an argument will enable this option. Disabled if `false` is provided.

Merge Priority: As described in Sec.1.3.2, ordering of cluster pairs for merging inspection matters to the result. Options are also described in the same section. This can be set by the following function.

```
void SetMergePriority(CMergePriority_t level)
```

1.3.7 Execute Merging

To execute merging, simply call

```
void Process()
```

1.3.8 Obtaining Result

There are two kinds of results that may be useful for a user. First is a resulting (i.e. merged) set of clusters in terms of `CPAN`. This is useful for further processing as various cluster parameters are already computed as a part of `CPAN`. You can execute

```
  const std::vector<cluster::ClusterParamsAlg>& GetClusters() const
```

to obtain such result set.

Second is a mapping of input cluster IDs (i.e. index) to the output (merged clusters). As described in Sec.1.2.1, this can be easily obtained through `CBookKeeper` instance. You can obtain the `CBookKeeper` instance held by `CMergeManager` through the following function.

```
  const CBookKeeper& GetBookKeeper() const
```

1.4 `CMatchManager`: Cluster Matching Tool

Tool not introduced yet in `LArSoft`. Will be documented when this is done.

Chapter 2

Analysis FMWK Interface

`CMTool` consists of a collection of simple `C++` and is (almost) independent of an analysis framework. Inparticular it is available in both `LArLight` and `LArSoft`. To support such structure, however, an interface to each framework is necessary. This chapter briefly describes an interface to supported analysis frameworks.

2.1 `LArSoft` Interface

This section describes the `LArSoft` interface, called `ClusterMergeHelper`, and an example data producer module, `SimplerClusterMerger`.

2.1.1 `ClusterMergeHelper`

This is a simple wrapper class for `CMergeManager` that can be found under:

```
larreco/RecoAlg/ClusterMergeHelper.h
```

It holds `CMergeManager` instance which a user can retrieve through

```
CMergeManager& GetManager()
```

and configure as (s)he wish.

There are two important wrapper functions that can take `LArSoft` data product as input, convert into a collection of `PxHit` and feed into `CMergeManager` instance. First one takes a vector of clusters where each cluster is represented as `std::vector<art::Ptr<recob::Hit> >`.

```
void SetClusters(const std::vector<std::vector<art::Ptr> > >&)
```

The second one takes `art::Event` reference with the producer module name for an input `recob::Cluster` data product.

```
void SetClusters(const art::Event&, const std::string&)
```

Finally, there is a function that can produce output data product and store in `art::Event`:

```
void AppendResult( art::EDProducer&,
                   art::Event&,
                   std::vector<recob::Cluster>&,
                   art::Assns<recob::Cluster,recob::Hit>& )
```

where arguments are producer module's object reference, `art::Event` reference that should have been provided to the module, output data products' reference (i.e. vector of `recob::Cluster` and `art::Assns`).

2.1.2 `SimpleClusterMerger`

This is an `art::EDProducer` inherit class, a data product producer module in `LArSoft`, that uses `ClusterMergeHelper` to perform cluster merging with some existing merge and prohibit algorithms. It can be found in here:

```
larreco/ClusterFinder/SimpleClusterMerger_module.cc
```

The code is commented fairly alright, so one should see the code for details of what is done. Basically all operations done in the module are functions already described in this document by this point.

2.2 `LArLight` Interface

Unfortunately there is no equivalent class of `ClusterMergeHelper` from `LArSoft` in `LArLight`. Instead there is a `ClusterMerger` analysis unit class (i.e. inherits from `ana_processor`, see Ref.[3]). There is a viewer application to compare input clusters, output (i.e. merged) clusters, and `MCShower` data products to debug merging/prohibit algorithms. These are described in this section.

2.2.1 `ClusterMerger`

As mentioned already, this is an analysis unit in `LArLight`. The usage is described in Ref.[3]. By default, however, it does not save output cluster data product. This option (i.e. saving the output) can be enabled by the function:

```
void SaveOutputCluster(bool)
```

with a boolean value `true` as an argument. It can be disabled (though that is default) by providing `false`. This module is mainly used to run merging on the fly by `MergeViewer`, the class that runs merging algorithm on the fly and show the result on the display. That is why the saving out output data product is turned off by default.

2.2.2 Viewer Application Classes

This will be documented when the author finds time as it can be lengthy. It is fully functional and used intensively by a few users. Contact the author if interested in.

Chapter 3

FAQ

Please ask questions, and I will add those questions I answered in here.

Bibliography

[1] Cluster reconstruction utility for shower 2d clusters. *MicroBooNE DocDB*, Unknown DocDB entry (not made yet).

[2] Geometry utility helper tool. *MicroBooNE DocDB*, Unknown DocDB entry (not made yet).

[3] Simple analysis framework for larsoft. *MicroBooNE DocDB*, http://microboone-docdb.fnal.gov:8080/cgi-bin/ShowDocument?docid=3183.