LArLite
Simple LArSoft Data Analysis Tool

Kazuhiro Terao, kazuhiro@nevis.columbia.edu

May 14, 2015

Abstract

LArLite (previously known as LArLight) is a simple `C++` code development + analysis framework based on `ROOT` . The goal of the framework is to provide (1) a light-weight `C++` & `ROOT` code/framework development environment with build support on Linux/Unix platform, (2) analysis framework equipped with *ala* LArSoft data product and utiltiy classes definitions outside LArSoft, (3) a personal computer (laptop) based workstand for building and running (1) and (2) with a reasonably sized data file, and (4) a support to bridge `C++` compiled library and interpreters (`CINT` /`python`).

LArLite originally started as a simple `C++` code play ground for Columbia Summer students in 2013 with (1), (3) and (4). Then it is extended to support (2). It has been used for high level data analysis, reconstruction algorithm development, LArSoft code development, and even for developing other frameworks. This manual describes what LArLite is in details and how to use it.

One of design principles of LArLite remains same sicne the beginning: to provide a flexible and comfortable code development environment. Please feel free to use the tool for any purpose, and contact the author for requesting new features or fixing bugs any time.

Contents

Chapter 1

Installation

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

1.1 Prerequisite

LArLite 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 LArLite:

- 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`
  - For ROOT5, v5.34.08 or later
  - For ROOT6, v6.02.05 or later

If your system cannot 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 LArLite. It's strongly recommended to have resources noted in the following subsection so that you can use:

- `C++11` features

- `PyROOT`
- Analysis on LArLite files converted from LArSoft.

1.1.2 Recommended Resources

Following features are strongly encouraged:

- Operating Systems & Compilers ... to use `C++11` features
  - Mac OSX 10.9.1 (Darwin 13.0) or later with `clang` 3.3 or later
  - Linux 3.11 or later with `g++` 4.3 or later
- `python` 2.7 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` v5.34.18 or later with `PyROOT`
  - This version seems compatible with LArLite files created by LArSoft’s `ROOT` process.
- `doxygen`
  - LArLite includes `doxygen` script + uses `doxygen` style comment all the places

Please note there are dependencies among items listed above. In particular, LArLite 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 LArLite.

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

Nothing serious. Please refer to FAQ in Sec.10.1.

Why `PyROOT` ?

Simply because `python` is

1. Far more easier and stable (industrial level) to use compared to `CINT`
2. Real object oriented interpreter language with great popularlity
3. Full access to `ROOT` and LArLite 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 it!).

1.2 Checkout Repository

LArLite is maintained at the github.com public `git` repository:

`https://github.com/larlight/larlite`

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.

Watch out of the URL: it is larlight/larlite, not larlite/larlite!

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 Getting Commit Emails

`github.com` now supports commit email messages! Awesome! I don't need my dirty cron job anymore!

https://help.github.com/articles/configuring-notification-delivery-methods/

1.2.3 Checking Out

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

```
> git clone git@github.com:/larlight/larlite MyLArLite
```

where "MyLArLite" 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/larlite MyLArLite
```

but you cannot commit your code as mentioned already.

You might jump ahead and say "hey my git clone failed! I see nothing under directory MyLArLite!". No worries. Try:

```
> ls -la MyLArLite
```

and you should see ".git" which makes any directory a git repository (i.e. you checked out alright).

Checking Out Branch

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

```
> cd MyLArLite
> git checkout trunk
```

Fixed Releases

You can also checkout a fixed release.

```
> cd MyLArLite
> 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 `MyLArLite/core` directory created. This is the core LArLite package you will build regarding LArSoft data analysis.

1.2.4 What's in there?

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

- `core`
  - This directory contains the core part of LArLite framework including `Base` , `DataFormat` , `Analysis` , and `LArUtil` . Built by default (see Sec.1.4).
- `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.1.4).
- `bin`
  - Where various utility `python` run scripts live. Most of them are to generate a new package/class template for code development.
- `python`
  - Where various `python` modules live.
- `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.1.4 for how to use it).

1.3 Configure

There is a configuration script to build and/or use LArLite:

```
config/setup.sh
```

which sets numbers of environment variables and aliases useful/necessary for LArLite. This script should be sourced each time a user logs into a machine before using/building LArLite. One can source this script multiple times per log-in. If you have multiple LArLite installation and would like to switch one to be used, simply call this script from the top directory of the one you wish to configure.

1.3.1 Shell Environment Variables

In particular, by sourcing this script, the following shell environment variables will be set:

- `$LARLITE_BASEDIR` is set to the top directory in the repository
- `$LARLITE_LIBDIR` is set to the `$LARLITE_BASEDIR/lib`
- `$LARLITE_COREDIR` is set to the `$LARLITE_BASEDIR/core`
- `$LARLITE_USERDEVDIR` is set to the `$LARLITE_BASEDIR/UserDev`
- `$LD_LIBRARY_PATH` is prepended with `$LARLITE_LIBDIR`
- `$DYLD_LIBRARY_PATH` is prepended with `$LARLITE_LIBDIR`
- `$LARLITE_CXX` is set to either `clang` or `g++` w/ `C++` 11 flag

1.3.2 Aliases

In addition, some useful aliases are set:

- `cdtop` is "`cd $LARLITE_BASEDIR`"
- `maketop` is "`make --directory=$LARLITE_BASEDIR`" to which you can give an option flag such as "`-j4` to compile w/ 4 threads
- `llgen_repository` is "`python $LARLITE_BASEDIR/bin/gen_repository`" for generating a new user repository (see Ch.7).
- `llgen_package` is "`python $LARLITE_BASEDIR/bin/gen_package`" for generating a new user package under a specific repository (see Ch.7).
- `llgen_class_empty` is "`python $LARLITE_BASEDIR/bin/gen_class_empty`" for generating a new `C++` class under a package (see Ch.7).
- `llgen_class_anaunit` is "`python $LARLITE_BASEDIR/bin/gen_class_anaunit`" for generating a new analysis unit `C++` class under a package (see Ch.5 and 7).
- `llgen_class_eralgo` is "`python $LARLITE_BASEDIR/bin/gen_class_eralgo`" for generating a new `ERTool` algorithm class under a package (see Ch.7 and `ERTool` documentation).
- `llgen_class_erfilter` is "`python $LARLITE_BASEDIR/bin/gen_class_erfilter`" for generating a new `ERTool` filter class under a package (see Ch.7 and `ERTool` documentation).
- `llgen_class_erana` is "`python $LARLITE_BASEDIR/bin/gen_class_erana`" for generating a new `ERTool` analysis class under a package (see Ch.7 and `ERTool` documentation).

1.3.3 `USER_MODULE`

By default, sourcing `config/setup.sh` will configure to build the LArLite `core` package, which includes LArLite's framework constants, data format, and analysis toolkit. In addition, one can specify various directories under `UserDev` to be built by specifying then in `USER_MODULE` shell environment variable **prior to** sourcing the script. For instance:

```
> export USER_MODULE=‘‘BasicTool’’
> source config/setup.sh
> make -j4
```

will configure to build the `core` and `BasicTool` under `UserDev` directory.

