LArSoft algorithms and services

Gianluca Petrillo

University of Rochester/Fermilab

art/LArSoft course
Fermilab, August 3–7, 2015

Outline

Outline

What is LArSoft

LArSoft

A toolkit to facilitate simulation, reconstruction and analysis of events from liquid-argon TPC-based detectors.

Stress on:

toolkit a set of instruments: data structures, algorithms, services

facilitate includes many aspects:

- easy to use: recurrent interfaces, documentation
- easy to maintain: enables quick testing
- not tied to a specific execution environment
- fast development cycle (change/compile/test/run)

detectors $\Rightarrow$ generic enough to work with any liquid argon TPC

LArSoft content is contributed by: *you*!

You have a determinant role in helping it reach these goals.

Outline

LArSoft services

Most of LArSoft services are factorized into two parts:

service provider actually providing services

It can be used stand-alone (without *art* framework)

$\Rightarrow$ e.g. for a minimalist test environment

- configuring it correctly and keeping it updated with the current event may be complex

***art* service** interfacing the provider with the *art* framework

- initializes the provider correctly
- keeps it up to date with the current event and run

This factorization model allows service providers to be tested without pulling in *art* framework, and to be used in *art*-unaware environments.

Using service providers

Most of the user code should interact only with the provider interface:

1. grab the service provider (for *art* services that have one):

```
#include "art/Framework/Services/Registry/ServiceHandle.h"
#include "Geometry/GeometryCore.h"

void MyModule::analyze(art::Event const& event) {
  geo::GeometryCore const& geom
    = *art::ServiceHandle<geo::Geometry>();
  /* ... */
}
```

Listing 1: Obtaining the geometry service provider, `geo::GeometryCore`

2. use it, move it around, pass it to algorithms:

```
for (geo::PlaneID const& planeID: geom.IteratePlaneIDs()) {
  /* ... */
}
```

Listing 2: Using geometry provider to iterate through all wire planes in the detector

There may be additional ways to access the service provider functionalities, depending on the specific service.

LArSoft services: geometry

Provides a description of the physical and readout aspects of the detector:

TPC structure physical characteristics of cryostats, TPCs, wire planes and wires and their relations

optical detector structure

readout channel mappings

- TPCs (e.g. readout channels to physical wires)
- optical readout channels with photodetectors

auxiliary detectors (e.g. plastic muon detectors)

(being moved into its own service `AuxDetGeometry`*)*

art service	`geo::Geometry`	`larcore/Geometry/`
service provider	`geo::GeometryCore`	`larcore/Geometry/`

```
geo::GeometryCore const& geom = *(art::ServiceHandle<geo::Geometry>());
```

LArSoft services: physical properties

Two services provide information about physical properties:

`DetectorProperties` pertaining mostly readout

- readout sampling rates, readout window size
- conversion between readout ticks and times

`LArProperties` about the liquid argon environment

- drift velocity, $dQ/dx \rightarrow dE/dx$
- electron "lifetime", radiation length, argon temperature...

art service	`util::DetectorProperties`	`lardata/Utilities/`
service provider	not available yet	
art service	`util::LArProperties`	`lardata/Utilities/`
service provider	not available yet	

```
util::DetectorProperties const& detProp
  = &*art::ServiceHandle<util::DetectorProperties>();
util::LArProperties const& detProp
  = &*art::ServiceHandle<util::LArProperties>();
```

LArSoft services: database, timing, etc.

