

Optical Simulations in LArSoft - Technical Manual

Ben Jones, Massachusetts Institute of Technology

December 12, 2012

Abstract

This note describes the optical monte carlo simulation tools which have been developed within the LArSoft framework for MicroBooNE, LBNE and other detectors with optical systems. I try to give a full technical overview of the function of optical physics features in LArSoft and how they should be used. This note is intended to be used as a reference in conjunction with the LArSoft code repository, and of use to both users and developers of LArSoft optical simulations.

Contents

1	Introduction	2
1.1	Introduction to the LArSoft Optical Simulations	2
1.2	A Note on LArSoft Code Structure	3
1.3	LArSoft Packages with Optical Features	3
2	Optical Geometry in LArSoft	4
2.1	Optical Components within the GDML Description	4
2.2	Optical Component Handling in the LArSoft Geometry Service	6
2.3	Voxelization Schemes	7
3	Optical Data Products	7
3.1	Products in the Simulation package	8
3.1.1	The sim::OnePhoton class	8
3.1.2	The sim::SimPhotons class	8
3.1.3	The sim::SimPhotonsCollection class	9
3.2	Products in the RawData package	9
3.2.1	The raw::OpDetPulse class	9
3.3	Products in the RecoBase package	9
3.3.1	The recob::OpHit class	10
3.3.2	The recob::OpFlash class	10
4	Simulation and Analysis Tools	10
4.1	The LightSource event generator	10
4.2	The SimPhotonCounter analysis module	11
5	The Full Optical Simulation in LArG4	12
5.1	Configuring and Running the Full Optical Simulation	13
5.2	Physics List Handling in LArG4	13
5.3	The OpticalPhysics Constructor and Associated Physics Processes	15
5.3.1	Scintillation	16
5.3.2	Cherenkov Light	17
5.3.3	Rayleigh Scattering	17

5.3.4	Surface Reflection and Absorption	17
5.3.5	Bulk Absorption	18
5.4	Custom Material Property Handling	19
5.5	Sensitive Detectors, SimPhoton Logging and Parallel Worlds	19
6	The Fast Optical Simulation in LArG4	20
6.1	Configuring and Running the Fast Optical Simulation	20
6.2	The Mechanics of the Fast Simulation, and the PhotonPropagation Package	21
6.2.1	The PhotonLibrary Object	21
6.2.2	The PhotonVisibilityService	23
6.3	The Library Building Process	23
6.3.1	Configuring and Running a Library Build Job	23
6.3.2	Library building with the grid	24
6.4	The PhotonLibraryAnalyzer Tool	25
7	Digitization and Basic Reconstruction	26
7.1	Optical Digitization	26
7.1.1	The OpMCDigi digitization module	26
7.1.2	The OpDigiProperties Service	27
7.1.3	The OpDigiAna analysis module	28
7.2	The Hit Finder Reconstruction Module	28
7.2.1	The OpHitFinder module	28
7.2.2	The OpLowIntensityHitFinder module	29
7.2.3	The OpHitAna analyzer	30
7.3	Flash Finding	30
7.3.1	The OpFlashFinder module	30
7.3.2	The OpFlashAna analyzer	30
8	Conclusion	30

1 Introduction

1.1 Introduction to the LArSoft Optical Simulations

LArSoft is the software framework which supports the US liquid argon TPC experiments ArgoNeuT, MicroBooNE and LBNE. The package is designed to be as transportable as possible to other current and future liquid argon TPC detectors, with minimal effort : specification of a detector geometry and a small number of detector specific settings is, in the ideal case, all that is required to begin simulation and reconstruction tasks with a new liquid argon TPC experiment.

Optical simulation involves modeling the light emitted by charged particles moving in argon as it interacts with the detector and eventually produces detectable signals in optically sensitive elements, such as photomultiplier tubes. At the time of writing, LArSoft incorporates two methods for simulating light produced and propagating in the detector.

The “full optical simulation” implements a Geant4 simulation of individual optical photons produced along the path of charged particle tracks through both scintillation and Cherenkov processes. These photons are stepped around the detector through rayleigh scatterings, reflections and partial absorption, in order to produce a realistic detector response to the light source. This simulation has been carefully implemented and steadily improved over a two year period to ensure that it is efficient, flexible and adheres to LArSoft and Geant4 coding conventions.

Due to the vast number of photons typically produced in a neutrino physics event, the full optical simulation can take hours or days per event. The “fast optical simulation” mode was developed to overcome this problem for routine simulation tasks. This mode utilizes a library of stored visibility

data to sample an expected detector response given an isotropic emission of light at some point in the volume. Since the full details of rayleigh scattering, reflection, absorption, etc were treated when building the library, this response also incorporates all these effects. The fast simulation typically takes minutes rather than hours to complete. However, only isotropic scintillation light can be handled in this way. Cherenkov photons are directional, and so we cannot take advantage of the direction-averaged properties provided by the fast scintillation library. Methods for dealing with this limitation are discussed in subsequent sections.