1.4 Build

1.4.1 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:

```
> 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. The execution of this shell script sets necessary environment variables to build and use LArLite. In particular, it might be handy to remember aliases set by this script (see Sec.1.3 for details).

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 1 minute on the author's MacBook Pro (OSX 10.9.3) from scratch. 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 `libLArLite_Base.so, libLArLite_DataFormat.so, libLArLite_Analysis.so, libLArLite_LArUtil.so`. These are four building blocks of the analysis framework and will be described in the later section of this document.

1.4.2 Building UserDev

I hope you will eventually write your own package and enjoy coding, and `UserDev` is the place for you. This directory `UserDev` is meant to hold a users' (or possibly multiple) sub-repository which could be, for instance, a `git` repository.

For the sake of tutorial, let us use my example. Try:

```
> cd $LARLITE_USERDEVDIR
> git clone https://github.com/drinkingkazu/example Example
```

This should have created `Example` sub-directory under your `UserDev` . This is what I mean by users' sub-repository.

This newly created `Example` repository is, althogh it is under `UserDev` , not recognized as LArLite repository because of `.gitignore` rule placed in LArLite top directory. By default, any newly created items under `UserDev` is not a part of LArLite repository because of `.gitignore` rule of LArLite. Exceptions are those items you see under `UserDev` from the beginning. Those are sub-repositories that became popular enough amonge multiple users, and agreed to put in a shared LArLite repository instead of a single user's `git` repository.

How to create your own package is covered in Ch.7.

There are two ways to build sub-repositories under `UserDev` . A simple method is simply type "make" in that sub-repository:

```
> cd Example
> make
```

You may not want to do this, however, if you have multiple sub-repositories (simply troublesome). Instead, you can combine a compilation of chosen sub-repositories to the LArLite build. Here's what you can try:

```
> cd $LARLITE_BASEDIR
> export USER_MODULE=''Example''
> 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. Multiple sub-repositories can be specified by separating them with a space. 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.

1.4.3 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 $LARLITE_BASEDIR
> make doxygen
```

You can open the top page using firefox:

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

or, on Mac OSX,

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

1.5 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(``libLArLite_Base'');
0
root[1] gSystem->Load(``libLArLite_DataFormat'');
0
root[2] gSystem->Load(``libLArLite_Analysis'');
0
root[3] gSystem->Load(``libLArLite_LArUtil'');
0
root[4]
```

Each line above tells `CINT` to load compiled shared object libraries. They can be physically found under `$LARLITE_LIBDIR` 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.

1.5.1 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.

1.5.2 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 LArLite `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 LArLite `ROOT` file with a track data product. You can use an output of `simple_write` routine to try out.

1.5.3 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 LArLite `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.

1.5.4 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

LArLite 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 LArLite

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 LArLite, please (1) fix it if you can or (2) speak up and ask to make it better!

2.2 What Is LArLite 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 LArLite support these points. After those, IMHO the strength of LArLite is described.

It is important to note here the author has no intention to replace LArSoft by LArLite. 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 LArLite to develop a generic `C++` + `ROOT` project. This is just executing a one line command to generate your package, and type "make". See Sec.7.2 for an example.

2.2.2 For ala LArSoft data analysis

LArLite 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 LArLite `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 LArLite 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 LArLite to process data of our interest.

2.2.3 Strength: Why Using LArLite?

IMHO LArLite 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 LArLite 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.

LArLite 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

LArLite 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 LArLite in that sense. The purpose of LArLite 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, LArLite is suitable for accessing full detail of information outside LArSoft. In otherwords, one can produce `AnalysisTree` from LArLite. 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` ... `sim`, `simb` namespace constants copied from LArSoft

`RawConstants.hh` ... `raw` namespace constants copied from LArSoft

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

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 kINVALID_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 LArLite file if you copied data contents from LArSoft.

Data Type: `enum data::kDataType_t`

Probably most important `enum` . This defines a set of data types in LArLite. 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 kDATA_TREE_NAME[]`

This is an array of `std::string` to define a corresponding name for each `data::DataType_t`. The length of the array is, therefore, `data::kDATA_TYPE_MAX`. This is used to name `TTree` stored in LArLite `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, LArLite needs these constants, too. Both constants name and values are exact copy from LArSoft to avoid a confusion. Copied constants include:

`simb::Origin_t` ... specifies particle generator type

`simb::curr_type` ... specifies neutrino interaction current type

`simb::int_type_` ... specifies neutrino interaction categories

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.

3.2 Framework Base Class

There are two basic classes used throughout the framework: (1) `larlite_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: `larlite_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 `larlite_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 `larlite::` namespace specification to LArLite 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 LArLite 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.

4.1.1 Classification of "Data" Type

There are three types of data:

- event-wise data ... `enum larlite::DataType_t`
- run-wise data ... `enum larlite::RunDataType_t`
- subrun-wise data ... `enum larlite::SubRunDataType_t`

Each `enum` listed above contains further "data type" that belongs to each classification above. As you might guess, above classifications correspond to the unit data: there can be more than one event in a sub-run, and there can be more than one subrun in one run. An event-wise data is stored at each event boundary, and run/subrun-wise data are stored at each run/subrun boundary.

4.1.2 Data Products

Here is a list of LArLite data products (left) with corresponding LArSoft data product (right) as of now.

```
larlite::gtruth <==> simb::GTruth
larlite::mcflux <==> simb::MCFlux
larlite::mctruth <==> simb::MCTruth
larlite::mcnu <==> simb::MCNeutrino
larlite::mctrajectory <==> simb::MCTrajectory
larlite::mcpart <==> simb::MCParticle
larlite::mcshower <==> sim::MCTruth
larlite::simch <==> sim::SimChannel
larlite::rawdigit <==> raw::RawDigit
larlite::wire <==> recob::Wire
larlite::hit <==> recob::Hit
larlite::seed <==> recob::Seed
larlite::cluster <==> recob::Cluster
larlite::spacepoint <==> recob::SpacePoint
larlite::track <==> recob::Track
larlite::shower <==> recob::Shower
larlite::endpoint2d <==> recob::EndPoint2D
larlite::vertex <==> recob::Vertex
larlite::calorimetry <==> anab::Calorimetry
larlite::partid <==> anab::ParticleID
larlite::pfpart <==> recob::PFParticle
larlite::pcaxis <==> recob::PCAxis
larlite::potsummary <==> sumdata::POTSummary
larlite::user_info ... LArLite-only custom data product
larlite::event_ass ... LArLite-only custom data product
```

Additional data products, `user_info` and `event_ass`, are described later.

All data product class shares the base class `larlite::data_base` which stores only one variable: `larlite::data::DataType_t`. This `enum` value specifies the type of data stored. For instance, `larlite::hit` data product stores a type `larlite::data::kHit`.

4.1.3 What is stored per event?

Despite a few exceptions (which I describe next), each data product class `T` has a corresponding event-wise collection class named as `event_T`. For instance, in `cluster.h`, you find there is a class `larlite::cluster` and `larlite::event_cluster` where the latter is a type `std::vector<larlite::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 `event_base` which carries additional inforamtion:

- `unsigned int` Event number

- `unsigned int` Run number
- `unsigned int` Sub-Run number
- `product_id` Data product ID

Just like everything else, `event_base` inherits from `data_base` and hence knows its own data product type. Together with the producer's name string, this forms a unique data product identification instance called `product_id`. For instance, type `data::kHit` made by "gaushit" forms a specific data product ID. `product_id` is a class defined under:

```
core/DataFormat/larlite_dataformat_utils.h
```

and is simply `std::pair<data::DataType_t,std::string>` to hold such data product identity.

It should be noted *most of* `larlite::event_T` inherits from `std::vector<T>` with an exception of `event_ass`, the association data product. In fact there is no such data product called `ass` (of course, it's inappropriate, right?). More details about this data product is described later in Sec.4.1.5.

Exception to `event_T`

Exceptions to this event-wise collection storage scheme include `potsummary`, `mcnu`, and `mctrajectory`. A reason for `potsummary` is simple: this is a sub-run data type and not an event-wise data type. In LArSoft, `simb::MCNeutrino` is stored as a part of `simb::MCTruth`. Likewise, `simb::MCTrajectory` is stored as a part of `simb::MCParticle`. LArLite respects this scheme and stores `mcnu` in `mctruth` and `mctrajectory` in `mcpart`.