Other services deal with different aspects:

`DatabaseUtil` sets up database connections for other services and algorithms; you typically don't interact with it directly

`TimeService` provides a coherent view of times in the event related to the beam trigger

`LArFFT` facilitates the application of Fourier transform in simulation of TPC signals and their calibration (e.g. noise reduction)

art service	`util::TimeService`	`lardata/Utilities/`
service provider	`util::SimpleTimeService`	`lardata/Utilities/`
art service	`util::LArFFT`	`lardata/Utilities/`
service provider	not available yet	

```
util::SimpleTimeService const& timeSrv
  = &*art::ServiceHandle<util::TimeService>(); // automatic conversion
util::LArFFT const& FFT
  = &*art::ServiceHandle<util::LArFFT>();
```

LArSoft services: simulation

Some services are mostly used in physics simulation:

`LArVoxelCalculator` helps with computations in the virtual grid GEANT splits the detector volume into ("voxels")

`LArG4Parameters` stores the parameters GEANT is configured with

`PhotonVisibilityService` describes the simulation of the transportation of photons to the optical detectors

`SpaceCharge` describes the effect of distortions of TPC's electric field

LArSoft services: channel quality

A service is going to be introduced soon:

`ChannelFilterService` delivers information about quality of TPC channels: bad, noisy... even good ones.

art service	`filter::ChannelFilterService`	`larevt/Filters/`
service provider	`filter::ChannelFilterProvider`	`larevt/Filters/`

```
filter::ChannelFilterProvider> const& chanFilt
  = art::ServiceHandle<filter::ChannelFilterService>()->GetProvider();
```

Abstract service interfaces

This service implements an abstract service interface pattern:

- both `ChannelFilterService` and `ChannelFilterProvider` only describe *what* can be asked: they only expose an interface
- each experiment chooses which specific implementation to use for the service, to specify *how* to answer the requests

A typical service configuration will include the implementation type:

```
services.ChannelFilterService: {
  service_provider: SimpleChannelFilterService

  # list of bad channels:
  BadChannels:   [ 22, 65, 237, 307, 308, 309, 310, 311, 410, 412, ... ]
  # ...
}
```

Listing 3: Possible configuration of `ChannelFilterService` for ArgoNeuT

where `filters::SimpleChannelFilterService` is an implementation of `filter::ChannelFilterService` interface.

Outline

Algorithms

An algorithm is a piece of code that:

- performs one single task
  so that it can be a component of many execution paths
- must be thoroughly documented:
  what it needs, what it does (with a bit of how and why), *its assumptions*, what it produces, how to configure it
- needs a complete test
  so that it's easy to detect if it gets (or is from the beginning) broken

In addition, a LArSoft algorithm:

- should be able to perform its task using only:
  1. LArSoft data products and their associations
  2. services provided by LArSoft
  3. output from other algorithms
- similarly, it should produce only:
  1. LArSoft data products and their associations
     - (ok, ok, also histograms)

Algorithms

A LArSoft algorithm:

- is implemented as a self-standing class:
  one header file, one implementation file, minimal dependencies

```
#ifndef CLUSTER_MYCLUSTERINGALG_H
#define CLUSTER_MYCLUSTERINGALG_H

namespace cluster {
  class MyClusteringAlg {
      public:
    /* ... */
  }; // class MyClusteringAlg
} // namespace cluster

#endif // CLUSTER_MYCLUSTERINGALG_H
```

Listing 4: `Cluster/MyClusteringAlg.h`*: class header file*

```
#include "Cluster/MyClusteringAlg.h"

cluster::MyClusteringAlg::MyClusteringAlg(fhicl::ParameterSet const&) {
  /* ... */
} // cluster::MyClusteringAlg::MyClusteringAlg()
```

Listing 5: `Cluster/MyClusteringAlg.cxx`*: class implementation file*

Algorithms

A LArSoft algorithm:

- is implemented as a self-standing class
- has constructor configured by FHiCL parameter set

```
#include "fhiclcpp/ParameterSet.h"
// ...
  class MyClusteringAlg {
      public:
    explicit MyClusteringAlg(fhicl::ParameterSet const& pset);
  }; // class MyClusteringAlg
```

Listing 6: MyClusteringAlg class: constructor

Algorithms

A LArSoft algorithm:

