LArLight
Simple LArSoft Data Analysis Tool

Kazuhiro Terao, kazuhiro@nevis.columbia.edu

March 12th, 2014

Abstract

LArLight is a simple C++ + ROOT analysis framework for LArSoft data. The goal of the framework is to provide (1) a generic C++ & ROOT code development environment with build support on various platform, (2) an access to *ala* LArSoft data and utiltiy classes outside LArSoft framework, and (3) analysis framework for data processing. It fits somewhere between LArSoft and AnalysisTree. It is suitable for those who want to access details of LArSoft data and develop analysis/reconstruction code quickly. One of LArLight design principles is to be flexible and comfortable to developers. The framework builds on multiple platforms (Linux and OSX Darwin) and supports various ways to execute the compiled code (compiled executable or CINT /PyROOT interpreter scripts). It comes with a simple package generator that include a build system. It can be also used just as a C++ play ground for students. Please feel free to use it for any purpose. The author is happy to be contacted for questions/suggestions anytime.

Contents

Chapter 1

Installation

This chapter describes (1) prerequisite of installing LArLight, (2) how to checkout, (3) how to build, and (4) some tests to validate the installation.

1.1 Prerequisite

LArLight is a very simple `C++` extention package based on `ROOT` . At this point supported platforms include Linux and OSX Darwin.

1.1.1 Minimum Requirements

Here is a brief list of requirements you need to checkout and compile LArLight:

- Supported Operating Systems
  - Mac OSX 10.5.8 (Darwin 9.8) or later with Xcode 3.1.4 or later
  - Linux 2.6.0 or later with `g++` 4.1 or later
- Version Control Tool
  - `git` 1.7.1 or later
- `ROOT`
  - v5.28.00 or later

If your system does not meet the requirements above, or if you have requirements above fulfilled but have an issue with the build, please contact the author!

Note that these are minimal requirements to build LArLight. It's strongly recommended to have resources noted in the following subsection so that you can use:

- `C++11` features
- `PyROOT`
- Analysis on LArLight files converted from LArSoft.

1.1.2 Recommended Resources

Following features are strongly encouraged:

- Operating Systems & Compilers ... to use `C++11` features
  - Mac OSX 10.6 (Darwin 10.8) or later with `clang` 3.1 or later
  - Linux 3.11 or later with `g++` 4.3 or later
- `python` 2.6 or later (but not 3.X) with headers/libraries
  - Headers/libs for OSX can be obtained through Xcode or source tree
  - For RHL: "`yum install python-devel`"
  - For Debian: "`apt-get install python-dev`"
- `ROOT` v5.34.07 or later with `PyROOT`
  - This version seems compatible with LArLight files created by LArSoft's `ROOT` process.
- `doxygen`
  - LArLight includes `doxygen` script + uses `doxygen` style comment all the places

Please note there are dependencies among items listed above. In particular, LArLight depends on `ROOT` , `PyROOT` depends on `python` , and of course all compilation depends on the compiler. If you are to obtain/update those, you might want to do them in order as well. For instance, install `clang` , install `python` 2.7, then compile `ROOT` from source (to get `PyROOT` compilation with your new `python`), and then compile LArLight.

I don't have `clang` . What do I miss?

Please refer to FAQ in Sec.10.1.

Why `PyROOT` ?

Simply because `python` is

1. Far more easier and stable to use compared to `CINT`
2. Real object oriented interpreter language
3. Full access to `ROOT` and LArLight compiled libraries through `PyROOT`

If you never used `PyROOT` before, you might worry that using an interactive language like `python` (and `CINT`) is slow. That is not necessarily true. Compile the complicated part of your code (which takes computation time) in `C++` , then use that in `python` . This way, your `python` script runs very fast. Strength of using `python` + `PyROOT` is further discussed in Ch.8.

PyROOT Installation Help?

By default, recent ROOT installation (>5.28 as of now) includes PyROOT IFF ROOT configuration script finds python header and libraries. On some Linux installation including Ubuntu 12.04LTS, libpython2.X.so and/or python.h is missing, and ROOT abort PyROOT installation. If you wish, you can simply try installing python-dev, then re-configure ROOT and recompile.

Once you are sure ROOT is configured with python and compiled, check if you find ROOT.py under $ROOTSYS/lib. Here I assumed your ROOT library installation location. Change it to whatever you set if not same. This ROOT.py is what python actually uses to import ROOT . If you find it, try:

```
export PYTHONPATH=$ROOTSYS/lib:$PYTHONPATH
```

This way, python automatically finds your ROOT.py when you attempt to "import ROOT" in python session.

Contact the author if you have a problem installing PyROOT (as he recommends using PyROOT !).

1.2 Checkout Repository

LArLight is maintained at the github.com public git repository:

https://github.com/larlight/LArLight

Accordingly you need to have git installed on your machine. Once you have git installed, you can use standard set of commands to checkout and commit your code.

1.2.1 Getting Write-Access to Commit Code

It is recommended that you set up such that you can commit your code to the repository. Here's a step to follow.

1. Make your github.com account if you don't have one to share with the author. It's free and easy.
2. Notify the author with your account name. You will be added as a collaborator.
3. It is handy to automatically access through ssh. For this, you need ssh key registered on your github.com account. If you have no idea and just want instruction to follow, try these steps:

   ```
   (on your terminal)
   > ssh-keygen -t rsa
   ... (it asks you to set a password ... I set to empty but maybe a bad habit) ...
   > ls $HOME/.ssh/id\_rsa.pub
   ```

The content of id_rsa.pub file is your ssh key. Go to your account setting page on github.com. There, you find an option to add SSH-Key. Add the id_rsa.pub file content there. Now you can checkout/commit through ssh.

1.2.2 Checking Out

With github account and ssh-key set up, you can checkout the repository as follows.

```
> git clone git@github.com:/larlight/LArLight MyLarLight
```

where "MyLArLight" is the name of local repository created under your currently working directory.

Without ssh-key, you can still checkout:

```
> git clone http://github.com/larlight/LArLight MyLArLight
```

Checking Out "development"

Now, the working branch is very unforunately not called "develop" but "trunk". Check out this branch.

```
> cd MyLArLight
> git checkout trunk
```

Fixed Releases

You can also checkout a fixed release.

```
> cd MyLArLight
> git checkout tags/$TAG
```

where `$TAG` should be one of available tag names. Refer to Ch.11 for updates on each tags.

Whether you use "trunk" or a fixed tag, you should find `MyLArLight/core` directory created. This is the core LArLight package you will build regarding LArSoft data analysis.

1.2.3 What's in there?

Under the top directory, you should find the following list of contents.

- `core`
  - This directory contains the core part of LArLight framework including `Base` , `DataFormat` , `Analysis` , and `LArUtil` . Built by default (see Sec.1.3).
- `UserDev`
  - User code development area. This is where you can develop your code with a prepared package template generator.

- `config`
  - Where the user shell environment config script resides. Only important one is `setup.sh` (we will use it in Sec.1.3).
- `Makefile`
  - Where the build orders for `gmake` resides. Not to be modified by a user.
- `lib`
  - Where the built libraries (symbolic link though) will be gathered and reside. The configuration script adds this directory to your `LD_LIBRARY_PATH`.
- `doc`
  - Documentation directory containing doxygen script (see Sec.1.3 for how to use it).
- `README.txt`
  - A brief README for LArLight (you don't need it as you are reading this manual).
- `ASK_QUESTIONS.txt`
  - `Lists list of authors for packages written under UserDev, in case you have a question and wish to contact.`

1.3 Build

1.3.1 Building `core`

The repository contents are not discussed here, but we attempt build the framework blindly. Make sure you have a list of prerequisite (see 1.1) checked out. Try:

```
(... you are in MyLArLight directory ...)
> export MAKE_TOP_DIR=$PWD
> source config/setup.sh
```

If above commands give you a message telling you to type "make", then your set up is complete. If that is not the case, ask for a help. Assuming you were succesful, try:

```
> make -j4
```

where "-j4" means to use 4 threads/cores to compile. Change the number to what is suitable for your case (you can ask more than available but that might slow down the process).

Look for a compiler error... No? Great! Yes? Sorry... please contact the author!

The build takes about 30 seconds on the author's MacBook Pro (OSX 10.9.1). It takes about same time on Ubuntu 12.04LTS running on the virtualbox. It takes, however, a few minutes on uboonegpvm nodes for the first time build. This is likely related to the fact these machines are often under heavy use.

Success Build?

Once the build is finished, check:

```
> ls lib
```

You should find `libBase.so, libDataFormat.so, libAnalysis.so, libLArUtil.so`. These are four building blocks of the framework and will be described in the later section of this document.

1.3.2 Building `UserDev`

I hope you will eventually write your own package and enjoy coding. This happens under `UserDev` directory. When you checked out, you see there are lots of sub directories under `UserDev` . These are all packages written by someone from scratch.

In order to build packages under `UserDev` , you can go to a package directory and type "make". But hey, obviously we don't want to do that for many packages. Moreover, you might want to specify a default set of packages to be built each time. This can be done by the following steps:

Here is a way to specify two extra packages, `FEMPulseStudy` and `FEMPulseReco` :

```
> cd $MAKE_TOP_DIR
> export USER_MODULE=''FEMPulseStudy FEMPulseReco''
> source config/setup.sh
> make -j4
```

As you see, the packages to be compiled under `UserDev` can be specified by `$USER_MODULE` shell environment variable. Sourcing `config/setup.sh` creates a new GNUmakefile to compile the specified packages in the respective order to keep the dependency straight if there is any.