In case you are unfamiliar with these MC information, though this is a side-track, `mcnu` stores neutrino specific information from neutrino generator. It is a part of `mctruth` because it is created by a generator in general. Similarly, `mctrajectory` is part of `mcpart` because a particle's trajectory points (that is what `mctrajectory` is) belong to a particle information which is `mcpart`.

More useful info to store?

If you have some information you think is very useful, let's store it :) Your ideas can help others. So please share it!

Adding new data product?

By all means, go ahead. 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.

4.1.4 Dynamic Data Container ... `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.

4.1.5 Association

The author has little idea about how exactly association is stored in `ART` . But that does not matter.

As you have seen in the previous section, all event-wise data products are stored in terms of `std::vector` collection. So ultimately what you need to associate one object (A) to another (B) are:

1. `product_id` for A and B
2. A set of indexes (A) that point to the right `std::vector` index (B).

When referring an association of A to B, information such as 2 above can be represented as a two-dimensional array of integers. In LArLite, 2 is represented by `AssSet_t` defined as:

```
typedef std::vector<std::vector<unsigned int> > AssSet_t;
```

The association data product, event_ass, stores AssSet_t for a unique combination of product_id pair. Note that there is no limitation on how many of such association to be stored in this data product. Also, two separate instance of event_ass may store a different association information for an identical product_id pair, although that rarely happens and the author has never seen in a real practice.

Accessing a Stored Association

Retrieving association is as simple as calling the following function:

```
event_base::association(const& product_id A, const& product_id B)
```

You provide a pair of product ID, A and B respectively, and you receive an association of A=¿B.

For instance, here's a code snipset to loop over associated hits for clusters to calculate charge sum per cluster and fill a histogram.

```
auto clusters = storage->get_data<event_cluster>(``fuzzycluster''); // Get clusters
auto hits = storage->get_data<event_hit>(``gaushit''); // Get hits
auto ass_data = storage->get_data<event_ass>(``fuzzycluster''); // Get association

// Get association cluster => hit
auto const& cluster_to_hit_ass = ass_data->association(clusters->id(),hits->id());

// Loop over associated groups of hits
double cluster_q=0;
for(auto const& hit_indices : cluster_to_hit_ass)
  cluster_q = 0;
  for(auto const& hit_index : hit_indices)
  cluster_q += (*ev_hit)[hit_index].Charge();
  h->Fill(cluster_q);
}
```

where storage is a storage_manager class instance, and h is a TH1D histogram instance.

It might be also helpful to remember a simple command to dump association information:

```
ev_cluster.list_association()
```

will output stored association information to the text output stream.

Handy Utilities to *Find* Associations

event_ass::association function described above requires a user to provide two product IDs. Sometimes (actually quite often) you want to "search" an association by providing one product ID and a type of the other. For instance, if I am analyzing a cluster data product made by "fuzzycluster", I may not necessarily know the producer name of an associated hit data product. In that case, all I know for sure is "I want an associated *hit*."

event_ass implements a few functions to allow you a search with such partial product ID pair information. Those are:

```
AssID_t event_ass::find_one_assid(const T& a, const T& b);
AssID_t event_ass::find_unique_assid(const T& a, const T& b);
std::vector<AssID_t> event_ass::find_all_assid(const T& a, const T& b);
```

These are templated functions, and one of two input argument has to be product_id while the other can be also product_id or simply data::DataType_t, a data product type. The return, AssID_t is simply an unsigned integer that identifies a specific product ID pair and the corresponding association information that can bre retrieved by the following functions:

```
/// returns a pair of product ID A and B
std::pair<product_id,product_id> event_ass::association_keys(AssID_t);
/// returns AssSet_t for a specified association info
AssSet_t event_ass::association(AssID_t);
```

Note that a function find_one_assid returns *any* one of associations that matches with input argument information. That means, if there are more than one type of association that matches your request, you do not have a control on which one to be returned (although that is, again, very rare and the author has never seen such use of association data product). If you wish to ensure the returned association corresponds to a unique pair of product IDs, you may want to use find_unique_assid although this method takes more time to ensure the uniqueness of returned association information. Finally, if you wish to obtain *all* matched association information, you can use find_all_assid. But again, so far always one association data product stores (possibly multiple) unique pairs of product IDs.

More practical usecase of association data product usage is described in the later analysis section, and the author suggests to read there before you write a code to retrieve/use association information.

Please contact the author to ask more questions or suggestions to expand this section.

4.2 ROOT file I/O interface

LArLite uses ROOT file as storage because the author is not smart enough to write his own streamer. In a LArLite ROOT file, you find TTree per stored data type (i.e. larlite::data::DataType_t) and per data product producer's name. A standard approach of using TTree::SetBranchAddress works very easily, but LArLite 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 LArLite ROOT file structure

In LArLite ROOT file, TTrees are created per stored data type and producer's name. You find, for instance, "hit_gaushit_tree" if larlite::data::kHit data type is stored by a producer "gaushit".

Accessing "by hand"

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

```
> root
root[0] TFile* f = TFile::Open(``sample.root'',``READ'');
root[1] TTree* t = (TTree*)(f->Get(``hit_gaushit_tree''));
root[2] 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 `larlite::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] TFile* f = TFile::Open(``sample.root'',``READ'');
root[1] TTree* t = (TTree*)(f->Get(``hit_gaushit_tree''));
root[2] larlite::event_hit* my_hit_v = new larlite::event_hit;
root[3] t->SetBranchAddress(``hit_gaushit_branch'',my_hit_v);
root[4] t->GetEntry(0);
root[5] my_hit_v->size();
```

But this is tedious and the author personally hates it. Easier method is to use PyROOT:

```
> python
>>> from ROOT import *
>>> ch = TChain(`hit_gaushit_tree',`')
>>> ch.AddFile(`sample.root')
>>> ch.GetEntry(0)
>>> ch.hit_gaushit_branch.size()
```

As you can see, significantly less amount of typing.
Doing this for all Trees and keeping track of entry on each Tree is, however, very tedious. In addition, accessing data event by event interactively is very time consuming (interpreters are not meant to be fast). If you want a fast way to access, you can compile an executable to loop over your TTree. But if you want a fast *and* easy way, you can use LArLite use I/O interface class.

4.2.2 Framework I/O Interface: `storage_manager`

`storage_manager` is the LArLite'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 `larlite::data::kHit` type data product produced by "gaushit" producer.

Simple Example: Opening an input file

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

```
{
    larlite::storage_manager mgr;
    mgr.set_io_mode(mgr.kREAD); // Option: kREAD, kWRITE, kBOTH
    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.

```
{
    larlite::storage_manager mgr;
    mgr.set_io_mode(mgr.kREAD); // Option: kREAD, kWRITE, kBOTH
    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 larlite::event_hit* my_hit_v = mgr.get_data<larlite::event_hit>(``gaushit'')
      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;
}
```

Note you are still loading event interactively, so this method is not very fast unless you compile the code.

Simple Example: Event loop (Read & Write)

This is an example to run an event loop.

```
{
    gSystem->Load(``libLArLite_DataFormat'');
    larlite::storage_manager mgr;
    mgr.set_io_mode(mgr.kREAD); // 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 larlite::event_hit* my_hit_v = mgr.get_data<larlite::event_hit>(``gaushit'')
      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.

4.3 Accessing Association Information

This section describes some practical usage of an association information that is initially discussed in Sec.4.1.5. Note that not everyone needs to know/learn about association data product. If you don't have a plan to use it yet, the author recommends you to skip this section and come back when you need to learn it. It's cumbersome. Trust him.