Both simulation modes utilize the same detector geometry specification and produce the same data products. Therefore, the outputs from the two simulations can be used interchangeably. As well as the simulation methods themselves, a suite of tools for optical photon analysis and reconstruction have been implemented in LArSoft, and these are also described in this technical manual.

1.2 A Note on LArSoft Code Structure

Wherever possible, optical tools have been developed to adhere to LArSoft code structuring conventions. More detail about these can be found on the LArSoft wiki pages online. All LArSoft code is written in C++, and I will assume familiarity with C++ and basic object orientated design. In this section I briefly describe a few LArSoft principals which will help the uninitiated understand the structure of optical code.

Code in LArSoft is divided into packages. Each package contains one or more source code files. Usually, one file contains a description of one class. Every package also has its own C++ namespace, in which all of the classes defined in that package should live. The namespace name usually maps to the first few letters of the package name, so for example, code that defines the `sim::PhotonVoxels` class can be found the in Simulation package, which exists in the trunk/Simulation directory of the LArSoft repository. A list of all the packages in LArSoft, and their revision numbers at the time of writing are given in figure 1. This information may help to alleviate confusion in cases where what I describe in this note has gone out of date - the developer can look at the changes between the version here described and the current version to see specifically what changes have been made.

There are several types of classes in LArSoft. I now quickly describe four I will refer to often: modules, services, helper classes and data products. A module is a self contained piece of code which can be run as part of a job sequence - for example, an event generator or a reconstruction algorithm. It usually has a parameter set which is supplied in an fcl file, and there is usually one fcl file containing the default parameter sets of all modules in a package.

A service is a singleton class which is available globally to all packages in LArSoft, and often supplies useful information or methods to various packages. An example is the Geometry service, which allows all LArSoft packages to access information about the detector geometry. Services also have a parameter set supplied in an fcl file.

A helper class is neither a service nor a module, but is a class which is used by other services or modules for some particular purpose. The more complicated LArSoft packages, in particular LArG4, are built from complicated webs of helper classes which are all controlled by one module. This module may access services from both its own package and other packages.

Data products are any classes which are stored into the event. In LArSoft there is no special data product base class, but where possible these classes are kept flat and simple, coded in strict accordance with art framework standards, and the introduction of new data product classes is carefully controlled. All data products live in a few specific LArSoft packages, which exist solely for the purpose of keeping a complete catalogue of such classes. These are the RawData, RecoBase, Simulation and AnalysisBase packages.

1.3 LArSoft Packages with Optical Features

Table 1 lists the LArSoft packages are involved in the implementation of optical simulation and reconstruction. Where the specific files column contains a *, this means that optical detector code is

root / trunk

Name	Size	Revision
AnalysisBase		3418
AnalysisExample		3422
CalData		3375
Calorimetry		3368
ClusterFinder		3464
DetSim		3367
EventDisplay		3494
EventFinder		3440
EventGenerator		3495
Filters		3383
Genfit		3163
Geometry		3499
HitFinder		3430
LArG4		3502
MCCheater		3454
Monitoring		3353
OpticalDetector		3507
ParticleIdentification		3369
PhotonPropagation		3506
RawData		3493
RecoBase		3503
SRT_LAR		3484
ShowerFinder		3398
SimpleTypesAndConstants		3191
Simulation		3468
SummaryData		3420
TrackFinder		3456
Utilities		3476
VertexFinder		3380
setup		3491

Figure 1: The packages in the LArSoft repository and their revision numbers at the time of writing

implemented extensively throughout the package rather than in a few specific files.

2 Optical Geometry in LArSoft

2.1 Optical Components within the GDML Description

All geometry in LArSoft is specified in the GDML language. For each LArSoft experiment there exist a set of scripts to build the geometry of the detector in gdml language, according to a set of prescribed parameters (for example, TPC dimensions, number of wires, etc). At the time of writing, only MicroBooNE geometry has implemented optical systems, and they are also arranged in the global gdml file with a series of scripts.