1.3.3 Generating Documentation

If you have `doxygen` installed, you can generate your own web documentation that includes `C++` class/namespace index, just like that of `ROOT` .

```
> cd $MAKE_TOP_DIR
> make doxygen
```

You can open the top page using firefox:

```
> firefox $MAKE_TOP_DIR/doc/dOxygenMyProject/html/index.html
```

or, on Mac OSX,

```
> open $MAKE_TOP_DIR/doc/dOxygenMyProject/html/index.html
```

1.4 Testing Your Build

Here is a description of minimal set of tests to see if your build was successful or not. Try the following commands on `CINT` or `PyROOT` :

```
> root
root[0] gSystem->Load(‘‘libBase’’);
0
root[1] gSystem->Load(‘‘libDataFormat’’);
0
root[2] gSystem->Load(‘‘libAnalysis’’);
0
root[3] gSystem->Load(‘‘libLArUtil’’);
0
root[4]
```

Each line above tells `CINT` to load compiled shared object libraries. They can be physically found under `MAKE_TOP_DIR/lib` directory. Return value of 0 means the library was successfully found and loaded.

Once libraries are validated, you can compile some very basic test routines in each package. Each of `Base` , `DataFormat` , `Analysis` , and `LArUtil` has `bin` subdirectory. The `bin` directory holds test routine source code, README with descriptions, and its own GNUmakefile (i.e. you can type "make" there and compile `bin` directory). Following subsections describe briefly about available test routines.

1.4.1 Testing `Base`

You can read Ch.3 to learn about this package. A simple routine available here is `test_base`. This routine simply checks if `C++` classes defined in the package are available and functions as expected.

It means a success if there is no segmentation fault upon running the executable.

1.4.2 Testing `DataFormat`

You can read Ch.4 to learn about this package. A simple routine available here include:

- `test_simple_io` ... This is a self-contained test routine. It creates a sample LArLight `ROOT` file with some events using I/O interface class (`storage_manager`), then it uses the same class to run and evnet loop to validate the written file.
- `simple_write` ... This routine writes some events with fake track data product using I/O interface class defined in `DataFormat` . Output test file can be used for `simple_read`.
- `simple_read` ... This routine runs an event loop using the compiled I/O interface class. In particular it attempts to read a track data product (and does nothing else). So you want to feed in a LArLight `ROOT` file with a track data product. You can use an output of `simple_write` routine to try out.

1.4.3 Testing `Analysis`

You can read Ch.5 to learn about this package. A simple routine available here is `test_simple_ana`. This routine takes an input LArLight `ROOT` file, and runs an event loop using the `Analysis` framework. For an input file to test the package, if you do not have one, you can use a file generated by `simple_write` routine introduced in the previous subsection.

1.4.4 Testing `LArUtil`

You can read Ch.6 to learn about this package. A simple routine available here include:

- `print_constants` ... This is a simple routine to print some parameters from `Geometry`, `LArProperties`, and `DetectorProperties`, utility classes equivalent of those in LArSoft.

Chapter 2

Introdcution

LArLight was originally developed for Nevis 2013 summer students as a `C++` development play ground. Then it was expanded to perform analysis on ala LArSoft data products. This section briefly describes what it is about in the author's poor English skill.

2.1 Spirits Behind LArLight

There are some points that the author have kept in mind when he has not forgotten.

- Easy to checkout, configure and compile on Linux/Darwin
- Based on `C++` , depends only on `ROOT`
- Simple code development environment with fast build process
- Support to extend compiled library into interactive languages (`python` and `CINT`)
- Help user's code development as much as possible
- Allow flexibility in users' coding: anyone can be a developper

When you find a contradiction while using LArLight, please (1) fix it if you can or (2) speak up and ask to make it better!

2.2 What Is LarLight For?

Right, what is it for? Originally it was for `C++` + `ROOT` code development. This remains same now. It became capable of doing ala LArSoft data analysis. This is an on-going effort to become better. Following subsections describe briefly how LArLight support these points. After those, IMHO the strength of LArLight is described.

It is important to note here the author has no intention to replace LArSoft by LArLight. Please do not even start arguing about this with him. You make him sad.

2.2.1 For Simple C++ Code Development

You can forget about doing LArSoft analysis (or even about microboone) and use LArLight to develop a generic C++ + ROOT project. This is just executing a one line command to generate your package, and type "make". See Sec.7.1 for an example.

2.2.2 For ala LArSoft data analysis

LArLight is equipped with following features to perform ala LArSoft data analysis.

- Data structure capable to store LArSoft data contents
- LArSoft module that converts data from LArSoft to LArLight ROOT file
- Dedicated I/O interface, like LArSoft's art::Event
- Analysis framework, like LArSoft's art::EDAnalyzer/EDProducer

In case you are wondering why LArLight can do such things... in the end of the day, what we are analyzing/reconstructing is just a group of integers and floating points. Processing of those are certainly supported in a standard C++ and ROOT framework. So surely anyone can write a simple framework like LArLight to process data of our interest.

2.2.3 Strength: Why Using LArLight?

IMHO LArLight is useful because:

No extra dependency like art

- It means the framework build on my laptop. It builds fast, and easy to debug errors as it only depends on standard C++ and ROOT which most of us are familiar with. These imply to a comfortable environment for code development.
- It means your code in LArLight is **easily transportable to any framework with C++ and ROOT dependencies**, including LArSoft or your future experiments!

Capability of fast data processing

- Dedicated I/O interface saves you from treating TTree branches by hand
- Easy and fast data processing = you can study more with data
- Yet you have access to ala LArSoft details of data, including associations
- Modulated analysis unit design (ala EDAnalyzer/EDProducer) allows easy sharing and maintenance of code

Interactive language friendly ... python / CINT

- Supports import of compiled libraries = same execution speed as compiled code
- It means you can access data (for instance waveforms) in several lines of code

NOT LArSoft

- Do whatever you want to data file. For instance, reduce file size by only storing interesting info for you.
- Different design principle. Probably a bit more flexible.

LArLight probably fits a best bridge between LArSoft data and simple `AnalysisTree`. But it is not a replacement of either.

2.2.4 It is NOT a Replacement of Anything

LArLight is not a replacement of any software framework developed before.

Obviously it's not a replacement of LArSoft. The author refuses to discuss about this.

Even if you develop an amazing GUI interface for viewing events, it will not be a replacement of `Argo`. `Argo` does not require data product format, and hence is much more flexible than LArLight in that sense. The purpose of LArLight having ala LArSoft data products is to do detailed analysis AND to provide, like said before, a comfortable environment for code development.

Can it be a replacement of `AnalsisTree`? No. Not at least in the author's understanding of what `AnalysisTree` is for. The `AnalysisTree` is meant to be a simple `TTree` that stores higher level physics variables you know you are interested in. Hence it is not meant to be used for developing complicated algorithms that require more basic level information in data.

If `AnalysisTree` is to access very high level analysis information outside LArSoft, LArLight is suitable for accessing full detail of information outside LArSoft. In otherwords, one can produce `AnalysisTree` from LArLight. But again, it is not a replacement.

Chapter 3

Core Package: Base

The most important thing contained in `Base` package is constants and `enum` that are used throughout the framework. It also include some framework base classes. Both are described in this chapter.

3.1 Constants and `enum`

Constants are spread over several header files. The author is not sure if that was a good idea, but he thought that might appear more organized. It is not because making more header files look framework cool and complicated. Definitely not. A list of header files is shown below:

`FrameworkConstants.hh` ... constants used to configure this framework

`DataFormatConstants.hh` ... constants related to data products in this framework

`MCConstants.hh` ... MC constants copied from LArSoft

`GeoConstants.hh` ... geo constants copied from LArSoft

`FEMConstants.hh` ... FEM electronics related constants

`DecoderConstants.hh` ... Used in separate package (NevisDecoder), you can ignore

Following subsections describe what's defined in each of them.

3.1.1 FrameworkConstants.hh

This framework comes with some message service class (`Message`). This message service is not sophisticated at all but described in the later section. It works based on message "levels", which is specified by `enum MSG::Level`. This message level and color-coding scheme of each message can be found in this header file.

3.1.2 DataFormatConstants.hh

There are three kinds of important parameters defined in this file.

Numeric Limit: `const XXX INVALID_XXX`

There are list of numeric limits that can be used to label some "undetermined" variable. I follow a strategy used in LArSoft's MC default variables and use maximum possible value for each variable type. That being said, sometimes LArSoft uses "999" or this sort to mark "undetermined". So watch out: that is likely not corrected in your LArLight file if you copied data contents from LArSoft.

Data Type: `enum DATA::DATA_TYPE`

Probably most important `enum` . This defines a set of data types in LArLight. You should find, for each type, a corresponding LArSoft data type. This is used to retrieve data using I/O interface class (described later).

If you added a new data product class, you should add it in this `enum` . You can add a new element anywhere, but the gentlemen's rule is to add it in the order of basic information =¿ higher level information. For instance, this order: `Wire, Hit, Cluster, Shower`.

Data Name: `std::string DATA_TREE_NAME[]`

This is an array of `std::string` to define a corresponding name for each `DATA::DATA_TYPE`. The length of the array is, therefore, `DATA::DATA_TYPE_MAX`. This is used to name `TTree` stored in LArLight `ROOT` file and such, described in later sections.

3.1.3 MCConstants.hh

This defines MC constants used in LArSoft data products. To store same information, LArLight needs these constants, too. Both constants name and values are exact copy from LArSoft to avoid a confusion. Copied constants include:

`Origin_t` ... specifies particle generator type

`curr_type` ... specifies neutrino interaction current type

`int_type_` ... specifies neutrino interaction categories

Unfortunately you cannot blame the author for inconsistent naming scheme. If you insist, please change in LArSoft! Then we may propagate a change to here.

3.1.4 GeoConstants.hh

This defines two `enum` that belong to `geo` in LArSoft that is stored in the data product. The first is `geo::View_t` that specifies wire plane IDs. The second is `geo::SigType_t` that specifies signal type, induction or collection.