- is implemented as a self-standing class
- has constructor configured by FHiCL parameter set
- has a setup phase to get service providers (e.g. detector geometry)

```
#include "Geometry/GeometryCore.h"
// ...
  class MyClusteringAlg {
      public:
    explicit MyClusteringAlg(fhicl::ParameterSet const& pset);

    void Setup(geo::GeometryCore const* geo);
  }; // class MyClusteringAlg
```

Listing 7: MyClusteringAlg class: setup methods

Algorithms

A LArSoft algorithm:

- is implemented as a self-standing class
- has constructor configured by FHiCL parameter set
- has a setup phase to get service providers
- has an input phase, a running phase, an output phase

```
#include "RecoBase/Hit.h"
#include "RecoBase/Cluster.h"
#include <vector>
// ...
  class MyClusteringAlg {
      public:
    explicit MyClusteringAlg(fhicl::ParameterSet const& pset);

    void Setup(geo::GeometryCore const* geo);

    void SetHits(std::vector<recob::Hit> const& hits);
    void Run();
    void GetClusters(std::vector<recob::Cluster>& clusters);
  }; // class MyClusteringAlg
```

Listing 8: MyClusteringAlg class: algorithm methods

Algorithms

A LArSoft algorithm:

- is implemented as a self-standing class
- has constructor configured by FHiCL parameter set
- has a setup phase to get service providers
- has an input phase, a running phase, an output phase (may be combined)

```
#include "RecoBase/Hit.h"
#include "RecoBase/Cluster.h"
#include <vector>
// ...
  class MyClusteringAlg {
      public:
    explicit MyClusteringAlg(fhicl::ParameterSet const& pset);

    void Setup(geo::GeometryCore const* geo);

    std::vector<recob::Cluster> ComputeClusters
      (std::vector<recob::Hit> const& hits);
  }; // class MyClusteringAlg
```

Listing 9: MyClusteringAlg class: algorithm methods

Algorithms

A LArSoft algorithm:

- is implemented as a self-standing class
- has constructor configured by FHiCL parameter set
- has a setup phase to get service providers
- has an input phase, a running phase, an output phase
- uses LArSoft data structures

```
#include "RecoBase/Hit.h"
#include "RecoBase/Cluster.h"
#include <vector>
// ...
  class MyClusteringAlg {
      public:
    explicit MyClusteringAlg(fhicl::ParameterSet const& pset);

    void Setup(geo::GeometryCore const* geo);

    std::vector<recob::Cluster> ComputeClusters
      (std::vector<recob::Hit> const& hits);
  }; // class MyClusteringAlg
```

Listing 10: MyClusteringAlg uses LArSoft data products and services

Algorithms

A LArSoft algorithm:

- is implemented as a self-standing class
- has constructor configured by FHiCL parameter set
- has a setup phase to get service providers
- has an input phase, a running phase, an output phase
- uses LArSoft data structures
- **comes with documentation** (and a test!)

```
/**
 * @brief My clustering algorithm
 *
 * Designed for events that have this and that. More information at:
 * http://some.public.enough.place/or/inspire/document
 *
 * Configuration parameters
 * ========================
 *
 * - *Threshold* (real, default: 2): minimum threshold [MeV]
 */
class MyClusteringAlg {
  float Threshold; ///< the threshold [MeV]
  /* ... */
```

Code factorization

- none of this depends on the framework (*art* or otherwise)
  - no `art::Handle` s, no `art::ServiceHandle` s ...
- the algorithm can be run in a bare test environment:

```
int main() {
  fhicl::ParameterSet algo_config;
  fhicl::make_ParameterSet("Threshold: 35.5", algo_config);
  MyClusteringAlg algo(algo_config);
  std::vector<recob::Hit> hits;
  // add some input...
  hits.emplace_back(1 /* channel */, 25 /* start tick */, ...);
  std::vector<recob::Cluster> clusters = algo.ComputeClusters(hits);
  std::cout << clusters.size() << " clusters!" << std::endl;
  return clusters.empty()? 1: 0; // non-zero means error
} // main()
```

Listing 12: MyClusteringAlg test stub (pretending no Setup() needed)

$\Rightarrow$ the algorithm needs a broker to communicate with the framework

- modules are *art*'s brokers

This is another facet of the factorization model.