The section of gdml representing optical components for MicroBooNE is generated by the scripts in the `Geometry/gdml/microboone` directory, including `gen_pmt.fcl`, `gen_pmtrack.pl`. The details of the geometry of a single PMT assembly are contained in the `gen_micropmt.gdml` file, and coordinates representing the centre and direction of each assembly are supplied in the `pmtcoordinates.csv` file. After

Package (/subdir)	Specific Files	Features
RawData	OpDetPulse	Optical system raw data product
RecoBase	OpHit, OpFlash	Optical reconstruction data products
Simulation	SimPhoton, PhotonVoxels	Simulated MC truth photon objects, photon voxelization scheme
Utilities	LArProperties + supporting files	Supply of argon optical properties to LArSoft
EventGenerator	LightSource + supporting files	Optical light source event generator
EventDisplay	RecoBaseDrawer	OpFlash evd drawing method
Geometry	*	Details of OpDet geometrical configuration for access through geometry service
/gdm1	*	The gdm1 implementation of optical components for specific detector geometries
LArG4	*	Implementation of optical simulations
PhotonPropagation	*	Fast sim methods, photon library tools
/LibraryData	*	Optical photon library data files
/LibraryBuildTools	*	Scripts and configs for grid submission, library building, etc
OpticalDetector	*	Digitization and low level reconstruction methods, analysis tools

Table 1: List of LArSoft packages where optical simulation and reconstruction code is implemented

building each gdml fragment in turn, the full geometry can be zipped together into one single gdml file to be loaded into LArSoft using the `generate_gdml.pl` script.

The geometry file to use in a LArSoft job is specified as a string to the `geo::Geometry` service, as the “GDML” parameter. To enhance performance for large detectors, a second geometry can be built without any wire placements. This geometry file is used by Geant4 only, where the precise positions of wires is largely unimportant to the physics of particle stepping. If the parameter “DisableWiresInG4” is set to true in the geometry service, a geometry with the same file name but “_nowires” appended before the .gdml file extension will be searched for and passed to Geant4 instead of the default with-wires geometry. For most accurate optical simulations, and specifically library building jobs, it is recommended that this mode not be used, as some optical photons are inevitably blocked by the wireplanes - an effect which is not accounted for if a wire-free simulation geometry is utilized.

Any logical volume in the gdml description with a name containing “volOpDetSensitive” will be assigned a unique optical detector channel ID, and assumed to represent an optical sensitive detector element in LArG4 (and elsewhere). This name is hard coded into the Geometry module, and supplied when required to LArG4, or other modules requiring access to the GDML description, via the function `Geometry::OpDetGeoName()`. If there are multiple placements of this volume, each will be assigned its own sensitive detector ID.

2.2 Optical Component Handling in the LArSoft Geometry Service

The `geo::Geometry` service is a store of geometry information and tools accessible by all modules in LArSoft. The optical detector geometry is accessible through this service, and most modules making use of optical geometry acquire the information they need about optical detector IDs, orientation and placement from here. The LArSoft geometry description is hierarchical, with one geometry containing one or more cryostats, which each contain one or more TPCs. Each cryostat also contains zero or more OpDets, which each represent a single optical detector unit. This scheme is shown diagrammatically in figure 2.

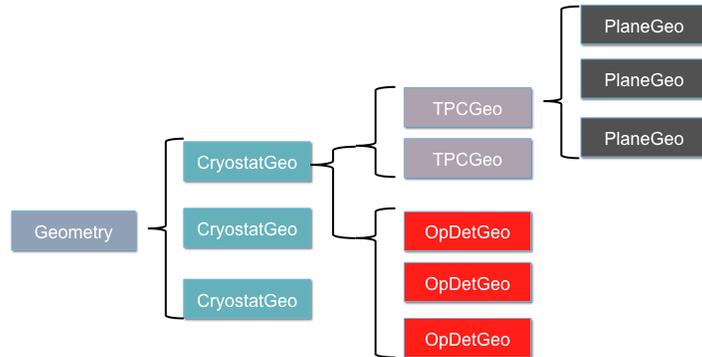


Figure 2: Hierarchy of objects in the `geo::Geometry` service

Just as a single wire in the detector can be referred to either by its channel ID, or by its coordinates $[c,t,p,w] = [\text{cryostat}, \text{tpc}, \text{plane}, \text{wire}]$, an optical detector can be referred to either by its optical channel ID, running from 0 to N in the order the physical volume was loaded into the LArSoft geometry table, or by its coordinates $[c,o] = [\text{cryostat}, \text{opdet}]$. The functions `Geometry::OpDetCryoToOpChannel(o, c)` and its inverse function `Geometry::OpChannelToCryoOpDet(channel)` provide a mapping between these two schemes. In a detector with only a single cryostat, the optical channel ID will in general be identical to the opdet number o. However, in the interest of ensuring compatibility of tools with multi-cryostat geometries, developers are always encouraged to use the conversion methods in Geometry to supply the OpDet ID in the correct coordinate system for the purpose at hand.

The number of OpDets in each cryostat can be queried with the function `Geometry::NOpDet(c)`. If `c` is not specified, the number of OpDets in the first (or only) cryostat is returned.