3.1.5 FEMConstants.hh

Some FEM specific constants and `enum` are defined in this file. This is originally written for `NevisDecoder` package and now merged with LArLight. It has a correspondency to LArSoft's constants in `optdata` namespace, defined under OpticalTypes.h.

All users care, I believe, is `FEM::DISCRIMINATOR` which defines which discriminator type is fired in PMT-FEM. If you are dealing with TPC-FEM, you can ignore this `enum` .

3.2 Framework Base Class

There are two basic classes used throughout the framework: (1) `larlight_base` and (2) `Message`. The first one is used as the base class of all framework classes except for data products. The second class is a very simple `std::cout` and `std::cerr` message carriers.

Before wasting your 5 minutes, it's not very important to know about these classes unless you want to use a color-coded message scheme.

3.2.1 Message Carrier: `Message`

This is actually not a logger. It does only one thing: color code the message and add a prefix based on message level.

3.2.2 Framework Base: `larlight_base`

This is the base class of all framework classes except for data products. It comes with a verbosity level and makes a use of `Message` class to deliver messages. It is implemented with `larlight_base::print` attribute which can mask out some low level messages based on the message level (i.e. `MSG::Level`).

Chapter 4

Core Package: DataFormat

This package contains:

- Data product class
- `ROOT` file I/O interface class

In this chapter, we explicitly put `larlight::` namespace specification to LArLight classes to avoid confusion with LArSoft data products.

4.1 Data Product Class

The data product classes are meant to store LArSoft output data, hence their design follows closely with what is in LArSoft. If you look at a header file of data product class, you should find LArLight data product classes carry same or very similar attributes name to reduce confusion. For adding further data product classes to store LArSoft data, it is recommended to keep this trend to reduce confusion. Here is a list of LArLight data products (left) with corresponding LArSoft data product (right) as of now.

```
larlight::mctruth <==> simb::MCTruth
larlight::mcnu <==> simb::MCNeutrino
larlight::mctrack <==> simb::MCTrajectory
larlight::mcpart <==> simb::MCParticle
larlight::tpcfifo <==> raw::RawDigit
larlight::pmtfifo <==> optdata::FIFOChannel
larlight::wire <==> recob::Wire
larlight::hit <==> recob::Hit
larlight::cluster <==> recob::Cluster
larlight::spacepoint <==> recob::SpacePoint
larlight::track <==> recob::Track
larlight::shower <==> recob::Shower
larlight::endpoint2d <==> recob::EndPoint2D
larlight::vertex <==> recob::Vertex
```

```
larlight::calorimetry <==> anab::Calorimetry
larlight::trigger <==> (not yet existing in LArSoft)
larlight::user_info ... LArLight-only custom data product
```

There is additional data product called `user_info` which will be described later.

All data product class shares the base class `larlight::data_base` which stores two variables:

- `larlight::DATA::DATA_TYPE`

  This `enum` specifies the type of data stored. For instance, `larlight::hit` can be assigned with `larlight::DATA::FFTHit`, `larlight::DATA::APAHit`, and so on. By accessing this enum value, a user knows what kind of "hit" is stored.

- `std::map<larlight::DATA::DATA_TYPE,std::vector<std::vector<unsigned short> > >`

  This is used to specify the association. This is described later in detail.

4.1.1 What is stored per event?

Despite a few exceptions (which I describe next), each data product class `T` has a corresponding collection class which inherits from `std::vector<T>`, named as `event_T`. For instance, in `cluster.hh`, you find there is a class `larlight::cluster` and `larlight::event_cluster` where the latter is a type `std::vector<larlight::cluster>`. What is stored per event is `std::vector<T>`, just like a collection of object is stored in LArSoft. This collection data product also inherits from `data_base`, but it carries additional inforamtion:

- Event number
- Run number
- Sub-Run number

Exceptions to this event-wise collection storage scheme include `larlight::mcnu` and `larlight::mctrac`
In LArSoft, `simb::MCNeutrino` is stored as a part of `simb::MCTruth`. Likewise, `simb::MCTrajectory` is stored as a part of `simb::MCParticle`. LArLight respects this scheme and stores `larlight::mcnu` in `larlight::mctruth` and `larlight::mctrack` in `larlight::mcpart`.

A few remarks on MC data products

In case you are unfamiliar with these MC information, though this is a side-track, `larlight::mcnu` stores neutrino specific information from neutrino generator. It is a part of `larlight::mctruth` because a generator (whether it is neutrino generator or not) stores `larlight::mctruth` in general. Similarly, `larlight::mctrack` is part of `larlight::mcpart` because a particle's trajectory points (that is what `larlight::mctrack` is) belong to a particle information which is `larlight::mcpart`.

More useful info to store?

If you have some information you think is very useful, let's store it :) For instance, it was advised that knowing a part of `larlight::mctrack` that is inside the fiducial volume is very useful information. So we added this information in `larlight::mctrack` for users so that a user can access particle trajectory points inside the fiducial volume quickly instead of looping over all trajectory points and inspect whether it is in the fiducial volume or not.

Your ideas can help others. So please share it!

Adding new data product?

By all means, go ahead. There are important pieces missing, for instance vertex! That being said, however, a mistake can break DataFormat which is shared among all users. So feel free to consult with the author.

Alternative solution is to ask the author to add it. Believe me, it's so simple and it won't take me ¿ 10 minutes to add another on the list above.

4.1.2 Custon data product for dynamic variable declaration ... `user_info`

Sometimes you want a custom data product and store information in the output data root file. For instance:

- you want to store some parameter values but not worth making a new data product (say studying a parameter)
- you want to store some parameter values for certain events/objects, but not all.

Then `user_info` might be handy for you. This data product class contains several `std::map` which allows you to store following basic variable types:

```
Bool_t
Int_t
Double_t
std::string
std::vector<Bool_t>
std::vector<Int_t>
std::vector<Double_t>
std::vector<std::string>
```

When you store these variables in `user_info`, you store with a `std::string` key. Then you can access the stored value by this key. You can use the same key to store different type of variables: mixing is prevented. If you reuse a key that is already used to store the same type of variable, you will overwrite the value. There is a function to check if the key is already in use or not, so a user can avoid overwriting the value.

Using `std::map` is not great for speed when retrieving the variable. But this seems to be a matter of micro-seconds or less, depending on number of keys used.

4.1.3 Association

The author has no idea how association is stored in `ART` . But that does not matter.

As you have seen in the previous section, all data product are stored in terms of `std::vector` collection. So ultimately what you need to associate one object to (an)other are:

- A set of indexes that point to the right `std::vector` index.
- Associated data type to know which collection `std::vector` to access.

To do this, `data_base`, the base of all data product, has:

```
std::map<larlight::DATA::DATA_TYPE,std::vector<std::vector<unsigned short> > >
```

My god, such a long specialization name...

Then we wonder why we store a "vector of vector" instead just a "vector"? This is because `ART` is capable to store more than one set of association (I think). Even if `ART` won't do that, having LArLight being capable to store multiple association of the same data type could be handy. So that's why we have such thing in `data_base`.

Retrieving association is as simple as calling the following function:

```
data_base::association(larlight::DATA::DATA_TYPE, size_t i=0)
```

You provide the association type in the 1st argument. The 2nd argument is to specify which association of the provided type to access. Because we have only one set of association in many cases (for example, one cluster has only one set of hits associated), almost always the 2nd argument is 0 (i.e. 0th element of vector of associations). For this reason, the default value of the 2nd argument is 0, and most users do not need to provide 2nd argument.

4.2 `ROOT` file I/O interface

LArLight uses `ROOT` file as storage because the author is not smart enough to write his own streamer. In a LArLight `ROOT` file, you find `TTree` per stored data type (i.e. `larlight::DATA::DATATYPE`). A standard approach of using `TTree::SetBranchAddress` works very easily, but LArLight also has dedicated I/O interface to make our life easier (at least that was the author's intention).

If you did not find the I/O functionality you want in here, please suggest and help us to improve our interface ;)

4.2.1 LArLight `ROOT` file structure

In LArLight `ROOT` file, TTrees are created per stored data type. You find, for instance, "ffthit_tree" if `larlight::DATA::FFTHit` data type is stored.

Accessing "by hand"

Here's some simple commands you can try to access in CINT (or PyROOT) session:

```
> root
root[0] gSystem->Load(‘‘libDataFormat’’);
root[1] TFile* f = TFile::Open(‘‘sample.root’’,‘‘READ’’);
root[2] TTree* t = (TTree*)(f->Get(‘‘ffthit\_tree’’));
root[3] t->Show(0);
```

Similarly you can try TTree::Scan or TTree::Draw function. Note that ROOT supports class attribute access upon CINT dictionary loading. Since we do have CINT dictionary, you can try accessing functions like larlight::hit::Charge() in the string argument you provide to those TTree functions.

Of course, you can also invoke TTree::SetBranchAddress to retrieve data product class pointer.

```
> root
root[0] gSystem->Load(‘‘libDataFormat’’);
root[1] TFile* f = TFile::Open(‘‘sample.root’’,‘‘READ’’);
root[2] TTree* t = (TTree*)(f->Get(‘‘ffthit_tree’’));
root[3] larlight::event_hit* my_hit_v = new larlight::event_hit;
root[4] t->SetBranchAddress(‘‘ffthit_branch’’,my_hit_v);
root[5] t->GetEntry(0);
root[6] my_hit_v->size();
```

But this is tedious. Let's use I/O interface class.

4.2.2 Framework I/O Interface: storage_manager

storage_manager is the LArLight's C++ interface class for ROOT file I/O. The work flow is the following:

- Configure (tell input and/or output file information)
- Open
- Loop over events
- Close

We go over some examples in the followings. I assume we have an example file, "hit_sample.root" which contains larlight::DATA::FFTHit type data product.

