LArSoft: toolkit for simulation, reconstruction and analysis of liquid argon TPC neutrino detectors

IOP Conf. Series: Journal of Physics: Conf. Series **898** (2017) 042057 doi:10.1088/1742-6596/898/4/042057

LArSoft: toolkit for simulation, reconstruction and analysis of liquid argon TPC neutrino detectors

E.L. Snider and G. Petrillo

Fermi National Accelerator Laboratory[1], P.O. Box 500, Batavia IL 60510, USA

E-mail: erica@fnal.gov, petrillo@fnal.gov

Abstract.

LArSoft is a set of detector-independent software tools for the simulation, reconstruction and analysis of data from liquid argon (LAr) neutrino experiments The common features of LAr time projection chambers (TPCs) enable sharing of algorithm code across detectors of very different size and configuration. LArSoft is currently used in production simulation and reconstruction by the ArgoNeuT, DUNE, LArIAT, MicroBooNE, and SBND experiments. The software suite offers a wide selection of algorithms and utilities, including those for associated photo-detectors and the handling of auxiliary detectors outside the TPCs. Available algorithms cover the full range of simulation and reconstruction, from raw waveforms to high-level reconstructed objects, event topologies and classification. The common code within LArSoft is contributed by adopting experiments, which also provide detector-specific geometry descriptions, and code for the treatment of electronic signals. LArSoft is also a collaboration of experiments, Fermilab and associated software projects which cooperate in setting requirements, priorities, and schedules. In this talk, we outline the general architecture of the software and the interaction with external libraries and detector-specific code. We also describe the dynamics of LArSoft software development between the contributing experiments, the projects supporting the software infrastructure LArSoft relies on, and the core LArSoft support project.

1. Introduction

Liquid argon time projection chambers (LArTPCs) [1] have become important tools for precision measurements of neutrino properties and their interactions with nuclear matter, as evidenced by the raft of recent and future accelerator-based neutrino experiments that utilize LArTPCs [2]–[6]. Among the leading features in their appeal are a large, uniform sensitive volume, and the high-precision position and energy deposition information obtained from the TPC. The readout systems among all of the large scale TPCs currently in operation or in planning utilize multiple, parallel planes of strips, each of which provides a 2-D projected image of charge deposition onto a plane perpendicular to the strips. By combining these 2-D images from strips at different angles, it is possible to construct a a 3-D image of the charge deposition. In addition to the collection of charge deposited in the TPC volume, most detectors utilize photo-detector systems to sense scintillation light created in conjunction with ionization processes within the liquid argon. The arrival time of this light is used to determine event interaction times.

[1] Operated by Fermi Research Alliance, LLC under Contract No. DE-AC02-07CH11359 with the United States Department of Energy.

The similarity of this basic geometry and readout scheme across detectors, combined with the common physics of the LArTPC itself, opens a unique opportunity within high energy physics for sharing primary simulation, reconstruction and analysis software that is interoperable among the various LArTPC-based experiments. Supporting such a shared ecosystem presents challenges of design, coordination and organization.

The LArSoft collaboration [7] is a group of experiments, laboratories, university groups and software projects formed to meet these challenges, and to contribute to the shared, detector-independent software tools and algorithms needed for the simulation, reconstruction and analysis of data from their respective experiments. By working together, member experiments leverage expertise across the community, and share knowledge and solutions as embedded directly in the software. At the time of writing, the collaboration includes the ArgoNeuT [2], DUNE [3], LArIAT [4], MicroBooNE [5] and SBND [6] experiments.

The product of the LArSoft collaboration is the body of shared code, developed under a set of common design principles and practices. At present, the shared repositories contain about 250k lines of C++, with approximately 200k additional lines in experiment-specific repositories. More than 100 authors from over 25 institutions have contributed to the core software suite.

To provide for the common needs of the collaboration and its members, a dedicated group of physicists, software engineers, developers and others comprise a core "project" team that works under management oversight by the experiments. Major elements of the core project's mission include provisioning and supporting the LArSoft framework, software architecture and design; performing release management and testing functions; coordinating code integration and planning across experiments; and providing software engineering expertise to the LArSoft community.