The `geo::OpDetGeo` object holds information about each optical detector, and is accessible from the `geo::Geometry` service through the chain `geo::Geometry->Cryostat(c)->OpDet(o)`. As well as containing information about the OpDet’s position and orientation, this class also holds some simple geometry methods for calculation of distances to points in the detector and solid angle factors. Any geometrical calculations developed for OpDets which can be of wider use in LArSoft in future can and should be added to this class.

2.3 Voxelization Schemes

Several LArSoft modules, including the fast optical simulation, require optical voxelization of the active detector region. A voxel is a cuboidal region of 3D space - analogous with a pixel in 2D. So voxelization effectively means making a coarse graining of the detector into individually labeled volume units. The details of a voxelization scheme for an optical volume are represented by a `sim::PhotonVoxelDef` object. This object provides a map between a unique identifier, `VoxelID`, and a cuboid in XYZ space. The spatial limits of the voxelized region and the number of voxels to divide it into in each direction are supplied through the objects constructor. Helpful geometrical methods for dealing with the voxelized space are provided in this object and its helper object, `sim::PhotonVoxel`. These include methods for finding the voxel ID a given point lies inside, finding the center and edge coordinates of a particular voxel, getting the total voxel count, etc. These methods are mostly self explanatory, so please see the source code for more information.

In order to prevent discrepancies between modules, one global voxelization scheme is provided to all LArSoft modules through the `phot::PhotonVisibilityService` service, which owns a globally accessible `sim::PhotonVoxelDef` object. The voxelization parameters are specified in the parameter set for this service, and can be accessed through the method `PhotonVisibilityService::GetVoxelDef()`. This service is also responsible for all interfaces with the `PhotonLibrary` within LArSoft, and this is what most of the code in its `.cxx` file is dedicated to - we will discuss with this aspect later, in section 6.

The parameters relevant to voxel specification in the `fcl` file are listed in table 2. Note that the `UseCryoBoundary` parameter, if set to true, will define the limits of the voxelized space to be the outer edges of the cryostat, as queried from the `Geometry` service. In this case, the supplied `Max` and `Min` parameters will be ignored. If `UseCryoBoundary` is set to false, the voxelized region will use the boundaries specified. For most purposes, including running the fast optical simulation, voxelization of the entire detector area is required and so `UseCryoBoundary` will be set to true. In special cases, for example if we wanted to voxelize the TPC region only, we could specify its coordinates manually.

Parameter	Type	Purpose
<code>NX, NY, NZ</code>	int	The number of voxels in the X, Y and Z directions
<code>XMin, XMax, YMin, YMax, ZMin, ZMax</code>	double	Boundaries of the custom voxel region
<code>UseCryoBoundary</code>	bool	Whether to use a custom region (0) or full cryostat (1)

Table 2: Voxelization parameters specified to the `PhotonVisibilityService` parameter set

3 Optical Data Products

This section briefly describes the data types in LArSoft which store the output of optical simulations. The methods for producing and filling these objects will be the focus of subsequent sections. According to LArSoft coding conventions, all data products stored in LArSoft events are held in the `Simulation`, `RawData`, `RecoBase` and `AnaBase` packages, and where possible should be simple, “flat” classes with only basic datatypes.

3.1 Products in the Simulation package

The simulation package contains three optical simulation objects representing MC-truth level information. They form a hierarchy, with `sim::SimPhotonsCollection` representing many `sim::SimPhotons` objects indexed by optical detector ID, and `sim::SimPhotons` being a collection of `sim::OnePhoton` objects, each representing a single detected photon. This is shown in figure 3.

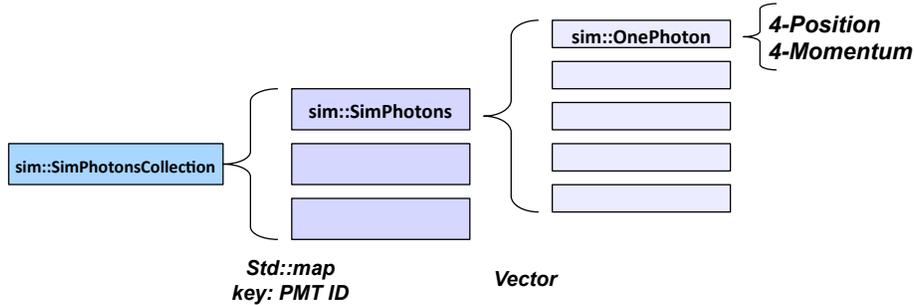


Figure 3: Hierarchy of data objects in the Simulation package

3.1.1 The `sim::OnePhoton` class

This object is the basic data product output by the LArSoft detector monte carlo (LArG4). It represents one photon which has reached an optical detector element. This “detected” photon may have been simulated in several ways, and the information content is somewhat dependent upon how it was produced.