Simple Example: Opening an input file

Here's a CINT script to configure/use storage_manager to open a ROOT file.

```
{
    gSystem->Load(``libDataFormat'');
    larlight::storage_manager mgr;
    mgr.set_io_mode(mgr.READ); // Option: READ, WRITE, BOTH
    mgr.add_in_filename(``hit_sample.root''); // Specify input file name
    mgr.set_in_rootdir(``''); // Specify TDirectory name if there exists
    if(!mgr.open()) {
      std::cerr << ``Failed to open ROOT file. Aborting.'' << std::endl;
      return 1;
    }
    std::cout << Form(``Retrieved \%d events!'',mgr.get_entries()) << std::endl;
    mgr.close();
    return 0;
}
```

Simple Example: Event loop (Read Only)

This is an example to run an event loop.

```
{
    gSystem->Load(``libDataFormat'');
    larlight::storage_manager mgr;
    mgr.set_io_mode(mgr.READ); // Option: READ, WRITE, BOTH
    mgr.add_in_filename(``hit_sample.root''); // Specify input file name
    mgr.set_in_rootdir(``''); // Specify TDirectory name if there exists
    if(!mgr.open()) {
      std::cerr << ``Failed to open ROOT file. Aborting.'' << std::endl;
      return 1;
    }
    std::cout << Form(``Retrieved \%d events!'',mgr.get_entries()) << std::endl;
    while(mgr.next_event()) {
      const event_hit* my_hit_v = (const event_hit*)(mgr.get_data(larlight::DATA::FFTHit
      if(my_hit_v) {
        std::cout << Form(``Found event \%d!'',my_hit_v.event_id()) << std::endl;
      }
    }
    std::cout << ``Closing a file...'' << std::endl;
    mgr.close();
    return 0;
}
```

Simple Example: Event loop (Read & Write)

This is an example to run an event loop.

```
{
    gSystem->Load(``libDataFormat'');
    larlight::storage_manager mgr;
    mgr.set_io_mode(mgr.READ); // Option: READ, WRITE, BOTH
    mgr.add_in_filename(``hit_sample.root''); // Specify input file name
    mgr.set_in_rootdir(``''); // Specify TDirectory name if there exists
    mgr.set_out_filename(``out_sample.root'');// Specify output file name
    if(!mgr.open()) {
      std::cerr << ``Failed to open ROOT file. Aborting.'' << std::endl;
      return 1;
    }
    std::cout << Form(``Retrieved \%d events!'',mgr.get_entries()) << std::endl;
    while(mgr.next_event()) {
      const event_hit* my_hit_v = (const event_hit*)(mgr.get_data(larlight::DATA::FFTHit
      if(my_hit_v) {
        std::cout << Form(``Copying event \%d!'',my_hit_v.event_id()) << std::endl;
      }
    }
    std::cout << ``Closing a file...'' << std::endl;
    mgr.close();
    return 0;
}
```

Note that this script makes an exact copy of input file to the output. If you want to modify it, you can use non-const `event_hit` pointer in the script above.

Advanced Example: Optimizing I/O by limiting TTree to read

Sometimes your LArLight file may contain much more information than you needed. The file size can be huge if you have stored `RawDigit, Wire, MCTrajectory` and such. By default, `storage_manager` reads in all LArLight type `TTree`, and this can slow down your event loop.

If you don't need it, let's cancel reading in. There is a switch you can enable TTree-wise read:

```
storage_manager::set_data_to_read(larlight::DATA::DATA_TYPE type, bool enable);
```

The first argument specifies the `TTree` type, and second argument specifies enable (or disable) reading. Call this function before calling `storage_manager::open()` function.

Chapter 5

Core Package: Analysis

This chapter describes briefly about `Analysis` package, the analysis framework of LArLight. It comes in largely two pieces:

- Analysis Unit
- Processor

The first bullet corresponds to LArSoft's "module" (i.e. `art::EDAnalyzer/EDProducer`). The second bullet is an executor of analysis unit(s) by interfacing with `storage_manager` (i.e. LArLight I/O interface class, see Sec.4.2).

Before going into details, do not misunderstand that you are constrained to do your analysis this way. Like it was described in Sec.4.2, you have an interface to run your own event loop in whatever the way you wish. Using the analysis framework is just a suggestion and not a requirement. Or please share if you made something better ;)

5.1 Analysis Unit

An analysis unit is a `C++` class that inherits from `ana_base` class. This base class has three important functions that would concern your analysis unit code development.

```
bool ana_base::initialize()
bool ana_base::analyze(larlight::storage_manager* storage)
bool ana_base::finalize()
```

As it is described in the next subsection, an analysis unit is held by analysis processor which runs an event loop.

5.1.1 Flow of Function Calls

`initialize()`

The attached analysis unit's `initialize` function is called only once at the beginning of an event loop. If the analysis unit returns `false` at that point, event loop will not be even started by analysis processor.

`analyze(storage_manager* storage)`

Then for each event, analysis processor calls `analyze` function. Event-by-event analysis code should be implemented within this function. Access to an event's data information is done through `storage_manager` pointer. The return value of `analyze` function is currently unused. Your idea to expand the framework using this portion is certainly welcome. One example is to use this return boolean to implement a "filter unit".

`finalize()`

At the end of event loop, `finalize` function is called. This is where you might want to do final wrap-up of your analysis.

Storing Output

This is sometimes confusing, but there are two kinds of outputs you can store:

- Data product
- Analysis output

To store modified/new data product (i.e. classes defined in Sec.4.1), simply use provided `storage_manager` pointer that is given as a function argument of `analyze`.

To store your analysis output `ROOT` objects, like your `TH1` histogram, `TGraph` and such, use `ana_base::_fout` attribute. As it is defined in `ana_base`, all analysis units should have it. This is a `TFile` pointer provided by analysis processor before calling `initialize` function.

The analysis output file (i.e. `TFile` object to which `_fout` points) is separately created by analysis processor if a user configured to create it. So your object will be stored in a separate output `ROOT` file.

For example, let's say I have analysis unit called `KazuAna` and it has `TH1D` pointer called `h1` with real object on the heap. To save this object in an analysis output file, I can write the following `finalize` function.

```
bool KazuAna::finalize() {
    if(_fout) {
     _fout->cd();
     h1->Write();
    }
}
```

Note that:

- Checking whether `_fout` is valid or not will avoid code crash in case analysis processor is configured not to create output file
- Since `_fout` is shared among all analysis units, no unit should not selfishly close the file.

5.1.2 Why Unit Modulation?

I hope we agree it is better to have multiple functions rather than one function with thousands of lines of code. The basic idea is same: modulation helps to maintaine code and also sharing/reusing the code. For this reason, it is better to restrict one analysis unit does one thing. It does not mean one analysis unit should create more than one histogram. But it means to have a single or a few words answer to the question, "what does this unit do?" Don't continue your "a few words answer" by saing "it also does...". That's cheating ;)

If this is not clear, here is a practical example. Let's say we want to reconstruct number of photo-electrons from PMT signal. You are given a digitized waveform, and the goal is to reconstruct a pulse from waveform and you compute its area or height to estimate electric charge. Then you convert the charge into number of photo-electrons. Unit modulation means, in this example, you write one module to reconstruct a pulse and a separate module to calculate number of photo-electrons from charge in a pulse. So if someone asks what each module does, your answers are: "pulse reconstruction" and "charge to photo-electron conversion".

What we benefit from this? Imagine tomorrow the smarter you come up with a better reconstruction algorithm but you have to prove it is better. Instead of copy and paste the entire code, you can simply add another pulse reconstruction unit in your package. Then you can run your analysis by switching one unit to the other to compare the result. Your analysis unit to convert charge into photo-electrons is undisturbed. You do not have to do any "copy and paste" of code, which often introduces a bug in your program. This is the benefit you get.

Even possibly greater benefit: tomorrow you are doing TPC waveform analysis and you want to use a similar way of reconstructing a pulse, though there is no sense of "photo-electron" involved in this case. By doing unit modulation you can use exact same pulse reconstruction unit to perform the task.

5.2 Processor

Analysis processor runs an event loop and is responsible for executing analysis units. You can attach as many analysis units as you wish. Those analysis units are executed in the respective order. You find `ana_processor` is the actualy `C++` class that takes this role.

Here is an analysis processor's task list in order:

- Start I/O using `storage_manager`
- Call `initialize` function of attached analysis units
- Start event loop via `storage_manager`
- For each event, call `analyze` of analysis units
- Call `finalize` of analysis units at the end of an event loop
- Close I/O

Before the 1st bullet point, a user has to (1) configure `ana_processor` with file I/O information and (2) attach analysis units.

5.2.1 Example: Running Analysis Processor

Probably seeing an example code is much easier. Using, again, `KazuAna` analysis unit, here is an example `CINT` code:

```
{
    gSystem->Load(‘‘libAnalysis’’);
    // Create ana_processor instance
    larlight::ana_processor my_proc;
    // Set I/O mode, like storage_manager.
    my_proc.set_io_mode(larlight::storage_manager::READ);
    // Set analysis output file: not data output file.
    my_proc.set_ana_output_file(‘‘ana.out’’);
    // Set TDirectory name under which TTree resides
    my_proc.set_input_rootdir(‘‘scanner’’);
    // Now add input file(s)
    my_proc.add_input_file(‘‘hit_sample.root’’);

    // Create analysis unit on heap
    larlight::KazuAna* k = new larlight::KazuAna;

    // Attach
    my_proc.add_process(k);

    // Now run event loop
    my_proc.run();
}
```

The I/O configuration is very similar to what you learned about `storage_manager`. As you guessed, this is because `ana_processor` is using `storage_manager` underneath.

After the I/O configuration, like described before, you attach analysis unit(s). You can add as many as you wish.