OK fine, you really need it. Let's take a simple example case of accessing a `cluster` data product produced by "fuzzycluster". Say we want to access associated `hit` data product, which happened to be produced by "gaushit" with an association information The following is an example code to do such thing:

```
// get cluster
auto clusters = storage->get_data<event_cluster>(``fuzzycluster'');
// get hit
auto hits = storage->get_data<event_hit>(``gaushit'');
// get association
auto ass_data = storage->get_data<event_ass>(``fuzzycluster'');

// retrieve cluster=>hit association info
auto const& ass_info = ass_info->association(clusters->id(), hits->id());
```

There are two typical complaints to above lines of code: 1) too many letters to type, and 2) a user has to know a producer name of `hit` data product. The latter is particularly cumbersome because the code does not work if a user provided an wrong hit producer name. So the bottom line is this is terrible.

We clearly want an ability to ask: "here's my cluster, give me associated hits." In other words, by a user specifying an associated data product type (i.e. `hit`), we want the utility to go look for an associated information. You might remember a similar functionality existed in `event_ass` data product, namely `event_ass::find_one_assid`. Yes, you could do this:

```
// get cluster
auto clusters = storage->get_data<event_cluster>(``fuzzycluster'');
// get association
auto ass_data = storage->get_data<event_ass>(``fuzzycluster'');

// Search association I want ... clusters=>hit
auto ass_id = ass_data->find_one_assid(clusters->id(),data::kHit);
// Get association I want
auto const& ass_info = ass_data->association(ass_id);
// Get hit data product
auto hit_product_id = ass_data->association_keys(ass_id).second;
auto hits = storage->get_data<event_hit>(ass_id.second);
```

Great! This approach ended up 0) more letters to type, and also 1) longer time to process as the approach involves a "search" for a pair of product IDs that matches my request.

Please do not throw away your laptop (or something to the author's office) just yet. Let's first approach to reduce number of letters to type. For this, we introduce a function `storage_manager::find_one_ass`.

```
// get cluster
auto clusters = storage->get_data<event_cluster>(``fuzzycluster'');

// Retrieve hit data product & association at the same time
```

```
event_hits* hits = nullptr;
auto const& ass_info = storage->find_one_ass(clusters->id(), hits);
```

Now the number of letters you have to type is dramatically reduced. But let's be careful and make sure we understand what we are doing. First of all, the idea is still the same: you provide key information for a utility to search an association that corresponds to a specific pair of product IDs, A and B. The product ID of A is given by the first argument, which is `clusters->id()`. The *type* of product ID B is identified as `hit` because you provided `event_hit` type pointer in the second argument. The return of the function is a matched association's `AssSet_t`, and when this return is valid the pointer `hits` is pointed to the associated data product. Prior to this function call, `hits` variable must be a null pointer.

So this all sounds good. Note that this function call is very similar to `event_ass::find_one_assid`, and indeed `storage_manager` is internally using this function to go look for a specific product ID pair. So how does `storage_manager` knows *which* `event_ass` data product to look for requested product ID pair? The answer is *it does not know*, and hence perform a linear search in *all* `event_ass` data products that exist in the input file. This means you potentially read lots of `event_ass` data products from a file, and hence causing a huge I/O overhead.

The above mentioned problem can be avoided if you can provide a "producer name" of a specific `event_ass` data product, which is something you usually know. In our example, an association from `cluster` to `hit` is created by the same producer as the `cluster`. In such case, you can specify which `event_ass` to search for an association you requested:

```
// get cluster
auto clusters = storage->get_data<event_cluster>(‘‘fuzzycluster’’);

// Retrieve hit data product & association at the same time
event_hits* hits = nullptr;
auto const& ass_info = storage->find_one_ass( clusters->id(),
                                              hits,
                                              clusters->id().second );
```

As you can see, now the third argument is provided in the `find_one_ass` function which is a string specifying `event_ass` producer name (which is `cluster` producer name in this case). This last example runs significantly faster than the previous one.

Finally, we should know the actual timing optimization scale we are talking about here. A `cluster` to `hit` association is one of the heaviest simply because there are many objects (i.e. hits) to make an association. A typical time to load this association in a busy events (like cosmics) is a few hundreds of micro-seconds. If you are running a computation that might take more than that per event, you might find such timing improvement rather negligible. Just always know whether it makes sense to perform an optimization or not.

I hope this section described a practical usecase of association data products well enough. If you think there can be more utility methods to be added, please let the author know! Also if you need more examples, again just ping the author!

Chapter 5

Core Package: Analysis

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

- `AnalysisUnit`
- `AnaProcessor`

The first bullet corresponds to LArSoft's "module" (i.e. `art::EDAnalyzer/EDProducer`). The second bullet is an executor of `AnalysisUnit` (s) by interfacing with `storage_manager` (i.e. LArLite 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 AnalysisUnit

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

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

In addition, there are two functions recently added and might be useful for some of your analysis.

```
bool ana_base::begin_run(storage_manager* data);
bool ana_base::begin_subrun(storage_manager* data);
```

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

5.1.1 Flow of Function Calls

`initialize()`

The attached `AnalysisUnit` 's `initialize` function is called only once at the beginning of an event loop. If the `AnalysisUnit` 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 has a meaning if one enables "filter mode" for running `ana_processor`. See Sec.5.2.2 for details.

`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.

`begin_run(storage_manager* storage)`

This function is called at each run boundary. In particular it is called before `begin_subrun` and `analyze`.

`begin_subrun(storage_manager* storage)`

This function is called at each sub-run boundary. In particular it is called after `begin_run` if it is concurrently a run boundary, and before `analyze`.

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 `AnalysisUnit` 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 `AnalysisUnit` 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 `AnalysisUnit` , 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 `AnalysisUnit` does one thing. It does not mean one `AnalysisUnit` 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 `AnalysisUnit` 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 AnaProcessor

AnaProcessor runs an event loop and is responsible for executing AnalysisUnit (s). You can attach as many AnalysisUnit (s) as you wish. Those AnalysisUnit (s) 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 AnalysisUnit (s)
- Start event loop via storage_manager
- For each event, call analyze of AnalysisUnit (s)
- Call finalize of AnalysisUnit (s) at the end of an event loop
- Close I/O

Before the 1st bullet point, a user has to (1) configure AnaProcessor with file I/O information and (2) attach AnalysisUnit (s).

5.2.1 Example: Running AnaProcessor

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

```
{
    gSystem->Load(``libLArLite_Analysis'');
    // Create ana_processor instance
    larlite::ana_processor my_proc;
    // Set I/O mode, like storage_manager.
    my_proc.set_io_mode(larlite::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
    larlite::KazuAna* k = new larlite::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 `AnaProcessor` is using `storage_manager` underneath.

After the I/O configuration, like described before, you attach `AnalysisUnit` (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.

5.2.2 Filter Mode

`AnaProcessor` has a feature called "filtering mode". This can be enabled by a function:

```
    ana_processor::enable_filter(bool doit=true)
```

When enabled, attached `AnalysisUnit` 's `analyze()` function return value causes to abort the execution of the rest of `AnalysisUnit` (s) in an event. Let's say we attached multiple `AnalysisUnit` (s) and enabled a filter mode. `AnalysisUnit` (s) are executed in the order we attach them to an `AnaProcessor` instance. If filter mode is enabled and if any one of `AnalysisUnit` returns false, `AnaProcessor` aborts the execution of the rest of `AnalysisUnit` and move onto the next event. It does not mean to filter out the subject event from an output file (which is a feature that can be easily implemented if desired... let the author know).

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 LArLite, 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 `larlite` 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(``libLArLite_LArUtil'')
root[1] larutil::Geometry::GetME()
```

or in `PyROOT`

```
> python
>>> from ROOT import *
>>> gSystem.Load(``libLArLite_LArUtil'');
>>> 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 LArLite 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(``libLArLite_LArUtil'')
root[1] larutil::Geometry::GetME()->Nchannels()
(const unsigned int) 8254
```