In particular, if the `sim::OnePhoton` was produced by stepping a real Geant4 photon around the detector, as in the full optical simulation or cerenkov simulation, the `SetInSD` flag will be set to true. In all cases, `InitialPosition` gives the production point of the photon. The time data member gives, in the full simulation, the accurate detection time of the photon. In the fast simulation, however, the precise optical path taken is unknown, so the time reported is the production time, which inserts an error of $O(10\text{ns})$ into the detection time measurement. Finally, in the full simulation, `Energy` gives the photon energy at production, sampled from the scintillation spectrum provided. In the fast simulation, the energy of the photon is set to a constant 9.7eV, which is the peak of the argon scintillation spectrum.

The `sim::OnePhoton` object does not store an optical channel ID, as it is always stored as a part of a `sim::SimPhotons` object, which indexes the individual photons per optical channel.

Member	Type	Meaning	Notes
InitialPosition	TVector3	(x,y,z) of photon production	
Energy	double	Energy of photon at production	
Time	double	Time of the photon	Fastsim: production time. Fullsim: detection time
SetInSD	bool	Was this photon set in a sensitive detector?	0 for fast sim photons, 1 for G4 stepped photons

Table 3: Data Members of the `sim::OnePhoton` object

3.1.2 The `sim::SimPhotons` class

This object represents a collection of photons detected by one optical detector element. It is an overloaded `std::vector<sim::OnePhoton>`. This means that it can be used exactly as an STL vector of single photon objects, with the normal `[], push_back()` and iterator operations. It has two pieces of added functionality. First, it stores the optical channel ID, which is the unique ID assigned to each

optical channel in the detector geometry. Second, it has overloaded operators for adding together two `sim::SimPhotons` objects. This operation corresponds to combining the two vectors into one vector containing all of the original `sim::OnePhoton` objects. A vector of `sim::SimPhotons` objects is stored in the event by LArG4 at the end of an optical simulation event.

Member	Type	Meaning	Notes
<code>fOpDetChannel</code>	int	Channel ID for this optical detector	
<code>sim::SimPhotons[0...n]</code>	<code>sim::OnePhoton</code>	One object for each photon detected	Get photon count with <code>.size()</code>

Table 4: Data Members of the `sim::SimPhotons` object

3.1.3 The `sim::SimPhotonsCollection` class

This object is a helper object which represents all of the photons detected by all of the optical detectors in the event. It is an overloaded `std::map<int, sim::SimPhotons*>`, with each `SimPhotons` object indexed by its unique optical detector channel ID. In the latest version of LArSoft, this object does not contain additional particularly useful information beyond a simple collection of `sim::SimPhotons` objects, and so is gradually dropping out of the code base - however, the multiple cryostat geometry of LBNE may require such an object be reinstated. The additional functionality above a standard map are a string to hold the name of the sensitive detector in which this photon collection was produced, and overloaded operators for adding together `SimPhotonCollections` (which combine the photon collections for each sensitive detector in both objects into a single `sim::SimPhotonsCollection` object).

3.2 Products in the `RawData` package

The `RawData` package contains objects which resemble or are equivalent to data read out from a detector. There is one optical simulation object in `RawData`, which may require tweaking or expansion when we there are real DAQ interfaces.

3.2.1 The `raw::OpDetPulse` class

This object represents a single waveform as read out from an optical detector. It contains a unique ID identifying the optical detector in question, and a vector of shorts representing ADC samples. In order to facilitate suppression of long tails of zeroes in the sample vector, the number of samples is stored as an unsigned short.

Member	Type	Meaning
<code>fOpDetChannel</code>	unsigned short	Channel ID for this optical detector
<code>fSamples</code>	unsigned short	number of samples in this waveform
<code>fWaveform</code>	<code>std::vector<unsigned short></code>	Vector of ADC values

Table 5: Data Members of the `raw::OpDetPulse` object

3.3 Products in the `RecoBase` package

The `RecoBase` package contains “reconstructed” objects. We interpret this to mean only objects which can be produced through a chain which begins with raw data, rather than requiring MC truth. Optical reconstruction will be the focus of much work in the future, but at the time of writing this note there are two simple reconstructed objects in this package.

3.3.1 The recob::OpHit class

A recob::OpHit represents a single detected pulse in an optical detector, and may represent many photoelectrons. The OpHit is a completely “flat” object, merely holding data members but not performing any processing. It stores several pieces of information redundantly (for example, pulse area and number of PE). Eventually we expect this redundancy to be lifted once more advanced digitization methods, accounting for system nonlinearities for example, are implemented.