Finally, `ana_processor::run()` starts a batch (uninterrupted) event processing. Another option is to use `ana_processor::process_event()` which process one event per function call. This can be useful when you want to do an interactive event processing.

Chapter 6

Core Package: LArUtil

When writing analysis/reconstruction code, eventually you will hit the point and ask "where can I look up detector geometry?" or "How can I get detector parameters like electron lifetime?" In LArSoft, there are `Geometry` , `LArProperties` ,`DetectorProperties` , and `GeometryUtilities` service modules that can help you for such need. In LArLight, we implemented similar utility classes with the exact same name. This chapter describes about those utility classes.

6.1 Utility Classes: How It Works

All of `LArUtil` classes are defined under `larutil` namespace, which is different from the rest of core packages. The author is not sure why it was done so. If you think they should be moved into `larlight` namespace, tell him to do so.

There are two features that all utility classes in `LArUtil` shared. These are (1) singleton design pattern, and (2) use `ROOT TTree` to instantiate data members. These are described briefly in the following subsections.

6.1.1 Basic Usage

Like they are in LArSoft, all of `Geometry` , `LArProperties` , and `DetectorProperties` are implemented as singleton class. That means you cannot access these class instances by invoking their constructor. Instead, all of them have `GetME()` function which returns the constant pointer to its own instance.

```
> root
root[0] gSystem->Load(‘‘libLArUtil’’)
root[1] larutil::Geometry::GetME()
```

or in `PyROOT`

```
> python
>>> from ROOT import *
>>> gSystem.Load(‘‘libLArUtil’’);
>>> larutil.Geometry.GetME()
```

You might say "I worry about overhead of calling `GetME()` function each time I need it." First, this won't take even a micro-second. If your code takes more than 10 micro-seconds to run per this function call, you can discard such worry! Secondly, feel free to store `larutil::Geometry` const pointer so that you don't have to call `GetME()`. But never delete it (i.e. respect it should be const)!

As you see above, `LArUtil` classes are supported in `CINT` and `PyROOT` just like other LArLIght classes. Accessing the geometry or detector property information is, therefore, very staright-forward. When you forget how many TPC channels exist in the detector, just try:

```
> root
root[0] gSystem->Load(‘‘libLArUtil’’)
root[1] larutil::Geometry::GetME()->Nchannels()
(const unsigned int) 8254
```

Or maybe getting the position of PMT channel 1:

```
> root
root[0] gSystem->Load(‘‘libLArUtil’’)
root[1] Double_t xyz[3]={0.}
root[2] larutil::Geometry::GetME()->GetOpChannelPosition(1,xyz)
root[3] std::cout<<xyz[0]<<‘‘ : ‘‘<<xyz[1]<<‘‘ : ‘‘<<xyz[2]<<std::endl;
-5.755 : 59.0965 : 938.5
root[4]
```

Or maybe getting the view type of TPC channel 5000

```
> root
root[0] gSystem->Load(‘‘libLArUtil’’)
root[1] Double_t xyz[3]={0.}
root[2] larutil::Geometry::GetME()->View(5000)
(const enum larlight::GEO::View_t)2
```

6.1.2 Support for Other Experiments

By default all utility classes are set for MicroBooNE experiment. Though the design of `Geometry` is simplified compared to LArSoft, it can support experiments as long as it has only 1 cryostat and 1 TPC. Please make a request and the author can help to support experiments not yet supported.

Currently another experiment supported is ArgoNeuT. To reconfigure all utilities for ArgoNeuT, you can do:

```
> root
root[0] gSystem->Load(‘‘libLArUtil’’)
root[1] larutil::LArUtilManager::Reconfigure(larlight::GEO::kArgoNeuT)
```

The class `LArUtilManager` has a single static function that takes experiment type, which is an `enum` defined under `core/Base/GeoConstants.hh`, and reconfigure all utility classes. If the input experiment type is same as what is already set, then it does nothing. You can call above

function at the beginning of your script or `main` function, and the rest of code execution uses the specified experiment type.

6.1.3 Custom Geometry

As described above, utility classes under `LArUtil` have their member variables instantiated through `ROOT TTree`. Accordingly their attribute variables depend on values stored in `TTree`. These `TTree`s are created from LArSoft module and is updated whenever there is a change on the LArSoft side.

That being said, you can create your own version of TTrees using routines under `LArUtil/bin` directory. There, you find `gen_tree_XXX.cc`. Simply type `make` in this directory, and you find a complied executable that can generate a data `TTree` for a utility class `XXX`. The author understands that this way of changing the data member values is certainly cumbersome. If anyone wants a text-file interface, please ask him and he will be happy to implement.

If you have your custom made `TTree` to instantiate the data member, you can feed it in the following manner:

```
> root
root[0] gSystem->Load(‘‘libLArUtil’’)
root[1] larutil::Geometry::GetME()->SetTreeName(‘‘tree_name’’)
root[2] larutil::Geometry::GetME()->SetFileName(‘‘tree.root’’)
root[3] larutil::Geometry::GetME()->LoadData()
```

where we assumed the `TTree` name “tree_name” stored in a `ROOT` file “tree.root”. If you want to re-load data after instantiating the class object, you can also use `SetFileName()` and `SetTreeName()` functions, and then invoke `LoadData()`. This sequence of commands force re-loading of data members from specified input file/tree name.

6.2 *ala* LArSoft Utility Classes

Currently implemneted *ala* LArSoft utility classes include:

- `DetectorProperties`
  - Utility class to access detector properties such as readout window size, sampling rate of a digitizer, etc. All functions available in LArSoft are implemented.
- `LArProperties`
  - This is for liquid Argon properties such as density, temperature, electron life time, scintillation yield, etc. All functions available in LArSOft are implemented.
- `Geometry`
  - This is for detector geometry. You can access a mapping between channel number to plane and wire number, nearest wire for a given position in the detector, wire intersection point, etc. The class is simplified by (a) taking assuming there is only

one tpc and cryostat and (b) not using TGeoManager (i.e. the class is merely carrying constants to do most of useful calculation).

- `GeometryUtilities`
  - This is a utility class mainly used by `ClusterParamsAlg` for shower 2D cluster reconstruction. It is equiped with useful functions to make an easy interpretation of detector geometry such as conversion of 3D point onto a 2D (time vs. wire) plane.

If you find any function to add, please ask the author. The point for utility classes are to be useful for you! In addition, if you are missing LArSoft utility class you wish to use, feel free to ask the author about this as well.

Chapter 7

Generate & Build Your Package

This chapter discuss about how to generating user's own package. There are two kinds of projects for the moment:

- A simple C++ project package
- An analysis package that uses Analysis framework

7.1 Simple C++ Project

Like advertised many times, LArLight was originally a C++ project play ground for summer students. The author thinks it's very important to have an empty C++ package generator that comes with build system, so that a user can focus on writing the algorithm instead of figuring out how to compile and such.

So here it is: there is a python script to generate an empty C++ package:

```
UserDev/bin/gen_empty_package
```

Running it:

```
> cd $MAKE_TOP_DIR/UserDev
> python bin/gen_empty_package MyProject
```

will create a directory UserDev/MyProject which include minimal set of source codes to compile an empty C++ class. You can have your name choice in place of "MyProject". After running the command, try:

```
> cd MyProject
> make -j4
```

This compiles your code and makes libMyProject.so under lib directory. What you compiled is a C++ class called "MyProject" defined in MyProject.hh.

7.1.1 Using in Interpreter

So how can you "use" this C++ class? You can write a binary executable code, or try out in CINT or PyROOT . Here is an example:

```
> root
root[0] gSystem->Load(‘‘libMyProject’’);
root[1] MyProject k;
root[2]
```

Above lines work (i.e. your class instance is created) because CINT is informed about your class.

7.1.2 “Hello World” Development

Let’s try a simple modification to your class. Here’s an example “hello world” program using C++ class. Open MyProject.hh and add a void function as shown below:

```
class MyProject{
public:
  /// Default constructor
  MyProject(){};
  /// Default destructor
  virtual ~MyProject(){};
  /// Hello world!
  void HelloWorld() { std::cout << ‘‘Hello World!’’ << std::endl; }
};
```

Save, close and compile (i.e. type “make”). Now, try the following in CINT or PyROOT :

```
> root
root[0] gSystem->Load(‘‘libMyProject’’);
root[1] MyProject k;
root[2] k.HelloWorld();
Hello World!
root[3]
```

Whenever you want to start a new project with an empty class, you can come back to this example and create your new C++ project!

7.2 Your Own Analysis Package

Here, by Analysis package, we mean a directory with a collection of analysis unit source code. So it is meant to use LArLight analysis framework.

Note that, if you do not want to be in the Analysis framework, you can choose to generate an empty C++ package instead of Analysis package. See Sec.7.1 to learn how to do this.

7.2.1 Generating Your Analysis Package

When you develop your code, you probably don’t want to write your code under Base , DataFormat , LArUtil , or Analysis (i.e. sub directories under core directory) because those directories are basis of the framework and everyone shares them. So how can you work without

disturbing others and, most importantly, without being disturbed? A simple solution is to generate your own analysis package and work in there.

There is a `python` script that allows you to generate a new package:

```
> cd $MAKE_TOP_DIR/UserDev
> python bin/gen_ana_package MyAna
```

Executing above commands create a new directory called MyAna under `UserDev` . There, you find `MyAna.cc` and `MyAna.hh` together with `GNUmakefile` which is necessary for building the package. Try:

```
> cd MyAna
> make -j4
```

This should compile your package and create `lib/MyAna.so` which includes `CINT` dictionary. You just made your new analysis unit class, `MyAna`, whieh inherits from `ana_base`! Your new analysis unit is accessible from `CINT` or `PyROOT` like any other classes in LArLight:

```
> root
root[0] gSystem->Load(‘‘libMyAna’’);
root[1] larlight::MyAna my_ana_unit;
```

Now go ahead and code up this analysis unit, and run with `ana_processor`!