In the following two sections, we describe the general architecture of the software, some of the design principles and practices, and how the shared software interfaces to experiment-specific code and external libraries. Next, we explore how the contributing experiments, software projects and the core project team interact to coordinate the support of the software and related infrastructure, as well as the integration of code into the LArSoft ecosystem. We conclude with future plans and a summary.

2. The event-processing framework

LArSoft is built upon the *art* [8] event-processing framework developed at Fermilab, and which is used by most of the neutrino and muon experiments based at Fermilab [9]. The framework provides a number of facilities to define and configure algorithm workflows that operate on input data streams The data itself is organized into a hierarchy of runs, sub-runs and events, with user-defined data structures ("data products") comprising the content.

Steps within a workflow are performed by experiment-written "modules", which have methods called by the framework at well-defined states within the event-processor, for instance, before or after processing a run, or during a workflow on an event. Algorithms within each module operate on input data accessed through the data model, and either produce output data to be placed into the data stream or other analysis streams, or make a filtering decision that can alter downstream processing or data output actions. Filtering decision can, for instance, stop processing of a workflow at that point, or cause output data to be written to a particular file.

In addition to modules, users may define "services" that have access to the event processor state transitions, and that can be used to provide global access to a particular resource within modules, such as detector geometry. Users may optionally define abstract interfaces for services, which allows the implementation to be selected at run time via an input configuration file.

The configuration of modules, services, and module workflows is specified by a user-provided FHiCL [10] configuration file. These files may also specify input and output data streams, actions that result from filter decisions, data products to be dropped on input or output, etc.

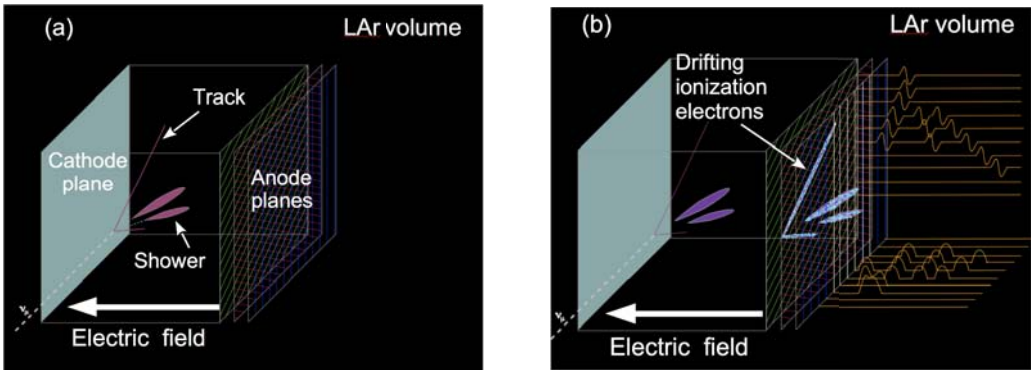

Figure 1. Operation of a LArTPC neutrino detector. (a) Neutrino interactions create secondary charged particles that leave tracks of ionization in the liquid argon. Electrons and photons create electromagnetic showers, leaving regions of ionization with irregular shapes. (b) The ionization electrons are swept from the TPC under the action of the applied electric field, and drift to the anode wires or strips, where they induce signals on the associated readout channels. The resulting waveforms are then digitized.

The *art* framework also tracks the provenance of all data products generated during execution through a combination of tags attached to data products recording the module instance that created it, and the full FHiCL configuration of the job.

3. Design principles and practices

Writing simulation and reconstruction software that is shared between different detectors and experiments demands adherence to a number of common design principles and practices. Together, these practices form the pillars that support the entire software sharing regime. We discuss a few of these below. Other practices, such as continuous integration, peer analysis of code, and centralized infrastructure support and coordination are discussed in later sections.

3.1. Detector interoperability

The interoperability of the algorithm code forms the cornerstone of the entire project, and depends upon the common features of LArTPCs. That commonality extends to the output data from the detector, which consists of digitized waveforms from each TPC readout channel. These waveforms represent the charge induced by the motion of ionization electrons swept by the drift field from the volume of the TPC, as shown in Fig. 1. Similarly, the output data from the photo-detectors consists of digitized waveforms representing the signals from detected scintillation photons. Each of these waveforms has a detector-independent data product representation.