Member	Type	Meaning
fOpDetChannel	int	Channel ID for this optical detector
fPeakTime, fPeakTimeError	double	Time of peak of pulse in ns (+error)
fWidth, fWidthError	double	Time width of pulse in ns (+error)
fArea, fAreaError	double	Area of pulse in ns * ADC (+error)
fAmplitude, fAmplitudeError	double	Amplitude of pulse in ADC counts (+error)
fPE, fPEError	double	Number of PE in pulse (+error)

Table 6: Data Members of the recob::OpHit object

3.3.2 The recob::OpFlash class

A recob::OpFlash represents an approximately simultaneous pulse of light observed by several optical detector elements. One OpFlash is assumed to correspond to one distinct interaction or cosmic ray in the detector, and is the basic unit which is matched to track objects to find their drift time coordinate.

Member	Type	Meaning
fTime	double	The time in ns at which the flash occurred
fPEPerPMT	std::vector<double>	A list of the number of PE seen by each optical detector
fYCenter, fYWidth	double	Center and width of pulse in the y direction, based on PMT positions
fZCenter, fZWidth	double	Center and width of pulse in the z direction, based on PMT positions
fWireCenters, fWireWidths	std::vector<double>	Center and width of pulse in the y direction projected onto wire pitch directions
fOnBeamTime	int	Flag stating whether pulse is in beam window. 0=no, 1=yes

Table 7: Data Members of the recob::OpFlash object

4 Simulation and Analysis Tools

4.1 The LightSource event generator

This module lives in the EventGenerator package and provides a source of an isotropically produced Geant4 photons from some area in the detector. There are two modes in which the light-source can be run, which are illustrated in cartoon form in figure 5 and can be selected between with the “SourceMode” parameter. All parameters described in this section are supplied in the EventGenerator/lightsource.fcl file.

SourceMode = 0 specifies file mode, where the light source is placed event by event in locations specified by an input text file. This mode is useful for scanning the light response for particular regions of interest - for example, near to light-blocking elements like field cage supports, or to quickly scan a few representative positions in the TPC volume. To use the light source in file mode, a filename must be specified via the “SteeringFile” parameter in the modules parameter set. An sample guiding file, LightSourceSteering.txt is provided in the EventGenerator package, and an excerpt is shown in figure . The top line of the file is ignored and contains column titles. On subsequent lines, the required

format is tab separated values, with one record per line containing $\{x,y,z,t,dx,dy,dz,dt,p,dp,n\}$, where these represent central coordinates of the region $\{x,y,z,t\}$, extent of the region about this central value $\{dx,dy,dz,dt\}$, the central momentum of photons $\{p\}$, width of the momentum distribution $\{dp\}$, and the number of photons to generate in this region $\{n\}$. The file is read one line per event, and when all have been read, the light source starts again at the beginning of the list.

root / trunk / EventGenerator / LightSourceSteering.txt

History | View | Annotate | Download (252 Bytes)

1	x	y	z	t	dx	dy	dz	dt	p	dp	n
2	0.0	0.0	0.0	0.0	1.0	1.0	1.0	1.0	9.7	0.1	1000
3	1.0	0.0	0.0	0.0	1.0	1.0	1.0	1.0	9.7	0.1	1000
4	2.0	0.0	0.0	0.0	1.0	1.0	1.0	1.0	9.7	0.1	1000
5	3.0	0.0	0.0	0.0	1.0	1.0	1.0	1.0	9.7	0.1	1000
6	4.0	0.0	0.0	0.0	1.0	1.0	1.0	1.0	9.7	0.1	1000

Figure 4: Excerpt from the sample LightSource steering file, LightSourceSteering.txt

SourceMode = 1 specifies scan mode. A defined region is divided up into voxels, and light is produced in each voxel one event at a time. The lightsource can be instructed to either use the global voxelization scheme as supplied by the PhotonVisibilityService (see section 2.3), or with a custom voxelization by supplying the region size and number of steps in each dimension. To use scan mode, the user must also supply the central momentum and momentum width of photons to fire, the central time and time interval in which to generate them.

Given a particular voxelization scheme the generator can be instructed to only step over a limited range of voxels in the geometry. This feature is particularly useful for grid based jobs, where we may want to sample a different part of the detector with each instance. The first and last voxel IDs to step through are defined by the “FirstVoxel” and “LastVoxel” parameters. If LastVoxel is set to -1, the full range of voxels will be used.

Finally, there are three parameters called PosDist, PDist, TDist which specify how the randomly drawn positions, times and momenta should be distributed. A value 0 indicates a uniform distribution over the interval supplied, and a value 1 indicates a gaussian distribution of the specified width and center.