7.2.2 Run Your Analysis Unit: Compiled Executable

In the generated package, you should see `bin` subdirectory. There, you find a `GNUmakefile` and `example.cc` (and also other things, but they are not relevant for this discussion). Simply try:

```
> cd bin
> make
```

This should compile `example.cc` and you should get a `C++` executable `example`. You can run this executable with an input file to run your analysis unit. Take a look at example.cc for an example of how to run `ana_processor` with your analysis unit. If you want to quickly test, you can use an output of `DataFormat` test routine, `simple_write`, found under DataFormat/bin directory.

In future you might want to add anohter executable to be compiled here. Open your bin/GNUmakefile, and simple append your program name under the variable `PROGRAMS`.

To be more exact, here is how you can do this. Let's say I want to compile my executable, `kazu.cc`. You place this source code under `bin` directory, and change the following line (I think line 18):

```
PROGRAMS = example
```

to

```
PROGRAMS = example kazu
```

and type "make". You will have an executable `kazu` in the same directory if compilation is successful.

7.2.3 Run Your Analysis Unit: PyROOT

Like example.cc introduced in the previous subsection, there is an equivalent program written in python using PyROOT interface to import C++ libraries. Under your package, you should find mac directory which contains example.py.

This is an equivalent of bin/example.cc but written in python . You need a LArLight sample ROOT file to run this program just like example. If you have a sample file called trial.root, you can run the program as follows.

```
> python mac/example.py trial.root
```

7.2.4 Expanding Your Analysis Package

Like mentioned earlier in Sec.5, the design of analysis unit encourages you to write each analysis unit with a specific purpose. But that does not mean you cannot have more than one analysis unit per package (else you will have too many packages). Feel free to add more analysis unit and/or generic C++ classes to your package! There are handy scripts to do this for you as shown below.

Adding Analysis Unit to Your Package

There is a handy script to generate your new analysis unit under your bin directory. Say I want to generate a new analysis unit called "MyAnotherUnit" under my package, "MyAna". I can do:

```
> cd $MAKE_TOP_DIR/UserDev/MyAna
> python bin/gen_newana MyAnotherUnit
```

Now I can find MyAnotherUnit.hh and MyAnotherUnit.cc created under MyAna package. Also, if I look carefully, I see LinkDef.h and MyAna-TypeDef.hh are modified to include a new analysis unit. So I can just re-compile my package:

```
> make -j2
```

and my new analysis unit, MyAnotherUnit, is available just like the original analysis unit.

Adding Generic C++ Class to Your Package

Analysis unit is just an interface to storage_manager, a data handling I/O class, through ana_processor. Sometimes you want to generate a completely generic C++ class for very good reasons: to develop some algorithm that is independent of the framework (= easy portability to outside).

Here is a script for you:

```
> cd $MAKE_TOP_DIR/UserDev/MyAna
> python bin/gen_empty_class MyEmptyClass
```

Now you should see MyEmptyClass.hh and MyEmptyClass.cc created under MyAna package. There also made appropriate modification to LinkDef.h and MyAna-TypeDef.hh so that you can just type:

```
> make -j2
```

to compile your new class. Now develop your awesome algorithm and make it a non-empty class ;)

Chapter 8

Using LArLight in python

If you never ever want to use an interpreter language like python and CINT , then you should really skip it.

Still here? Well, sure, still you don't have to use python and skip this chapter ;) But the author strongly recommends, at least, to learn how to use it. Let us flip the question: why do we use CINT ? Probably only because CINT was the only interpreter language (= a language that is "easy" to use) and ROOT classes are available in CINT . But now ROOT classes are also available in python , and here's why you might want to consider this option.

- python is one of the most popular languages. Knowing it, you can even get a job.
- IMHO it is a better option than CINT because it is
  - far more robust
  - purely object oriented
  - much better documented (ask Prof. Google)
- You can use compiled libraries in python through PyROOT . It means your execution speed can be as fast as compiled executable while flexibility and simplicity as an interpreter language remains same.

Below we go through a few sections to see how one can use LArLight in python .

8.1 python 101

UNDER CONSTRUCTION

8.2 PyROOT : Using ROOT Classes in python

PyROOT is a python interface to ROOT classes. Basically it makes ROOT classes available in your python module, just like ROOT CINT knows about ROOT classes.

Obtain PyROOT

It is recommended to use python 2.7 or later (see Sec.1.1). You can refer to Sec.1.1.2 To install PyROOT . Here we assume you have PyROOT installed. Check your installation in your python session:

```
$> python
>>> import ROOT
>>>
```

If you see an error with above commands, then you don't have PyROOT properly set up.

8.2.1 Creating and Filling a Histogram in PyROOT

Like mentioned above, with PyROOT , you have an access to ROOT classes in python . Here is an example usage:

```
$> python
>>> from ROOT import TH1D
>>> h=TH1D('h','My Histogram; X-axis; Y-axis',100,0,10)
>>> h.Fill(5)
>>> h.Draw()
```

You should see TCanvas popped up with your histogram drawn on the pad. This is equivalent to the following CINT commands:

```
$> root
root[0] TH1D* h = new TH1D(''h'',''My Histogram; X-axis; Y-axis'',100,0,10)
root[1] h->Fill(5)
root[2] h->Draw()
```

Let's take a break, back up and review basic facts here...

- **ROOT classes can be imported into python**
  - As advertised. If you want to import *ALL* ROOT classes, you can try "from ROOT import *" though that will take a little bit longer time to import a whole ROOT classes (takes roughly 1 second).
- **python does not need type declaration**
  - When creating a histogram, "h", in python , we did not specify it is a type of TH1D object. Like C++ auto, python figures out the type of "h" from left-hand-side of "='' operator (i.e. it knows it's TH1D).
- **Everything in python is a pointer (objects are created on HEAP)**
- **python object attributes are accessed by "."**
  - Basically you can replace "→" and "::" in C++ with "."

Especially the fact that we do not need any type declaration is handy. This reduces lots of mistakes that can otherwise happen in CINT where one can cast a pointer to wrong type.

8.2.2 C-array, Reference, and STL Container in PyROOT

Including myself many people encounter an issue with the usage of C-array in PyROOT . For instance this happens when your C++ function takes a double pointer which is treated as an array in the function. Someone who struggled enough with C++ probably come up and tell you: don't write such function. This is because there's no way of knowing the size of provided array inside the function, often causing invalid memory access. So it's not a good implementation for public functions.

Enough comments from me and THAT C++ guy/gal: here's how you can use C-array in python . We take an example of constructing ROOT object TGraph.

```
> python
>>> from ROOT import *
>>> from array import array
>>> x_points = array('d',[0,1,2])
>>> y_points = array('d',[1,2,3])
>>> g=TGraph(len(x_points), x_points, y_points)
>>> g.SetMarkerStyle(22)
>>> g.SetMarkerSize(2)
>>> g.Draw('AP')
```

As you can see, we used array object as if it is a C-array. The instantiation of array takes 2 arguments: 1st is the value type where 'd' specifies a double-precision type while the 2nd argument specifies the length of array and values to be initialized to. The 2nd argument can be a list which is what I did. In above example, x_points is initialized to an array of length 3 (as my input list has length 3) with elements initialized to 0, 1, and 2 in respective order (as those are my input list values). For y_points, I initialized values to 1, 2, and 3. You should see three points from the TGraph drawn on canbas, showing (0,1), (1,2), and (2,3).

Now, what about reference? You can simply provide a python object just like you provide C++ object when using pass-by-reference. But you might have a problem when the reference is a simple data type such as double or int. This is because ROOT has a wrapper on those simple types. You can instead use ROOT provided types. For instance, say I have a function called FillDouble(double &value). You can do:

```
>>> import ROOT
>>> my_value = ROOT.Double(0)
>>> FillDouble(my_value)
>>> print my_value
```

Finally, if you love using STL containers (I do), they are available through PyROOT . Here's how you can do this:

```
> python
>>> import ROOT
>>> my_int_vector=ROOT.std.vector(int)()
>>> my_int_map=ROOT.std.map(int,int)()
```

The equivalent of C++ 's std::vector<int>, which is a specialization of template std::vector class with double type, is ROOT.std.vector(int) in the code above. Then I call a constructor, (), to create an object.

If you have more questions abouve these items, contact the author and he is happy to expand this section. If you are into more cool statistical/mathematical tools, there are extension modules available in python called numpy and scipy. They are used in science research and private sector, and has a very robust and fast implementation of computing-intensive routines, probably better than ROOT .

8.2.3 Creating an Array of Objects

A typical interpreter language like python has a very useful array module that can hold a set of various objects. Here is an example python script to make a list of histograms.

```
> python
>>> from ROOT import *
>>> myHistoArray=[]
>>> for x in xrange(10):
>>> h=TH1D('h%d' % x, 'Histo %d' % x, 100,0,10)
>>> myHistoArray.append(h)
```

Well, this can be done similarly in CINT using std::vector<TH1D*>. So what is great about python ? Remember, python list does not restrict a type of object to be held. This is because python is completely object oriented language. See following:

```
> python
>>> from ROOT import *
>>> myDataCollection = []

>>> h=TH1D('h','Histogram',100,0,10)
>>> g=TGraph(10)
>>> v=TVector3(0,0,0)

>>> myDataCollection.append(h)
>>> myDataCollection.append(g)
>>> myDataCollection.append(v)
```

There is no need of preparing separate container per data type, nor preparing a dedicated data holder class or struct. Some ROOT collection classes can also do a similar thing. But recall python knows each object's type without specifying it. This makes it very easy to access the list element. In above script, by myDataCollection[0], python knows this is TH1D while CINT requires a correct type casting.