Or maybe getting the position of PMT channel 1:

```
> root
root[0] gSystem->Load(``libLArLite_LArUtil'')
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(``libLArLite_LArUtil'')
root[1] Double_t xyz[3]={0.}
root[2] larutil::Geometry::GetME()->View(5000)
(const enum larlite::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(``libLArLite_LArUtil'')
root[1] larutil::LArUtilManager::Reconfigure(larlite::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(``libLArLite_LArUtil'')
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 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 code development space. In particular following itmes are discussed in the respective order.

- Getting started: your own code repository
- A simple `C++` project package
- Stuffing your package
  - Simple `C++` class
  - `AnalysisUnit` (see Ch.5).
  - `ERTool` algorithm class (see `ERTool` documentation)
  - `ERTool` filter class (see `ERTool` documentation)
  - `ERTool` analysis class (see `ERTool` documentation)
  - `C++` functions
- Details: LArLite build basics
- Advanced Code Development
  - `C++` functions outside classes
  - Inter-package dependence
  - Data product: storing `C++` class instance in `ROOT` file
  - `python` in `C++` (i.e. opposite of `PyROOT`)

7.1 Creating Your Repository

LArLite supports user code development under `UserDev` . More precisely, it is assumed to be under any path that is set to the value of shell environment variable `$LARLITE_USERDEVDIR`. For the sake of simplicity we stick with `UserDev` in this section.

7.1.1 Note About `UserDev`

Important note first:

- `UserDev/GNUmakefile`, if it exists, is generated by setup.sh and it does not belong to LArLite repository (ignore it).
- Some sub-directories belong to LArLite (see below). Some of these depend on LArLite and some don't. It is useful to note them briefly here.
  - `BasicTool` contains useful packages like `GeoAlgo` and has no dependency outside
  - `SelectionTool` contains useful packages like `ERTool` and depends on `BasicTool`
  - `RecoTool` contains shower reconstruction code and depends on `core`
  - `LArLiteApp` contains LArLite application from `GeoAlgo` and `ERTool`
- You can create new directories under `UserDev` and that won't bother other users. We'll do this below. This is all done through `.gitignore` rule placed under the top directory.

These features of `UserDev` are there so that you feel more free to make a mess (sorry, I meant, to develop code) there!

7.1.2 Creating Your Sub-Repository in `UserDev`

Obviously I cannot just say "do whatever under `UserDev` " and leave: I would love to support a very easy way to develop code (or make a mess) under `UserDev` . I will just show you how to do things here:

```
> llgen_repository MyRepo
```

where the execution command is an alias explained in Sec.1.3. This should create a new directory called `MyRepo` under `UserDev` . That is your new repository. As said, this does not affect LArLite repository. If you don't want to keep `MyRepo`, simply "`rm -r`" it.

Another thing to note here: your repository will contain code and you will compile them, making a compiled shared object library. Your repository name will be used to name your library file (if you follow LArLite code generation scripts described in the followings). So pick a name that suits for your purpose. You do not want to pick a generic name that might coincide with some other libraries on your machine.

7.1.3 What's in MyRepo?

Your new repository comes with a GNUmakefile (which does nothing for now) and `doc` directory with a doxygen script but empty otherwise. Under this space, you can create your "packages" as a set of `C++` code to be compiled into a library. We will discuss about more later.

7.1.4 Creating Your Sub-Repository in `github`

In the previous section, we created a new repository `MyRepo` under `UserDev` . But that's just a directory on your laptop, and you may want to keep it as your code repository using either `svn` or `github` (or anything else you would like to use). Here, I briefly mention how you can do this using `github` because it's very easy given that you already have a `github` account.

First of all, go to `github.com` and create your own repository: go to your `github` account web page, and click on "+" symbol that is toward right-top of the web page. Choose "New repository". Then enter the repository name you are about to create. Ideally you may want to choose a somewhat unique name as described in Sec.7.1.2. You can always remove your repository if you don't like it.

Then checkout your empty repository. As an example, I use my empty repo called `EmptyRepo` (but note you cannot use my repository as this is my private code repository).

```
> cd $LARLITE_USERDEVDIR
> git clone git@github.com:drinkingkazu/
```

Now I simply run:

```
> python bin/gen_new_repository EmptyRepo
> cd EmptyRepo
> git add .
> git commit -m ‘‘new repository’’
> gitpush -u origin master
```

and I am done! From the next time, if I want to install LArLite from scratch, I would simply checkout my repository under UserDev with the same name. As said many times, of course, your repository is independent of LArLite repository unless you wish to merge for sharing purpose (in which case contact the author).

7.2 Creating a Package

Here, I assume we are working under `UserDev/MyRepo`. In fact, since it's tedious, I will just refer to `MyRepo`.

7.2.1 What is a "Package"?

"Package" is a directory under a repository that is compiled and generates one **shred object library** (i.e. a file with `.so` extension). This library is loaded at a run-time or linked via compiler as one byte code file. This gives you a sense of what to include in one package: having too many classes (and especially unrelated classes) in one package means an extra overhead cost for a run-time loading or linking at compilation. In other words, you probably do not want make one library per class, but also not one library for all classes. A package should contain a group of `C++` classes/functions which you think it makes sense to put together.

7.2.2 Making a Simple C++ Package

Like advertised many times, LArLite 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:

```
$LARLITE_BASEDIR/bin/gen_package
```

This script takes one input argument which is used to name a "package", a directory to be created under MyRepo. This script **needs to be executed somewhere under** MyRepo (otherwise you'll see an error message). One can run this script via alias:

```
> llgen_package MyProject
```

which will create a directory MyRepo/MyProject that 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 libMyRepo_MyProject.so under lib directory. What you compiled is a C++ class called "sample" defined in sample.h.

7.2.3 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] sample k;
```

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

7.2.4 "Hello World" Development

Let's try a simple modification to your class. Here's an example "hello world" program using C++ class. Open sample.h and add a void function as shown below:

```
class sample{
public:
  /// Default constructor
  sample(){};
  /// Default destructor
  virtual ~sample(){};
  /// 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] sample k;
root[1] k.HelloWorld();
Hello World!
root[2]
```

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!

7.3 Adding Basic `C++` Classes

As discussed in the previous section, one should populate a package with a group of `C++` classes. This section describes how to add various types of `C++` classes to your package.

7.3.1 Simple `C++` Class

Sometimes (rather often) 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 LArLite).

Here is a script for you:

```
> cd $LARLITE_USERDEVDIR/MyRepo/MyProject
> llgen_class_empty MyEmptyClass
```

Now you should see `MyEmptyClass.h` and `MyEmptyClass.cxx` created under `MyAna` package. There also made appropriate modification to `LinkDef.h` 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 ;)

7.3.2 `AnalysisUnit` Class

If you wish to generate an empty `AnalysisUnit` class code (to be implemented by you), simply try:

```
> cd $LARLITE_USERDEVDIR/MyRepo/MyProject
> llgen_class_anaunit MyAna
```

Executing above commands create `MyAna.cxx` and `MyAna.h` (and an appropriate modification to `LinkDef.h`). Try:

```
> make -j4
```

You just made your new `AnalysisUnit` class, `MyAna`, whieh inherits from `ana_base`! Your new `AnalysisUnit` is accessible from `CINT` or `PyROOT` like any other classes in LArLite:

```
> root
root[0] larlite::MyAna my_ana_instance
```

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

Run Your Analysis Unit: `PyROOT`

There is an example `python` run script for `AnalysisUnit` under `mac` directory, called `example_anaunit.py`. You need a LArLite sample `ROOT` file to run this program. If you have a sample file, say `trial.root`, you can run the program as follows.