The output of the reconstruction — from low-level hits characterizing localized charge deposition to high-level 3-D objects, such as tracks and electromagnetic showers, to their physical properties, such as range, dE/dx and particle ID — also resides in experiment-independent representations. These common data products form the basis for uniform interfaces between various phases of the reconstruction and simulation algorithms.

A wide variety of detector configuration and operating condition information is needed inside both reconstruction and simulation algorithms. The code must in all cases remain free of implicit assumptions as to values or relationships that may vary from detector to detector. Again, the common features of LArTPC geometries and physics allows all of this information to be described using a nomenclature that lends itself to detector-independent interfaces. TPCs, for instance, are rectangular prisms with an applied electric field that drives electrons to a set of anode

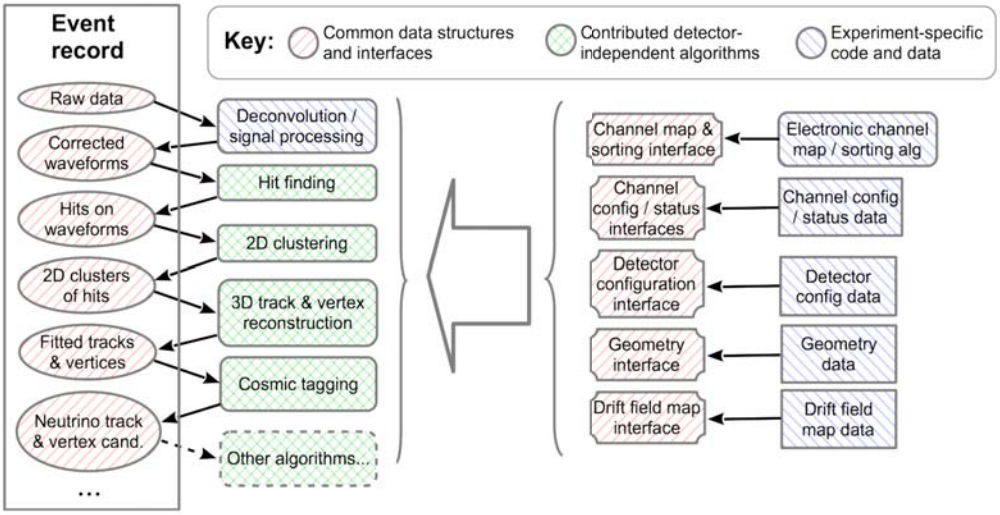

Figure 2. Simplified sample LArSoft workflow for 3D track / vertex reconstruction showing the detector independent elements in LArSoft, and some of the detector-specific code and data needed. Other workflows, such as electro-magnetic shower or photo-detector reconstruction, branch and merge at various points. The overall configuration of the workflow, algorithm configurations, data sources, etc., is specified via a detector-specific FHICL file.

planes. Each plane has "wires" (or strips) charactered by a pitch between channels and a fixed orientation within the plane. Experiments with multiple TPCs are easily accommodated within this scheme.

To facilitate navigation through various levels within the readout channel and geometry hierarchies, LArSoft provides generic geometric and readout IDs at each of the various levels, and tools for sorting and constructing generic loops over elements at each level.

The combination of common interfaces to all this information and generic navigational tools allows the simulation and reconstruction code to be completely detector-independent. Figure 2 illustrates a simplified reconstruction workflow, and the relationship between the common data structures, interfaces and contributed detector-independent algorithms in LArSoft, and some of the detector-specific data and implementations that may be required.

3.2. Configurable handling of detector-specific customizations

All detectors require a certain level of detector-specific customization during simulation and reconstruction. LArSoft employs two basic schemes for providing this. The preferred method is via run-time configuration data, since that allows more sharing of implementation code.

As previously discussed, geometric information, detector configuration and operating conditions can be accessed and applied via detector-independent interfaces. In the case of the geometry, initializing the interface for a particular detector is a simple matter of specifying the appropriate geometry description file[2].

Similarly, differences in a number of detector operating parameters, such as drift field maps, common calibration parameters, etc., can be obtained from sources specified at run-time. Typical sources include databases, configuration data files, or the input FHiCL file itself. Where

[2] LArSoft uses GDML [11] as the standard geometry and material description format. The source of the GDML description is left to the experiments, which are free to use scripts or databases as the reference source.

necessary, different implementations for the services that provide this information can also be specified as part of the input FHiCL configuration.