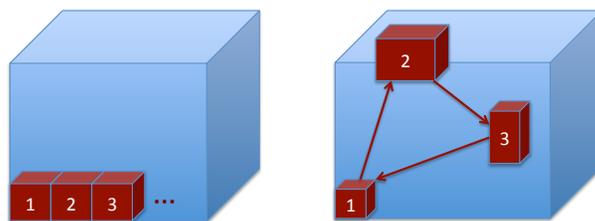


Figure 5: Cartoon illustrating the two modes of the lightsource event generator. Left: scan mode. Right: file mode.

4.2 The SimPhotonCounter analysis module

After a monte carlo job has been run, either through the full or fast simulation, any photons which arrive at a volume designated as a sensitive optical detector are used to produce SimPhoton objects (section 3.1). The `opdet::SimPhotonCounter` module is an analyzer which extracts information from these objects and writes it to TTrees for offline analysis. If required, a wavelength cut and a quantum

Tree Name	Entries	Contents
AllPhotons	Per photon	Event ID, Wavelength, OpChannel, Time
DetectedPhotons	Per photon after QE	Event ID, Wavelength, OpChannel, Time
OpDets	Per OpDet	Event ID, OpChannel, photons at OpDet, photons detected
OpDetEvents	Per Event	Event ID, total photons at OpDet, total photons detected

Table 8: The four TTrees which can be output by the SimPhotonCounter module

efficiency can be applied, to sample a random subset of the incident photons at the optical detector. A plot generated with the PerOpDetTree is shown in figure 6

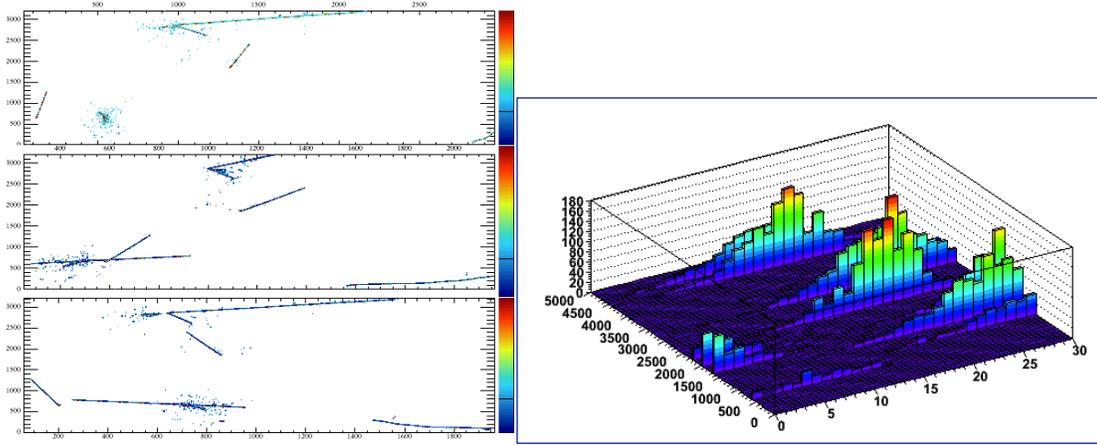


Figure 6: The right shows a plot generated with information from the DetectedPhotons tree of the SimPhotonCounter module. The number of photoelectrons recorded by each PMT as a function of time is shown for an event containing a neutrino interaction and cosmic rays. The left image is the 3 wireplane event display of the same event. Note that the optical window is 3 frames long, even though only one readout frame is shown in the event display. This is necessary since out-of-frame-time cosmic rays can still be visible in the beam triggered readout frame.

Care must be taken when using the QuantumEfficiency feature, for several reasons. For some applications quantum efficiency can be set further upstream - for example, by dropping the scintillation yield in argon by a relevant factor to speed up full simulation computations. In this case, MC truth photons all represent detected photoelectrons. Conversely, for the fast simulations, quantum efficiency losses are usually not applied until the digitization stage. In this case MC truth photons represent photons arriving at a sensitive element, of which some fraction will be detected. This is the more accurate way to treat the quantum efficiency, as it ensures that Cerenkov photons and scintillation photons both suffer the same fractional cut, whereas simply dropping the scintillation yield only affects the latter set.

The analyzer can output up to four TTrees, the contents of which are described in table 8. There are flags to switch each tree on or off in the modules parameterset. If not interested in per-photon information, it is advisable to switch off the AllPhotonsTree and DetectedPhotonsTree as in some cases many photons are detected and this will cause the output file size to explode.