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

7.3.3 `ERTool` Classes

Just as we exercised how to add a new `C++` class to a package, there's analogous python scripts to generate `ERTool` reconstruction algorithm, filter, and analysis class:

```
> cd $LARLITE_USERDEVDIR/MyRepo/MyProject
> llgen_class_erfilter Trial
> llgen_class_eralgo Trial
> llgen_class_erana Trial
```

Above three commands generate three `C++` classes: `ERFilterTrial`, `ERAlgoTrial`, and `ERAnaTrial`. These are implementation of base classes defined in `ERTool` , hence must be used with `ertool::Manager`. For details, see `ERTool` documentation (which is to come...)

7.4 Understanding the Build

This section covers the basics of how the LArLite build works, aiming to reduce a black-box content for users and developers. The following topics are ordered such that a topic of interest to more people gets covered first.

7.4.1 `Compiler` and `Linker` for Dummies

Skip if you already know about them, obviously.

Our `C++` source codes are merely descriptions of what we want our computer to do in human-friendly language (disagree on "human-fiendly"? Welcome to the club). A `compiler` takes in our source code and generate a machine-friendly description, often called as an *object file* or *byte code*. When you compile your package in LArLite, you find files with `.o` extension: these are the object files.

Now having many granular object files is not very helpful. A `linker` allows us to combine multiple object files and create a *shared object library* file. You find such files with `.so` extension under `$LARLITE_LIBDIR` if you compiled any package. The scope of what objects should be put together is really a developer's choice. In LArLite, this is done per package. Another (and more important) advantage of shared object libraries kicks in when you compile another code that depends on it. Say you have a `C++` class `A` and `B` where `B` depends on `A`. Once you have `A` compiled with `.so` file, a separate compilation of `B` does not require a re-compilation of `A`'s source code. Instead, you can *link* the `A`'s library upon compilation of `B`'s library.

In LArLite, we compile files with `.cxx` extensions with a compiler, and create a shared object library using a linker, which is nothing special compared to any other softwares' build system.

7.4.2 `INCFLAGS` and `LDFLAGS`: LArLite Compiler Flags

In LArLite package's `GNUmakefile`, you may see variables named as `INCFLAGS` and `LDFLAGS` where the latter may not appear in some simple code packages (i.e. don't worry about not seeing `LDFLAGS` in your `GNUmakefile`). These are called *compiler flags* and passed onto a compiler and linker respectively.

INCFLAGS

`INCFLAGS` is used by a compiler to search for files you specify with `#include` preprocessor command in your source code. In other words, if your source code calls `#include <TH1D.h>`, then the directory path which contains `TH1D.h` has to be added to `INCFLAGS`. The format of `INCFLAGS` is "-I$PATH" where you should replace "$PATH" with the relevant directory path.

That being said, by default, LArLite includes a `ROOT` 's include flags. `ROOT` follows a standard method to provide such flags, and you can try this in your installation as well:

```
> root-config --cflags
```

The output of above command is a part of LArLite's default `INCFLAGS`.

LDFLAGS

`LDFLAGS` is used by a linker to find libraries to be linked into your package. If your package depends on any externaly compiled code, for instance a `ROOT` class `TH1D`, you have to specify the library to be linked here. The format is `-L$PATH -l$LIBNAME` where "$PATH" is the directory path that contains a library named "lib$LIBNAME.so". Note that you should exclude the prefix "lib" when you specify it in `LDFLAGS` as that is the standard of how a linker takes in library names.

The default flags in LArLite includes a `ROOT` 's library flags. Again, as it was the case for `INCFLAGS`, you can try:

```
> root-config --libs
```

which include most of standard `ROOT` libraries such as IO, histogram, TTree, TMatrix, TVector3, etc.

Flags for LArLite Packages

You may write a package that depends on existing LArLite code. One example is an analysis code that inherits from `lalrite::ana_base`. Then you will have to specify `INCFLAGS` and `LDFLAGS`.

Specifying every single `INCFLAGS` and `LDFLAGS` for LArLite code is definitely annoying. So we follow the popular standard and the following scripts are prepared:

- `larlite-config` for LArLite's analysis framework
- `basictool-config` for code under `UserDev` /BasicTool
- `seltool-config` for code under `UserDev` /SelectionTool
- `recotool-config` for code under `UserDev` /RecoTool

These scripts can be run with either `--includes` or `--libs` option, and they simply prints out relevant string to be added to `INCFLAGS` and `LDFLAGS` respectively. For instance, you may try:

```
INCFLAGS += $(shell larlite-config --includes)
```

and/or

```
LDFLAGS += $(shell larlite-config --libs)
```

in your `GNUmakefile`.

If you use `gen_class_anaunit` or similar script, actually, this modification to a `GNUmakefile` is done by the same script. Hence you do not need to worry about it.

7.4.3 Compiler & Linker Used by LArLite

Let me make a brief note on how compiler/linker is picked and what default compiler/linker flags are set for LArLite build.

When possible LArLite attempts to use `clang` ++. In particular this is searched and set when one runs `setup.sh` script. The configured compiler name is set to a shell environment variable `$LARLITE_CXX`. Further, compiler/linker specific flags are defined in:

```
> $LARLITE_BASEDIR/Makefile/Makefile.X
```

where `X` may be either "Darwin" or "Linux".

7.5 Advanced Development

In this section we follow a prepared example that can be found in:

https://github.com/drinkingkazu/Example

You might have already checked out this repository as this was mentioned briefly in the introductory section (see Sec.1.4). If not, can certainly checkout under `UserDev` :

```
> cd $LARLITE_USERDEVDIR
> git clone https://github.com/drinkingkazu/Example
```

This repository contains following packages:

- `Empty` for the simplest example to introduce a simple `C++` class

- `Function` for demonstrating a `C++` function to be exported into a dictionary
- `Dependent` for showing how to make inter-package dependencies
- `DataProduct` as an example of how to store a class instance into a file

where we skip the first item, `Empty`, as that has been covered in the previous section.

7.5.1 `C++` Functions

I would like to avoid a confusion: `C++` class is an extension of data structure. In particular, you do not have to make `C++` class for inventing one or a few functions. Just like we have done some practice with `C++` class, it would be useful to have `C++` functions in a dictionary. The point of this package is to show how one can do this.

Take a look at `MyFunctions.h` and `MyFunctions.cxx` source code: there, I defined functions:

```
void hello_world();
Beer Brew(const int age);
```

where `example::Beer` is a `C++` class defined in `Beer.h`: it's very very simple class for playing around unlike the sophisticated name.

Now a compilation works just as expected, and there is nothing special about these functions. What you may find useful is a format to declare above functions in `LinkDef.h`:

```
#pragma link C++ function example::hello_world()+;
#pragma link C++ function example::Brew(const int)+;
```

As you can see, a) you do not specify the return type of the function, and b) you only need to specify the argument types.

You can try executing an example script `mac/example.py` which calls those functions. Take a look at the script's contents and see if that makes sense (... and ask a question if you have any!). **There is one caveat**, however: currently `ROOT` seems to require `C++` class compiled in the same library to be instantiated **before** `C++` functions to be called. The author will open a ticket for this to be fixed.

7.5.2 `ROOT` Data Product Class

Once you get familiar with all LArLite features, you might want to store `C++` object in a file as a `data product`. The easiest (and recommended) method is to use a `ROOT` file. A rule of the thumb is that any `C++` class that is generated with a `ROOT` *dictionary* can be stored in a `ROOT` file. You can find details in the `ROOT` manual.