Finally, remember python is completely object oriented. So even a class is an object. You can make a list of class as shown below:

```
> python
```

```
>>> from ROOT import *
>>> myClassList = [TH1D, TGraph, TVector3]

>>> myVector = myClassList[2](0,0,0)
>>> myGraph = myClassList[1](10)
>>> myHisto = myClassList[0]('h','Histo',100,0,10)
```

It looks strange, doesn't it? But this feature of object orientation allows very abstract way of writing a code, and becomes handy when writing a driver script (i.e. a code that executes different programs and functions).

... MORE DOCUMENTATION ONGOING ...

8.3 LArLight Classes

Like you can access `ROOT` classes in `python` , LArLight provides a support of `CINT` dictionary generation, which allows you to access LArLight classes from `python` in the very similar way. Let's try to instantiate `storage_manager` object.

```
import ROOT
from ROOT import larlight
my_storage = storage_manager()
```

Easy, right? This section describes why this can be useful for us.

8.3.1 Fast Execution

One downside of using `python` is that it can make your code slow if you do a computationally intensive task, such as running loop like `for`/`while`. Usually we compile our `C++` code to (1) debug mistakes and (2) optimize execution speed.

But if you have compiled your `C++` code in LArLight, and import your compiled code to use it in `python` , the execution speed of your code remains same as the compiled code. A good example is an event loop. Recall `ana_processor::run()` runs a batch event loop. Because it is a compiled code, whether calling this function in `python` through `PyROOT` or writing a `C++` compiled executable to call this function, you get the same speed.

`PyROOT` brings a very handy merging of compiled `C++` libraries with handy scripting language, `python` . This is fruitful and hence supported in LArLight as much as possible.

8.3.2 Easy (possibly dirty) Data Access

DOCUMENTATION ONGOING

Chapter 9

Where is LArLight Data File?

Enough about the framework: where can we find a LArLight format `ROOT` file to play with? There are three options to obtain LArLight format `ROOT` file.

1. Generate your own
2. Convert from LArSoft into LArLight
3. Use existing file

Following sections describe in details about these options.

9.1 Generating LArLight File

Of course, you can generate your own sample `ROOT` file. There is an example code to create a sample `ROOT` file under DataFormat/bin directory. You should find `simple_write.cc` and `test_simple_io.cc`, and those can generate a sample `ROOT` file to play with.

Also you can take a look at it and see how you can generate a data file from scratch.

9.2 Convert From LArSoft to LArLight Data Format

As described in Ch.4, LArLight has its own data format. Event though it is designed to be similar to that of LArSoft, it is not equivalent. So how can we analyze LArSoft data file using LArLight?

There are two possibilities:

- Convert LArSoft data file into LArLight format
- Implement a utility class to interface LArSoft data file

The first point is currently done by using a dedicated LArSoft analyzer module called `DataScanner`. This is described in details in this section.

The second point does not mean we have to introduce `art` framework dependency in LArLight: that would screw up the whole purpose of LArLight and probably almost all

"strength" listed in Ch.2 will be gone. Although this is not yet available, extracting data from LArSoft files without `art` or LArSoft framework seems possible as there is a beautiful working example, `Argo`. It would be great if someone can implement such feature also in LArLight someday...

9.2.1 LArSoft ⇒ LArLight: `DataScanner`

`DataScanner` is a LArSoft module that is maintained @ FNAL git repository:

`https://cdcvs.fnal.gov/redmine/projects/ubooneoffline/repository/kterao/`

0. Prerequisite

As mentioned above, it uses LArSoft. So you have to have LArSoft environment set up and running. In particular, you have to have a shell environment variable `$MRB_TOP` set and pointing to your local MRB repository. If you have this set up, then you can go to **1. Checking Out and Build** `DataScanner` .

If you are not sure how to do this at all, and if you do not mind blindly following the author's suggestion, there is a set of shell scripts to do this for you under LArLight repository (only for FNAL gpvm nodes, though). If you want to know how things work behind the scene (i.e. if you want to learn properly), there is a good guidance available [1].

First, checkout LArLight if you have not done so:

```
> cd /uboone/app/users/$USER
> git clone git@github.com:/larlight/LArLight LArLight
> cd LArLight
> export MAKE_TOP_DIR=$PWD
```

Do not source `setup.sh` yet. That will fail as you do not have `ROOT` set up. Next, try:

```
> source config/setup_mrb_lar.sh
> source config/setup.sh
> make -j4
```

This will build LArLight. It will take a while: gpvm machines are extremely slow (compared to the author's laptop, at least). Next, create a new MRB local repository. You have to specify where to create it. At this time, we will set this to a shell environment variable, `$MY_LARSOFT`:

```
> export MY_LARSOFT=/uboone/app/users/$USER/mrb_larlight_dev
> source config/checkout_mrb_local.sh $MY_LARSOFT
> source config/setup_mrb_local.sh $MY_LARSOFT
```

This should checkout your local mrb repository which is currently empty.

Now you are ready to checkout `DataScanner` . But how can you set up this from next time? The following commands set up your configuration from next time. You can add that to your configuration script:

```
> export MAKE_TOP_DIR=/uboone/app/users/$USER/LArLight
> source $MAKE_TOP_DIR/config/setup_mrb_lar.sh
> source $MAKE_TOP_DIR/config/setup.sh
> export MY_LARSOFT=/uboone/app/users/$USER/mrb_larlight_dev
> source $MAKE_TOP_DIR/config/setup_mrb_local.sh $MY_LARSOFT
```

1. Checking Out and Bulding `DataScanner`

OK, we assume you have gone through prerequisite and you have `$MRB_TOP` shell environment variable set. Here is how you can checkout `DataScanner` .

First, checkout LArLight, configure and compile. See Sec.1 for how-to. Then run the following command:

```
> source config/checkout_mrb_larlight.sh
```

This should create a directory:

```
$MRB_TOP/srcs/larlight/larlight_translator/DataScanner
```

Now you can build your local `mrb` repository:

```
> cd $MY_LARSOFT/build
> source mrb s
> mrb i -j4
```

Make sure, as said above, you have LArLight compiled before compiling `DataScanner` . This is because `DataScanner` library needs to link against LArLight libraries. If you see any compiler error message, feel free to contact the author.

2. Testing `DataScanner` Build

We go over how to configure `DataScanner` , but here is a blind run test to see if the build was successful or not. Go to `DataScanner` directory:

```
> cd $MRB_TOP/srcs/larlight/larlight_translator/DataScanner
```

Let's run a simple 2 muon event generation.

```
> lar -c job/my_prodsingle.fcl
```

If you do see an error, this error is unrelated to `DataScanner` . It is either (1) you have a problem with your LArSoft installation or (2) the author forgot to update this test fcl file. If you can debug and fix it, please do. Otherwise feel free to contact the author.

If the above execution of `lar` goes well, then you should have generated `single_gen_uboone.root`. Next, we try running `DataScanner` to convert TPC waveform into LArLight data format and store.

```
> lar -c job/scanner_tpcwf.fcl -s single_gen_uboone.root
```

If this goes well and you find `larlight_tpcwf.root` output file, you have a working `DataScanner` module!

3. Running `DataScanner`

You have already run `DataScanner` in **Testing `DataScanner` Build** section with a fcl file, `job/scanner_tpcwf.fcl`. If you look at the top of this fcl file, you see that it imports another fcl file, `scanner_base.fcl`. This is the fcl file to make a configuration of `DataScanner` module.

In `job/scanner_base.fcl`, you find there are two kinds of fcl parameters for `DataScanner` module:

- fModName_XXX
- fAssType_XXX

where "XXX" is replaced with LArLight's `enum` , `larlight::DATA::DATA_TYPE`.

The first parameter, `fModName_XXX`, specifies the producer's module name for the data product "XXX". If you have more than one producer for the same data product (e.g. "MCTruth" for GENIE and CRY), you can specify both by separating the module name by ":". So you have an ability to store data products from more than one producer. For instance:

```
fModName_MCTruth: ''largeant:generator''
```

looks for `simb::MCTruth` produced by "largeant" and "generator" and convert them into LArLight data format.

The second parameter, `fAssType_XXX`, specifies the associated data objects to store. The value has to be LArLight's `enum` , `larlight::DATA::DATA_TYPE` in string name. Note that a producer may have stored more than one associated data products. You can specify more than one associated data type to be stored by separating data products by ",". Recall that, you have an ability to store the same data product from more than one producer. Different producer may have different association stored. You can separate a set of associations to be stored per producer by ":".

I know `fAssType_XXX` is a bit complicated. So here's an example. Assume `SpacePoint` is generated by a producer named "kazu" and "david". These two producers used different hit and cluster finder algorithms. Here is an example to configure DataScanner to store the associations:

```
fModName_Bezier: ''kazu:david''
fAssType_Bezier: ''FFTHit,DBCluster:GausHit,FuzzyCluster''
```

If this is still confusing, feel free to contact the author.

Here is a few points to remember:

- If you put an wrong producer name to `fModName_XXX` value, you will get a corresponding LArLight data `TTree` filled with correct number of events. But each event will be empty.
- If you put an empty string to `fModName_XXX` value, then you will not get any corresponding LArLight data stored (better this way to save disk space a bit).
- Association is stored only if a corresponding data product is stored. For instance, if you want to store associations to `FFTHit`, you must have specified `fModName_FFTHit`.

Any question/comment/suggestion? As always feel free to contact the author.

9.3 Use Existing File

There are some common LArSoft data files converted into LArLight format. They can be found under the following directory @ FNAL:

`/uboone/data/users/kterao/perm_sample`

The following list of files is compiled to the best of the author's knowledge.