Some customizations require detector-specific code. Examples of such cases currently include readout channel to geometry mappings, raw data noise removal and signal processing, handling of the electronics response in the simulation and reconstruction, and the simulation of raw data digitization. In the case of the channel mappings, an *art* service interface allows a run-time implementation to be selected that is then accessed via the generic geometry service. For the other cases, LArSoft relies on the common data structures to define standard interfaces to the relevant algorithms. Most of these algorithms live in dedicated *art* modules in experiment repositories.

Work is currently in progress to create a more flexible framework for these algorithms that will allow more of the underlying code to be shared. This change would allow much of the signal processing to become a matter of configuration data.

3.3. Standardized interfaces and usage patterns

The use of standardized interfaces extends beyond the detector-specific customization and into the algorithm code itself. There are in many cases multiple algorithms available at each stage through the reconstruction and simulation. Examples include hit finding within the raw data waveforms, clustering of TPC hits into 2-D "cluster" objects, identifying photo-detector hits and determining interaction times, finding tracks and vertices, etc.

An even more important use case is the layering of algorithms that perform incremental refinements within a single reconstruction phase. Such strategies generally provide a more maintainable, extensible, and understandable model for introducing high degrees of sophistication within a larger reconstruction algorithm. Again, the communication of data between algorithms via common data structures allows for the definition of the interfaces needed to support this layering. LArSoft promotes the definition of common, standardized interfaces whenever possible in all of these cases.

Standard usage patterns develop from these standardized interfaces. The conventions that result can greatly simplify the task of learning the software, and reduce the time needed to become a valuable LArSoft contributor.

3.4. Framework independence of data structures and algorithm code

The architecture of LArSoft is based on a layering model in which data structures and algorithm code are uncoupled from the *art* framework, external software products (except by prior community agreement), and any experiment-specific code. In this scheme, *art* modules serve only as an interface to framework functionality, performing such tasks as retrieving input data products from the event record, writing output data products to the event record, retrieving *art* services from the service registry, or obtaining algorithm configuration information that was specified by the user.

Similarly, *art* services within LArSoft are designed to operate independently of the framework, and have no knowledge of the framework service registry from which they were obtained.

A number of benefits accrue from this separation, particularly given that the event data model is *also* independent of the *art* framework, and can be used as an external product with minimal dependencies. Algorithm code structured in this way can be fully unit tested outside the framework in a very lightweight test environment. In addition, such code is naturally more modular than algorithms tied to the framework, which broadens the toolkit nature of the LArSoft suite, and encourages the further layering of algorithms.

Code that is independent of the *art* framework also allows for the creation of minimalist development environments, which can facilitate or promote easier and faster development cycles. One such environment is enabled by the `gallery` application [12] developed by Fermilab, which

provides a simple event loop capability built on top of the *art* event data model. This tool is also widely used on its own as an analysis framework for the final stages of data analysis that is extremely light-weight, yet in principle still provides access to the full power of shared data structures, tools and algorithms within LArSoft.

3.5. Interfacing to external software products

There are a number of external products that provide simulation and reconstruction algorithms or capabilities that are of great utility to LArSoft users. Principal among these are event generators such as GENIE [13], the Geant4 simulation package [14], and the Pandora pattern recognition package [15]. In each case, there is a mapping between the common data structures within LArSoft, and some input or output data structure for the external product.

The strategy for interfacing to these products within LArSoft involves isolating the dependence on the product to a single library. Classes and functions within that library perform conversions between the relevant LArSoft data products and those structures used by the external product. An *art* module can then use this library to provide a direct interface between the product and LArSoft.

4. The development environment

The reference copy of LArSoft code is held in `git` repositories maintained at Fermilab [16]. Experiment-specific code lives in separate repositories maintained by the respective experiments. The features of `git` combined with the `gitflow` branching model [17] are well suited to maintaining a rapidly evolving yet stable development environment involving many users and developers.

The source code build infrastructure is currently based on that used for *art*. The system consists of four basic tools: `ups`, a product packaging and environment configuration tool maintained by Fermilab; the `cmake` open-source build management tool; `cetbuildtools`, a `cmake` macro package used by *art*; and `mrb`, a tool based on `cetbuildtools` that provides utilities to help manage the end-user development and build environment, and operate the build infrastructure across multiple `git` repositories.