If you use LArLite as your code development environment, `ROOT` dictionary file is always generated and built at compilation stage of your package.

An example can be found in the package `DataProduct`. There, a data product class called `example::Scotch` is defined.

An example script `mac/example.py` demonstrates how you can:

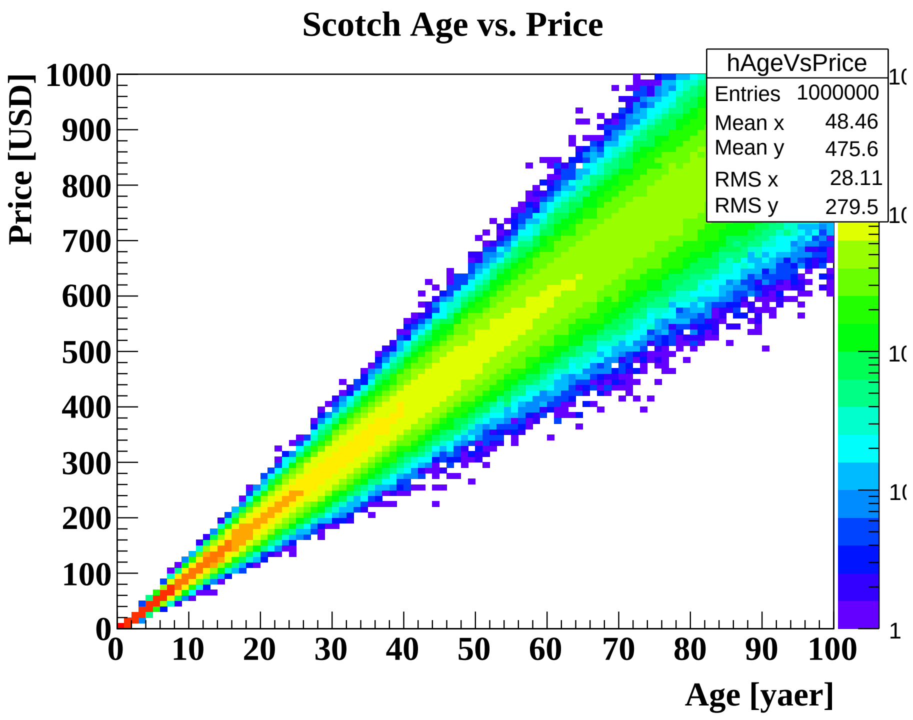


Figure 7.1: Drawing the age vs. price distribution of `example::Scotch` class from `TTree::Draw`.

- Instantiate `example::Scotch` and save it in a `ROOT` file
- Create a `TTree` of `example::Scotch` for 1e6 entries and save
- Read-in stored `TTree` and make a very simple access to the stored data product
- Read-in stored `TTree` and call `TTree::Draw` on the stored data product

Note that storing multiple variables into a separate branch of `TTree` (i.e. Ntuple approach) is much more tedious than storing a class instance as demonstrated in `Scotch::ShipScotch` function. There, you can find a single line to create a `TTree` branch:

```
tree.Branch("scotch",&data);
```

where "data" is a `example::Scotch` instance. With this single line call, all variables in `example::Scotch` instance is stored with a proper type, and will be readout correctly.

Another myth often people have is that it is very hard to access this object information from `TTree`. As you can see in the `example.py`, this is completely irrelevant: you can access the object directly via branch name. A single line in `Python` from the script is shown below:

```
ch.scotch.Speak()
```

which accesses the stored object and calling the class member function `example::Beer::Speak()`.

7.5.3 Package Dependency

The package `Example/Dependent` is prepared to demonstrate how one can make an inter-package dependency. In particular, this package depends on `Example/Function` package. Make sure you have finished building `Example/Function` before building this package.

As you can see in `Stout.h`, the `C++` class `example::Stout` inherits from `example::Beer`. In order to avoid re-compilation of `example::Beer` class, we take a usual approach of just *linking* the libraries together. Take a look at `GNUmakefile`, in particular the following lines:

```
...
INCFLAGS += -I$(LARLITE_USERDEVDIR)/Example
...
LDFLAGS += -L$(LARLITE_LIBDIR) -lExample_Function
```

As advertised in the previous section, you must include the path to find `Beer.h` (which is called from `Stout.h` via `#include ‘‘Beer.h’’`) in `INCFLAGS`, and both a path and name of a library that contains `example::Beer` class definition in `LDFLAGS`.

An example script `mac/example.py` demonstrates an instantiation of `example::Stout` class and also a call to its base class function `example::Beer::Speak()`.

7.5.4 `python` in `C++`

Finally, some users are interested in developing `C++` code to directly operate on `python` objects. An example `Example/PyExample` is prepared to give a tip on how one can write a `C++` API for `python` . This is much better documented as a part of `python` documentation:

https://docs.python.org/2/c-api/

Using a native `python` `C++` API is not very easy nor straight forward. Hence in this example, again, we use `PyROOT` binding that makes our life much easier. In short, a difference of two methods is that you do not have to write your own binding, which is an extra code that has nothing to do with your algorithm. Finally, that being said, there are other bindings available out there such as `Cython` (... and `PyROOT` uses one of them underneath anyways).

External Dependnecy

First of all, you will be using native `python` classes, hence you will depend on `python` . Accordingly you have to modify `GNUmakefile`, in particular `INCFLAGS` and `LDFLAGS`. Notice following 2 lines in `GNUmakefile`:

```
...
INCFLAGS += $(shell python-config --includes)
...
LDFLAGS += $(shell python-config --ldflags)
```

each specifying an extra path to find a `python` header file and libraries.

Hiding `python` Header From `CINT`

`python` header file is called `Python.h`, and is not parse-able via `ROOT CINT` compiler. So you have to hide it from `CINT` dictionary generation step. This can be seen in `PyExample.h` header file:

```
...
#ifndef __CINT__
// You have to hide native Python header include from CINT
#include "Python.h"
#endif
...
```

You also need to provide two magic lines to forward declare types:

```
...
struct _object;
typedef _object PyObject;
...
```

which is for `python` generic object pointers to be used (this is a part of `python C++` API).

`example::PyExample::Convert`

The example class `example::PyExample` has an example function to create a `python` list from a `C++ std::vector<std::string>` object. You can look at the source code as to how one can do this very simple operation, and take a look at `python` documentation for doing more elaborate operations.

Running `mac/example.py` should show you how this function works. Obviously you may want to expand the code to work with more advanced `python` objects such as `numpy` array, `matplotlib` axis, and such. The author does not have much experience to extend too far, but is happy to discuss if you need a help.