- Sub-Directory `SingleShowerStudy/FEB182014` contains:
  - `electron/shower_larlight_11.root` ... 0.1 to 1.9 GeV forward-going e^-
  - `electron/shower_larlight_22.root` ... 0.1 to 1.9 GeV forward-going γ
  - `electron/shower_larlight_13.root` ... 0.1 to 1.9 GeV forward-going μ
  - `electron/shower_larlight_2212.root` ... 0.1 to 1.9 GeV forward-going p
  - `electron/shower_larlight_111.root` ... 0.1 to 1.9 GeV forward-going π^0

Chapter 10

FAQ

This chapter is for FAQ and is meant to keep growing.

10.1 Build

List of build related FAQs.

What's the minimum requirements to install LArLight?

- See Sec.1.1.

I installed `ROOT` from macport. Can I still use LArLight?

- Certainly. When you run `config/setup.sh`, it may complain that you do not have `$ROOTSYS` set if you use release v2.1.1 or earlier. A walk around is to set `$ROOTSYS` to some non-zero value and re-run the configuration script. Note that, in such case, you should set `$PYTHONPATH` by yourself to use `PyROOT` (if you want to use `PyROOT`).

I own OSX version above 10.6 but I don't have `clang` . What do I miss?

- If you do not have `clang` , unless you installed additional tools by yourself, you miss `c++11` feature. Contact the author and educate him otherwise!
- One solution is to upgrade your system to OSX 10.9 Mavericks and install Xcode 5.X. The author found Xcode 5.X is fine for code development.
- Another solution is to install either (1) install `lldb` and `clang` version above 3.1 (above 3.5 recommended), or (2) install `g++` version above 4.3 (above 4.6 recommended). Contact the author for help / further assistance to enable `c++11` in LArLight compilation.

Can I use OSX 10.5 or earlier with `c++11` support?

- The author tested it is possible on 10.5.8 with Xcode 3.1.4. Earlier version than this one is not tested. If you wish, please contact the author and he will be willing to help.

– The point is to have either recent-enough version of `lldb` or `g++` . For `g++` , `c++11` features are added from 4.3 (and later versions have more features). The author took another choice and installed `lldb` 3.1 with `clang` 3.1. This was successful but took 5 hours to compile `lldb` 3.1 on his Core 2 duo from 2007.

I have one of OSX 10.6, 10.7, or 10.8 with Xcode 4.X and seeing issues. Help?

– Sure! The author does not have those machines and cannot test. So your help will be essential to support those OS versions. Make sure you have Xcode 4.2 and up: this seems to be necessary to have `clang` . Please contact the author.

I think I installed LArLight. How can I test?

– There are some test routines. See Sec.1.4.

How can I install `PyROOT` ? Do I need to re-compile LArLight?

– See Sec.1.1.2 for a help on installation. No, you do not need to re-compile LArLight.

Is there a class index list, like `ROOT` web page?

– You can generate your own `HTML` file of class index. See Sec.1.3.3.

10.2 Usage

List of usage related FAQs.

OK so I think everything is compiled. What can I "use"?

– You can use what you compiled :) As to what you have compiled, here's a brief list of things that are compiled by default.

* `Base` package ... framework constants and base classes. See Ch.3.
* `DataFormat` package ... data product classes that mimic LArSoft data products + I/O interface with which you can run an event loop. See Ch.4.
* `Analysis` package ... analysis framework, like EDAnalyzer/EDProducer in LArSoft. See Ch.5.
* `LArUtil` package ... *ala* LArSoft utitlity classes, described in Ch.6.

Where can I find a sample LArLight `ROOT` file?

– You have three options: (1) generate on your own, (2) convert from LArSoft, or (3) use existing files. See Ch.9.

How can I convert LArSoft file into LArLight format?

– See Sec.9.2.

How can I generate my own package? Is there a template or something?

– See Ch.7.

How can I run an event loop?

– There are three options to run an event loop.
  * Use a bare `TTree` instance and configure by yourself.
  * Use `storage_manager`. See DataFormat/bin/simple_read.cc for an example.
  * Use `ana_processor`. See Analysis/bin/test_simple_ana.cc for an example.

My file is big but I need only a few information. How can I speed up my event loop?

– You have a switch to cancel reading in each data type, which can dramatically improve your I/O speed. Look up `storage_manager::set_data_to_read` attribute function.

Can I run `ana_processor` to process only one event?

– Yes. Use `ana_processor::process_event` attribute.

Can I store something other than the data products in an output file in an analysis unit? Say my TH1 histogram?

– Yes. See Sec.5.1.

Why you recommend me `PyROOT` more than `CINT` ?

– Because it is more widely used across the world and simply better ;)

10.3 Development

List of development related FAQs.

Who do you mean by "developer"?

– Probably you and everyone who uses LArLight. As asid in Ch.2, LArLight has started as `C++` play ground for summer students. There is more support on writing code than using the existing code. Anyone who generates his/her own package and extend it is a developer!

OK so I want to write my own code. Can I have my working space or something in LArLight?

– Yes, you can have your own package. See Ch.7.

Should I care about compiler's warning messages?

– You really should.

OK I wrote some code. How can I test? Do I have to run an event loop?

- Not necessarily. If your have a specific function to test, why don't you write a few lines of `PyROOT` or `CINT` script and call that function? See Sec.7.1.2.

Need to test my code by running an event loop. How can I get a test LArLight data sample?

- You can either (1) generate your own, (2) convert from LArSoft into LArLight, or (3) use existing files. See Ch.9.

What are the options to "run" my code?

- You can
  * Write your own executable source code and compile it under `bin` directory (see Sec.7.2.2)
  * Write `CINT` or `PyROOT` script (see Sec.7.2.3)

  There is no better or worse in either method. There are different use cases. Either method works with a debugger such as `gdb` or `lldb` though the author has limited experience with `CINT` and strongly recommends `PyROOT` over `CINT` .

I want to develop my code in LArLight but need to transport to LArSoft soemtime in future. Any tip?

- LArLight only depends `ROOT` and `C++` , so you should be able to copy most part of your code except for some portion that uses other LArLight packages you are not exporting to LArSoft. For instance, functions that access LArLight data products. So, when you write your code, keep that in mind and structure your code to decouple a dependency to LArLight data products from the algorithm, the core part of your code. One solution is to define your own data container `C++` struct, copy LArLight data into it, and work on it. When you transport your code to LArSoft, then, you just need to change one function that fills your `C++` struct to use LArSoft data products instead of LArLight.

10.4 Repository

List of repository related FAQs.

I do not need to commit any code. Can I start w/o having `git` account?

- You can. Try:

  ```
  git clone http://github.com/larlight/LArLight LArLight
  ```

  Keep in mind that, however, github account is free and very easy to make. Also you are motivated to commit your code: your code won't break others' installation as long as you work in your area (see Ch.7).

Can I commit my code?

- Yes, of course! Here's gentlemen & ladies' agreement:
  - Do not commit a data file (binary, text, `ROOT` files or any format that contains "data").
  - Do not commit object/library files (files with `*.o` or `*.so` extension).
  - If you alter someone else's package, make sure (s)he is aware. Else, create and work in a separate `git` branch than the main (shared) one.

Does my commit break LArLight?

- If you are committing changes/additions to your own package (see Ch.7), then you won't break LArLight. By default, only minimal components are compiled: `Base` , `DataFormat` , and `Analysis` . Your package is not even included in the default compilation set (hence cannot break compilation chain)!

Does my commit break someone else's code?

- This is possible if someone else is using your code. But hey, if this person wants to keep a snapshot of your code in a history, (s)he can use older version from `git` repository. Keep developing your code ;)

I want to start a new `C++` framework. Can I take LArLight into parts?

- Go ahead! In fact the author has a few levels of framework templates. Feel free to contact him if you think he can help you.

Chapter 11

Releases

The following release versions are made based on major updates and testing of stability for framework base components (i.e. packages under `core` directory).

v2.3.1

- Bug fix in `mcshower`
- Bug fix in `Geometry`
- Bug fix in `ana_processor`
- Put capability to support multiple experiments in `LArUtil`

v2.3.0

- Changed `data_base` to sore association using `unsigned int` instead of `unsigned short`.
- Added `mcshower` object to store MC truth shower object.
- Use `larlight::hit` pointer instead of object in `ClusterParamsAlg`, better processing speed.
- Nice addition by Nathaniel which made `export MAKE_TOP_DIR` unnecessary ;)

v2.2.2

- Introduced ClusterRecoAlg under `UserDev` . Some LArSoft algorithms are transported including HoughBaseAlg, ClusterParamsAlg, and ClusterMergeAlg
- Introduced ShowerAngleCluster in ClusterStudy package under `UserDev`
- Fixed bugs in `LArUtil` (bug in the stored TTree geometry data)
- Fixed minor bugs in `DataFormat` and `Analysis`

v2.2.1

- Introduced `LArUtil` package that contains *ala* LArSoft utility classes. Currently `Geometry`, `LArProperties`, `DetectorProperties`, and `GeometryUtilities`.

– Structural change: main packages (`Base` , `DataFormat` , `Analysis` , `LArUtil`) are now under `core` directory. `AnaProcess` is renamed to `UserDev` and stays as user's development area.

v2.2.0

– Included `tpcfifo` and `pmtfifo` data products from David C.

– Included TriggerSim package for UB trigger algorithm simulation

v2.1.1

– Included Bill's suggestion on the comments in the template source code

– Included empty `C++` class generation script in the bin directory of a user's analysis package

– Included a sample waveform generation code for example code in the manual

v2.1.0

– Added `recob::EndPoint2D` and `recob::Vertex` equivalent data products

– Fixed `bin` directory compilation issue for Ubuntu 12.04LTS

v2.0.0

– Initial frozen release upon presenting in Analysis Tool meeting.

Bibliography

[1] The microboone guide to using larsoft. *FNAL Redmine MicroBoone Wiki*, https://cdcvs.fnal.gov/redmine/projects/uboonecode/wiki/Uboone_guide.