Modules (I)

To use a LArSoft algorithm in *art*, a module is needed:

```
#include "Clusters/MyClusteringAlg.h"

/// Documentation about the configuration goes here
class MyClustering: public art::EDProducer {
  std::unique_ptr<cluster::MyClusteringAlg> algo;
  art::InputTag inputHits;
    public:
  explicit MyClustering(fhicl::ParameterSet const& pset);

  virtual void produce(art::Event const& event) override;
}; // class MyClustering
```

Listing 13: `MyClustering_module.cc:MyClustering` *declaration*

```
MyClustering::MyClustering(fhicl::ParameterSet const& pset)
  : algo(new MyClusteringAlg(pset.get<fhicl::ParameterSet>("Params")))
  , inputHits(pset.get<art::InputTag>("InputHit"))
{
  produces<std::vector<recob::Cluster>>();
}
```

Listing 14: `MyClustering_module.cc:MyClustering` *constructor*

Modules (II)

A possible implementation of the module `produce()`:

```
void MyClustering::produce(art::Event const& event) {
  // the services for setup
  geo::GeometryCore const* geom
    = art::ServiceHandle<geo::Geometry>()->GetProviderPtr();

  algo->Setup(geom);

  // the input data
  art::ValidHandle<std::vector<recob::Hit>> hits
    = event.getValidHandle<std::vector<recob::Hit>>(inputHits);
  // the output data
  std::unique_ptr<std::vector<recob::Cluster>> clusters
    = new std::vector<recob::Cluster>;

  *clusters = algo->ComputeClusters(hits);

  event.put(std::move(clusters));

} // MyClustering::produce()
```

Listing 15: `MyClustering_module.cc: MyClustering::produce()`

In this case, one should also produce hit/cluster associations.

LArSoft: physics event generation

Many generator modules are present. The most commonly used are:

`SingleGen` generates a single particle

`GenieGen` generates neutrino interactions with GENIE generator

`CosmicGen` generates cosmic rays with CRY generator

`TextFileGen` reads events already generated in HEPEVT format

`RadioGen` generates events from “radiological” noise

LArSoft: detector simulation

`LArG4` module performs the simulation of the TPC detectors:

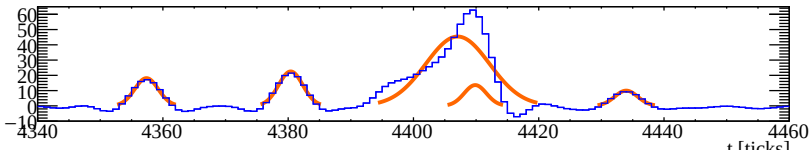
- physics reactions of the particles crossing the detector with GEANT (currently version 4.9.6)
- transportation of the charge produced in the argon volume to the readout wires

`MCReco` module (algorithms `sim::MCShowerRecoAlg` and `sim::MCTrackRecoAlg`) produces special track and shower objects that can be used as reference to evaluate the reconstruction algorithms

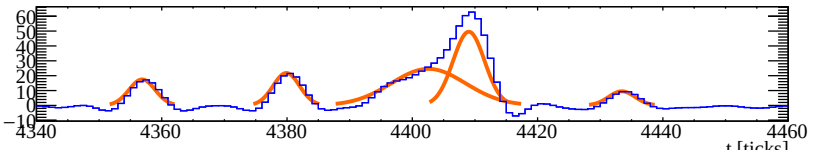
LArSoft: hit finding

Hit finder modules live in `larreco/HitFinder`:

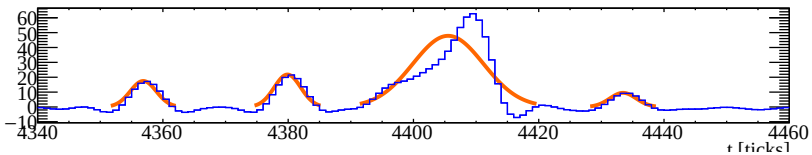
`GausHitFinder` will draw Gaussians to fit the calibrated waveform; each Gaussian becomes a hit:



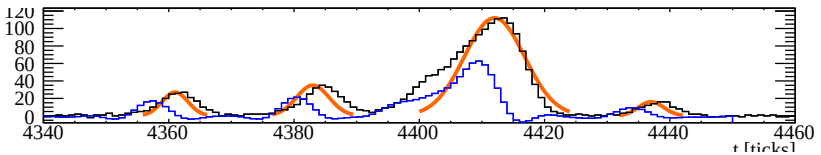
`HitFinder` also draws Gaussians, but it's tuned differently and it's faster; on failure, it produces a "crude hit"



`LineCluster` "refines" hits with additional knowledge from cluster topology



`RawHitFinder` compares the raw, uncalibrated waveform with fixed thresholds; when a threshold is passed, a hit is created



`MCHitFinder` creates special hits from *simulated* energy deposit, binding a hit to a simulated particle.

LArSoft: cluster finding

`DBcluster` (*density-based spatial clustering of applications with noise*) clusters hits that have density above threshold; it has few parameters and works with any cluster shape.

`fuzzyCluster` uses flame clustering (a refined DBSCAN) as pre-clustering, and applies Hough transform to each precluster to identify aligned hits; best performances on track-like clusters.

`LineFinder` (MicroBooNE Doc2831) tracks back long clusters from the "far end" (beam-wise) of the TPC, considering additional information on the neighborhood (e.g. something looking like a vertex) and energy deposit; best performances on track-like clusters.

`BlurredClustering` smears hit charge on a wire plane to create a smooth charge map and then clusters all "pixels" with charge above threshold.

...

LArSoft: track finding (I)

Many tracking algorithms start from “seeds”, small 3D segments:

`Track3DKalmanHit` applies a 3D Kalman filter to 2D hits, including Coulomb scattering in its prediction of where the track is going to develop

`Track3DKalmanSPS` also implements a Kalman filter, but based on 3D “space points”, previously reconstructed from 2D hits

`BezierTrackerModule` connects seeds by a Bézier curve

LArSoft: track finding (II)

Others are based on 2D clusters, treated as whole entities:

`CosmicTracker` correlates clusters from different views (that is, wire planes) by comparing their charge distribution

`CCTrackMaker` puts together the rich information from `LineFinder` and correlates clusters by means of their slopes and charge deposit at their ends

And there are more...

`PMAlgTrackMaker` (AHEP **2013**, 260820) matches track 2D projections to hits, simultaneously in all wire planes

...

LArSoft: shower and vertex finding

`ShowerReco3D` uses a mini-framework that matches clusters from 2 or 3 views:

- based on their time (x coordinate)
- prefers higher number of hits and ignores "track-like" clusters

and then applies energy calibration and reconstructs shower dE/dx

`VertexFinder2D` reconstructs vertices from long clusters on each view, then matches them

`FeatureVertex` produces "proto-vertices" by view matching, and ranks them by proximity to corners found by image processing

`LineCluster` also reconstructs vertices in its crawling

... what?! only one shower reconstruction algorithm??

No... but almost. There is a lot of effort ongoing on this algorithm and on a few other approaches. It's just more complicate.