In order to provide a stable development platform, LArSoft provides weekly integration releases. Developers typically work on a `git` branch against one of these releases until a desired functionality works, then merge changes from the main development branch (called "develop") into the working branch, and finally back to develop, where it is picked up in a subsequent integration release. Breaking changes are merged during the final stages of the release process, a procedure that ensure that the head of the develop branch always builds.

While most large scale processing occurs on Linux-based machines, much of the developer base uses machines running Mac OS X or Ubuntu. At present, LArSoft distributes code for Scientific Linux Fermi [18] versions 6 and 7, Mac OS X Yosemite, El Capitan and Sierra, and Ubuntu 16 LTS.

Code is distributed via both self-building source installs, or binary-only installs that are accessible via a dedicated web site. Source code and binaries for Linux and Mac OS X are also accessible via cvmfs [19], a software distribution service implemented as a POSIX read-only file system in user space. The use of cvmfs eliminates the need for local installs of any sort.

The core LArSoft project strives always to improve the overall quality the the code and compliance with design principles. One strategy LArSoft employs is to hold in-depth code analyses. Targets for these sessions are identified by the experiments, and are performed collaboratively with developers and C++ experts. The specific goals and scope of the analyses are highly tailored to the needs of the experiment for the particular code offered for inspection. Analyses typically focus on issues of CPU and memory performance, architecture, and general coding practices, and take about a day of effort from those participating in the analysis.

4.1. Continuous integration and testing

LArSoft relies on a continuous integration (CI) and testing model in order to ensure compliance with detector interoperability standards, check basic functionality for unexpected behavior, and obtain rapid feedback in cases where breaking changes are inadvertently committed to the central repositories. LArSoft uses a Fermilab-supported CI system built on top of the Jenkins open-source automation system [20]. The system provides a number of features and tools to facilitate construction of integration test workflows and analysis of program output.

With every commit to the develop branch of the central `git` repositories, this system launches a set of builds of the full software suite followed by rapid cycling unit and integration tests. Test results are monitored by LArSoft team members, and optionally via automated emails sent to the developers responsible for the changes.

In addition to basic functionality testing, the system tracks historical memory and CPU usage across builds. This feature can be used to identify both discrete problems introduced into the code, and long-term trends that can be addressed before they become problems.

Two important design features of the test system are local configurability and executability. All tests can be defined and configured by individual developers via simple configuration files within each repository. In addition, any user can locally execute the full test infrastructure without any interaction with the Jenkins automation system. These features allow users to create new tests, then run them locally on private branches in their local `git` repositories before committing any changes to the central repositories. Users can also execute the tests on arbitrary branches on the central repositories via the Jenkins system.

The system supports distributed and remote build nodes. This feature can be used to provide builds and CI testing on systems not supported at Fermilab. For instance, LArSoft recently added Ubuntu 16 LTS to the set of supported platforms via this mechanism.

5. Coordination of collaboration work

Developing code within such a large community working on different experiments requires good coordination and communication across experiment boundaries. The core LArSoft project facilitates this exchange in a number of ways. The primary means is via a regular "coordination meeting." Members of from the various experiments attend this highly technical forum to propose, discuss and coordinate code and policy changes, release plans and other issues relevant to the collaboration.

The project also gathers requirements on a regular basis via a special series of meetings with experiment offline coordinators, and more broadly from the community at dedicated workshops. Discussion of major technical issues and initiatives and short term project work occur in regularly scheduled meetings with experiment offline coordinators.

The overall direction of the collaboration, and the priorities and goals of the core project are set or approved in meetings with the experiment spokespeople held quarterly.

6. Future plans

The primary goal for future work is to expand and improve the capabilities, usability and design of the shared code base while maintaining a close collaboration among experiments.

Support for multi-threading will become increasingly important in order to fully exploit the continuing growth in the number of cores per processor, and be an important tool to control memory usage for large detectors. The project will begin to review and revise the architecture as needed to support multi-threading in LArSoft algorithms over the next year. The *art* framework is expected to support multi-threading on the time scale of summer 2017.

Vectorization of applicable algorithms represents a complementary approach aimed at improving processing throughput by accessing vector resources on existing and future processors that are not currently being utilized.