7.5.5 Documenting Code Using `Doxygen`

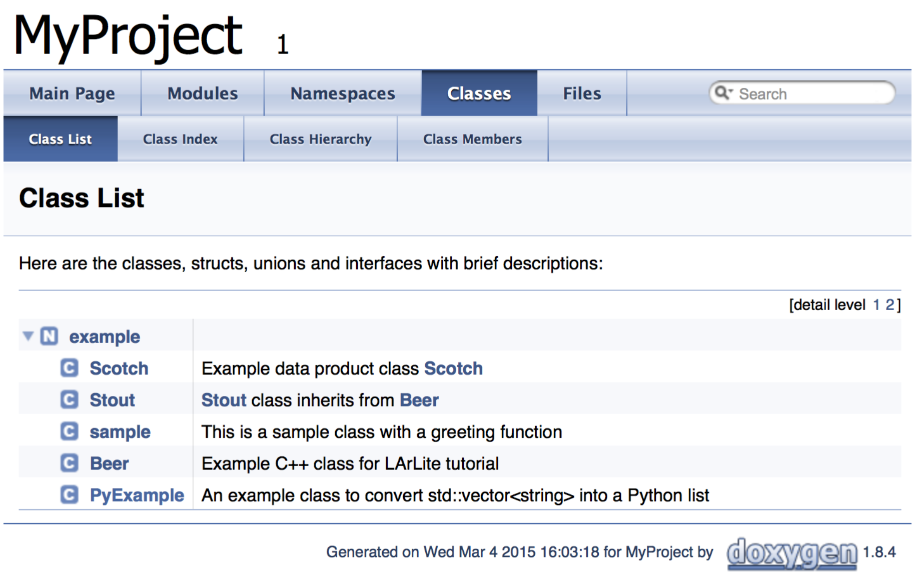
Though this is not everyone's favorite choice, `doxygen` is a popular method to provide an in-line documentation in the source code. It works pretty nicely with `C++` , and somewhat at acceptable level with `python` . It is the recommended choice for LArSoft to provide a minimal documentation.

LArLite repositories comes with a doxygen script. You can try generating a documentation in `Example` directory by typing:

```
> make doxygen
```

(**you need doxygen installed in your system!**).

After successfully running the above command, you should find a chain of HTML files to browse through (no internet needed). This way you can also check your doxygen comment format (whether this is correct or not) before you commit the code. You can look at the generated documentation using a command like below:



MyProject 1

Here are the classes, structs, unions and interfaces with brief descriptions:

example	
Scotch	Example data product class Scotch
Stout	Stout class inherits from Beer
sample	This is a sample class with a greeting function
Beer	Example C++ class for LArLite tutorial
PyExample	An example class to convert std::vector<string> into a Python list

Generated on Wed Mar 4 2015 16:03:18 for MyProject by doxygen 1.8.4

Figure 7.2: Doxygen web page generated by "`make doxygen`" command for `Example` repository.

```
> firefox -a doc/dOxygenMyProject/html/index.html
```

on `Linux` or

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

on `OSX`. The figure shows a generated documentation webpage on the author's laptop.

To understand `doxygen` comment style, refer to their web documentation:

http://www.stack.nl/~dimitri/doxygen/manual/docblocks.html

Chapter 8

Using LArLite 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 LArLite 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 "$\rightarrow$" 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 LArLite Classes

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

```
import ROOT
from ROOT import larlite
my_storage = larlite.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 LArLite, 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 LArLite as much as possible.

8.3.2 Easy (possibly dirty) Data Access

DOCUMENTATION ONGOING

Chapter 9

Where is LArLite Data File?

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

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

Following sections describe in details about these options.

9.1 Generating LArLite 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 LArLite Data Format

As described in Ch.4, LArLite 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 LArLite?

There are two possibilities:

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

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

The second point does not mean we have to introduce `art` framework dependency in LArLite: that would screw up the whole purpose of LArLite 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 LArLite someday...

9.2.1 LArSoft ⇒ LArLite: `LiteScanner`

`LiteScanner` is a LArSoft module that is maintained under `uboonecode` repository (official MicroBooNE code repository)

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. There is a good guidance available [1]. For the moment, the author recommends to use the `v03_04_01 -q e6:prof` version of `LArSoft`.

1. Checking Out and Bulding LArLite

Once you set up LArSoft environment, you should have `git` and `ROOT` . Then we can checkout and build LArLite:

```
> cd $WHEREVER_YOU_WANT_TO_BUILD
> git clone git@github.com:larlight/larlite LArLite
> cd LArLite
> git checkout trunk
> source config/setup.sh
> make -j4
```

Once the build finishes, you are good to go.

1. Checking Out and Bulding `LiteScanner`

OK, we assume you have gone through prerequisite and you have `$MRB_TOP` shell environment variable set. Let us checkout `uboonecode`:

```
> cd $MRB_TOP/srcs
> mrb g uboonecode
```

You find LiteMaker package here:

```
uboonecode/uboone/LiteMaker
```

Before start compiling `uboonecode`, you have to change

```
uboonecode/uboone/CMakeLists.txt
```

which, by default, excludes `LiteMaker` package from the build. It is as simple as un-comment the following line

```
#add_subdirectory(LiteMake)
```

Once that is done, try:

```
> mrbsetenv
> mrb i -j4
```

2. Running `LiteScanner` Build

We go over how to configure `LiteScanner` , but why not just running a fcl file first? If this fails, you the author should fix it w/o furhter wasting your time to learn how to configure `LiteScanner` . Go to `LiteScanner` directory:

```
> cd $MRB_TOP/srcs/uboonecode/uboone/LiteMaker
```

Let's run a simple 2 muon event generation.

```
> lar -c job/larlite_maker.fcl -s /uboone/data/users/kterao/sample.root -n 5
```

If the above execution of `lar` goes well, then you should have generated `larlite.root`. Otherwise please contact the author!

3. Configuring `LiteScanner`

You have already run `LiteScanner` in **Testing** `LiteScanner` **Build** section with a fcl file, `job/larlite_maker.fcl`. In this fcl file, you can find (hopefully) self-descriptive parameters.

There are two types of parameter name strings (other than module label):

- `store_association` ... enable/disable storing association information
- `larlite::data::kDATA_TREE_NAME` strings ... specifies data product type to store from LArSoft file. You provide producer modules' label in LArSoft file as a vector of strings.

If the `store_association` is set to `true`, then all associations among stored data products will be stored. Note that associations between stored data products and those not stored are not saved. The author thinks a little more flexibility is needed, so there will be an improved feature soon but not yet available. If you do not want to store certain data product type, you can either comment out the corresponding string or provide an empty vector as an argument.

Here is a few points to remember:

- If you put an wrong producer name, you will get a corresponding LArLite data `TTree` filled with correct number of events. But each event will be empty.
- If you put an empty string vector as producers' name, then you will not get any corresponding LArLite 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 `larlite::hit`, you must store this data product (for now).

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

9.3 Use Existing File

You can find some of LArLite files under:

`/pnfs/uboone/scratch/uboonepro/mcc6.0_lite`

directory path on `uboonegpvm` machines. You may `rsync` them (as opposed to `scp`) to keep them updated at your copied destination if you would like to move files to elsewhere. These are produced with MCC 6.0.

Note some of those files are huge as they store very heavy information such as individual MC particle information. Most likely what you need is a portion of them, and you should think before blindly copy everything (like down to your laptop!).

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 LArLite?

- See Sec.1.1.

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

- 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 LArLite 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 LArLite. How can I test?

- There are some test routines. See Sec.1.5.

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

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

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

- You can generate your own `HTML` file of class index. See Sec.1.4.3.

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 LArLite `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 LArLite 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 LArLite. As asid in Ch.2, LArLite 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 LArLite?

- 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.7.2.4.

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

- You can either (1) generate your own, (2) convert from LArSoft into LArLite, 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
  * Write `CINT` or `PyROOT` script

  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 LArLite but need to transport to LArSoft soemtime in future. Any tip?

- LArLite only depends `ROOT` and `C++` , so you should be able to copy most part of your code except for some portion that uses other LArLite packages you are not exporting to LArSoft. For instance, functions that access LArLite data products. So, when you write your code, keep that in mind and structure your code to decouple a dependency to LArLite data products from the algorithm, the core part of your code. One solution is to define your own data container `C++` struct, copy LArLite 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 LArLite.

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/larlite LArLite
  ```

  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 LArLite?

– If you are committing changes/additions to your own package (see Ch.7), then you won't break LArLite. 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 LArLite 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).

v3.0.0

- Should be tagged as the last LArLight release (does not exist yet)

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.