LArSoft interface to Pandora

Pandora is a pattern recognition toolkit developed at Cambridge:

- development started with Linear Collider tracking in mind
- internally, uses a lot of simple pattern recognition algorithms
- reconstructs hierarchies of particles (*particle flow*) from hits
  - characterizes each particle as track-like or shower-like
  - connects a particle with its daughters
  - associates clusters and 3D points to each particle
- communicates with LArSoft through a module (`LArPandoraParticleCreator`)

LArSoft: secondary reconstruction and analysis

Further levels of reconstruction are also present:

3D clustering pulling together a 3D cluster without passing through 2D ones

trigger reconstruction of collective optical signals (“flashes”)

calorimetry to calibrate the energy deposit of tracks and showers

particle identification including also cosmic ray identification

your algorithm doing cool stuff, coming soon

(I did not write all this code: it comes from the experiments!)

A lot of “analyser” modules are spread in and out of LArSoft:

- dumpers to show in text the content of data products
- plotters extracting distributions used for algorithm validation
- conversion to other formats: ROOT trees, JSON (that’s in *art*, really), LArLite...

Outline

Exercise: write a cluster selector

"Write a module selecting clusters with large width."

The new module

- receives an existing cluster list as input
- produces a new list with only the clusters that have width larger than a configurable value
- comes with a FHiCL configuration file executing this producer
- *[optional]* produces a list of rejected, narrow clusters
- *[optional]* produces associations between new clusters and hits[1]

[1] Note: you can't associate the old and new clusters: `art::Assns` does not support same-type associations.

Details for the exercise

- follow the prescriptions of factorization:
  1. an algorithm performing the filtering
  2. a separate module as interface between that algorithm and *art*
- follow the prescriptions of maintainability:
  1. drop comments about the *meaning* of the code
  2. document the interface of the algorithm and module
- follow the prescriptions of interoperability:
  1. make sure nothing in your algorithm assumes a specific detector feature

Starting point

- the area you have set up with larexamples is suitable for this exercise
- skeleton source is available in `larexamples`'s branch `School2015` under the directory in two versions under `${MRB_SOURCE}/larexamples/producers/skel`:

  `detailed/` contains structure for all both classes and for the implementation of their methods

  `bare/` contains structure for the module, and allows for a lot of freedom in the algorithm

  Pick your style and start by copying the 5 files in the `producers` directory.
- an input file is accessible via `alcourse?.fnal.gov` machines, at:
  `/home/larsoft/course_data/AnalysisExampleInput.root`
  (yes, it's the same as for the previous exercise)

LArSoft data products: `recob::Cluster`

What is inside a `recob::Cluster`?

- `ID()` a number to identify this cluster
- `Plane()` ID of the wire plane where the cluster lies
- `View()` wire orientation (u, v, z)
- `StartWire()`, `StartTick()` one end of the cluster: wire number and time
- `EndWire()`, `EndTick()` the other end of the cluster (as above)
- `NHits()` number of hits in the cluster
- `Width()` width of the cluster, in centimetres
- `Integral()` total charge from its hits
- a constructor with so many arguments
- ... many other quantities

What is *not* inside a `recob::Cluster`?

- its hits expect the same module to produce both clusters and their associations to their `recob::Hit` s

Testing

According to the maintainability prescription you need to:

- write a unit test covering each explicit feature of your algorithm;
  in this case, test on invalid clusters, 0-width clusters, ...
- have an integration test ensuring your module plays well with the rest of the *art*/LArSoft environment

But since we are in a hurry, tests are already in the `School2015` branch:

- unit test in `${MRB_SOURCE}/larexamples/test/producers`:
  should fit MyClusteringAlg interface from the “detailed” skeleton, might need some trivial function name change with “bare” one
- integration test configured in
  `${MRB_SOURCE}/larexamples/test/ci`:
  runs the FHiCL file `SelectWideClusters.fcl` on two events of the input file

Exercise checklist (I)