Integration with additional external products are also under way or on the horizon. The WireCell 3-D reconstruction package [21] developed at Brookhaven National Laboratory is one such package recently integrated. Architectural changes to the LArSoft simulation interface will simplify integration with the other detector simulation products, such as FLUKA [22].

Possible extensions needed to facilitate interfacing to image-based machine learning algorithms, such as convolutional neural networks, are also under review.

Extending LArSoft to other detectors is also a high priority. At present, LArSoft team members are working to extend support to the ProtoDUNE dual-phase detector. The project is also working with members of the ICARUS collaboration with the same goal. Expertise from both of those experiments have a great deal to offer the rest of the LArSoft community.

7. Summary

LArSoft demonstrates what has been a novel and successful model for sharing primary LArTPC simulation and reconstruction software across detectors and experiments. The common data structures and algorithms have proven themselves to be enabling technologies in the case of small experiments and teams where the effort needed to develop proprietary solutions is limited. The architectural choice to separate framework from data and algorithms has also shown itself to be a powerful tool in shortening code development cycles, integrating light-weight analysis frameworks, and enhancing user satisfaction.

The collaborating experiments remain highly engaged in LArSoft at all levels, from providing new ideas and requirements, to contributing new code, or using and improving existing code shared by other experiments. New experiments will find a welcoming and vibrant community that has many plans and ideas for future work.

References

[1] C. Rubbia, 1977, *Preprint* CERN-EP-INT-77-08.
[2] C. Anderson *et al* , 2012, The ArgoNeuT Detector in the NuMI Low-Energy beam line at Fermilab, JINST **7**, P10019. See also `http://t962.fnal.gov`.
[3] R. Accarri *et al* , 2016, Long-Baseline Neutrino Facility (LBNF) and Deep Underground Neutrino Experiment (DUNE) Conceptual Design Report Vol. 1: The LBNF and DUNE Projects, *Preprint* arXiv:1601.05471. See also `http://www.dunescience.org/`.
[4] J. Paley *et al* , 2014, LArIAT: Liquid Argon In A Testbeam, *Preprint* arXiv:1406.5560. See also `http://lariat.fnal.gov`.
[5] R. Accarri *et al* , 2016, submitted to JINST (*Preprint* FERMILAB-PUB-16-613-ND. See also `http://www-microboone.fnal.gov`.
[6] See R. Accarri *et al* , 2015, A Proposal for a Three Detector Short-Baseline Neutrino Oscillation Program in the Fermilab Booster Neutrino Beam, *Preprint* arXiv:1503.01520. See also `http://sbn-nd.fnal.gov`.
[7] R. Pordes and E. Snider, 2016, *Proceedings of Science (ICHEP2016)*, 182. See also `http://larsoft.org`.
[8] *http://art.fnal.gov*
[9] See `http://art.fnal.gov/who-uses-art/` for a list of experiments currently using *art*.
[10] The Fermilab Hierarchical Configuration Language, `https://cdcvs.fnal.gov/redmine/projects/fhicl/wiki`.
[11] R. Chytracek, 2006, Geometry Description Markup Language for Physics Simulation and Analysis Applications, IEEE Trans. Nucl. Sci **53**, p. 2892. See also `http://cern.ch/GDML1`.
[12] See `http://art.fnal.gov/gallery`.
[13] C. Andreopoulos *et al* , 2006, Acta Phys. Polon. **b37**, p. 2349. See also `https://genie.hepforge.org`.
[14] See `http://www.geant4.org`.
[15] J.S. Marshall and M.A. Thomson, 2015, Eur. Phys. J. **C75** no.9, 439.
[16] See `https://cdcvs.fnal.gov/redmine/projects/larsoft/wiki/_LArSoft_repositories_packages_and_dependencies_`.
[17] `http://nvie.com/posts/a-successful-git-branching-model`.
[18] `https://fermilinux.fnal.gov`.
[19] `https://cernvm.cern.ch/portal/filesystem`.
[20] `https://jenkins.io`.
[21] `http://www.phy.bnl.gov/wire-cell`.
[22] G. Battistoni *et al* , 2015, Annals Nucl. Energy **82** p. 10. See also `http://www.fluka.org`.