- set up your own feature branch in `larexamples`
  - merge the branch `School2015` in: `git rebase origin/School2015`
- create a new algorithm class
  (`WideClustersAlg.h`/`WideClustersAlg.cpp`)
  (check the ones provided in `skel/` directory)
- add/extend members that are required for it to perform its task:
  - data holding the input and the output
  - constructor, including configuration parser
  - input method, reading the input list from the event
  - run method, performing the selection
  - output method, returning the list of selected clusters
  - *[optional]* cleanup method, removing the unused data structures
- make sure the unit test has the correct names for the algorithm methods
  (it's at
  `larexamples/test/producers/WideClusterAlgTest.cpp`)

Exercise checklist (II)

- create a new module (`WideClusters_module.cc`)
  (check the ones provided in `skel/` directory)
- check `larexamples/producers/CMakeLists.txt`
  - `art_make()` includes all the needed libraries for the algorithm
  - `art_make()` includes all the needed libraries for the module
- modify module's constructor to
  - declare it will produce a vector of clusters
  - create and configure the algorithm
- modify module's `produce()` method to:
  - read the input from the event
  - pass the input to the algorithm
  - run the algorithm
  - read the result from the algorithm
  - write the result to the event
- compile the new module

Exercise checklist (III)

- run the unit test
  (modify it if you changed `WideClustersAlg` interface)
- complete the FHiCL file with the parameters
- "install", then run the integration test suite `default_larsoft`
- run the new module with the test input
- check that the results are as expected using as input tag:
  - `fuzzycluster`
  - `linecluster` (that produces clusters with width 0...)
- boast with your neighbours[2]
- think about how to add another collection for the rejected clusters
- think about how to port the associations to the new collection

[2] Solutions can be found in the subdirectories under `solutions/`.

Let's do it!

Cheat sheet

set up: see Saba's slides first!

set up for development, MRB flavour:

```
source /products/larsoft_setup.sh
cd <work directory>
source localProducts_*/setup
mrbsetenv
```

set up for running your new code: as development setup, then:

```
mrbslp
```

use a specific branch:

```
cd "${MRB_SOURCE}/larexamples"
git checkout School2015
```

create a feature branch and update it to another branch:

```
cd "${MRB_SOURCE}/larexamples"
git flow feature start "${USER}_School2015"
git rebase origin/School2015
```

Disclaimer: I did not try any of these, watch up for typos!

Setup panic mode:

```
mrb zapBuild
```

then log out and back in, follow MRB development setup, and recompile:

```
mrb build -j4
```

Can I speed up building? the following *most often* works fine after you have built in the regular way once:

```
cd "$MRB_BUILDDIR"
make -j4
```

Run unit tests: follow development setup, compile, then

```
mrb test -j4
```

Run integration tests: follow run setup, then

```
setup lar_ci
test_runner default_larsoft
```

Where is LArSoft documentation?? see Erica's slides; Doxygen documentation is hosted in `nusoft` server as `http://nusoft.fnal.gov/larsoft/doxsvn/html`

Solutions

A set of solutions to the execise can be found in `${MRB_SOURCE}/larexamples/producers/solutions`:

`minimal/` a module producing a single list of wide clusters

`rejected/` a module producing a two list of clusters, one with the wide ones, one with the non-wide ones ("narrow")

`associations/` a module producing two lists of clusters and their associations with hits

Why would `WideClustersAlg::YieldSelectedClusters()` return indices?

In the solutions, `WideClustersAlg::YieldSelectedClusters()` returns a vector with indices of the selected clusters in the original input collection (`std::vector<size_t>`).
It could have returned copies of the selected clusters (`std::vector<recob::Cluster>`) or even pointers to them (`std::vector<recob::Cluster const*>`).
This choice was done to make the creation of associations easier, since associations are based on indices in data collections.
An even more efficient way would have been to return `std::vector<art::Ptr<recob::Cluster>>` (or `art::PtrVector<recob::Cluster>`), but that would require the algorithm to know about *art* structure, that violates our current factorization prescription.

(this prerscription might be different in the future...)