5 The Full Optical Simulation in LArG4

As interdicted in section 1.1, the full optical simulation involves producing and tracking individual scintillation and cerenkov photons in Geant4. The Geant4 particle representing a photon is the

G4OpticalPhoton. Capabilities for enabling this particle and its attached physics processes have been implemented in LArSoft, as well as methods for handling the detection of such particles. Typical events involve stepping millions of optical photons, so the full simulation is very computationally intensive. Wherever possible, effort has been made to push the efficiency of the simulation as hard as possible. LArSoft’s interface to Geant4 is the module LArG4 and so this package is where the majority of the relevant code lives.

5.1 Configuring and Running the Full Optical Simulation

The sample fcl file `EventGenerator/prodsingle_fulloptical.fcl` shows how to configure and run the full optical simulation.

The parameter sets which control the Geant4 simulation in LArSoft are supplied to two services which are included in the standard `microboone_simulation_services` block, listed in table 9. To configure the full optical simulation, some relevant parameters of these services must be changed from their default values. We describe these parameters in this section.

Pset name	File	Purpose
LArG4Parameters	Simulation/simulation-services.fcl	Control of Geant4 physics list
LArProperties	Utilities/larproperties.fcl	Control of argon optical parameters

Table 9: Parameter sets which control the optical simulation

The following line

```
services.user.LArG4Parameters.UseCustomPhysics = true
```

is used to allow the user to control which items in the Geant4 physics list are switched on and off. Then, optical processes will be included if the line “Optical” is included in the vector of strings

```
services.user.LArG4Parameters.EnabledPhysics = ["Em", "Optical", ...
```

which specifies which physics constructors should be loaded into the simulation. These are the only two parameters which must be set in LArG4Parameters. The LArProperties service controls the parameters of the physics processes which are attached to the liquid argon volume, and so there are significantly more parameters to be set in this service. The settable parameters of the LArProperties parameter set are listed in table 10.

5.2 Physics List Handling in LArG4

In order to enable user controlled switching between different optical simulation implementations (including no optical physics enabled), a configurable physics list system was implemented in LArG4. Within this system, physics processes and particles are divided into (mostly) self contained blocks which inherit from the Geant4 class `G4VPhysicsConstructor`. When a `G4VPhysicsConstructor` is enabled in the LArSoft physics list, the particles and processes it contains become a part of the simulation. For an example physics constructor in LArSoft, see `larg4::OpticalPhysics`.

A list of all compiled `G4VPhysicsConstructors` is maintained by the `CustomPhysicsTable` object, which is a singleton class in the LArG4 package. A `G4VPhysicsConstructor` is registered with the table using a `CustomPhysicsFactory` object. This templated helper object is instantiated in the `.cxx` file of the physics constructor, and at runtime time passes the `G4VPhysicsConstructors` constructor method with a string to the `CustomPhysicsTable`. This scheme is illustrated in figure 7.

Parameter	Type	Purpose
RIndexEnergies	vec<double>	Energy bin limits of the refractive index
RIndexSpectrum	vec<double>	Values of refractive index per energy bin
RayleighEnergies	vec<double>	Energy bin limits of the rayleigh scattering coefficient
RayleighSpectrum	vec<double>	Rayleigh scattering length per energy bin (cm)
AbsLengthEnergies	vec<double>	Energy bin limits of the bulk absorption spectrum
AbsLengthSpectrum	vec<double>	Absorption length per energy bin (cm)
ReflectiveSurfaceEnergies	vec<double> >	Energy bin limits of the reflection spectra (same for all materials)
ReflectiveSurfaceNames	vec<double> >	Names of reflective surfaces - must match GDDL specification
ReflectiveSurfaceReflectances	vec<vec<double> >	Table of reflectances, [material1 per energy], [material2 per energy] ...]
ReflectiveSurfaceDiffuseFractions	vec<vec<double> >	Table of diffuse vs specular reflectance fractions, format as above
EnableCerenkovLight	bool	Whether to enable Cerenkov production as well as scintillation
FastScintEnergies	vec<double>	Energy bin limits of the fast time constant scintillation emission spectrum (eV)
FastScintSpectrum	vec<double>	Intensities per energy bin of fast time constant scintillation
SlowScintEnergies	vec<double>	Energy bin limits of the slow time constant scintillation emission spectrum (eV)
SlowScintSpectrum	vec<double>	Intensities per energy bin of slow time constant scintillation
ScintResolutionScale	double	Internal parameter for G4ScintillationProcess, not used in LArSoft (=1)
ScintFastTimeConstant	double	Time constant (in ns) of fast scintillation component
ScintSlowTimeConstant	double	Time constant (in ns) of slow scintillation component
ScintBirksConstant	double	Birks constant for scintillation quenching
ScintYield	double	Total (fast + slow) yield of scintillation photons per MeV
ScintYieldRatio	double	Ratio of fast to slow scintillation intensity

Table 10: Parameters relevant to optical physics in LArProperties service

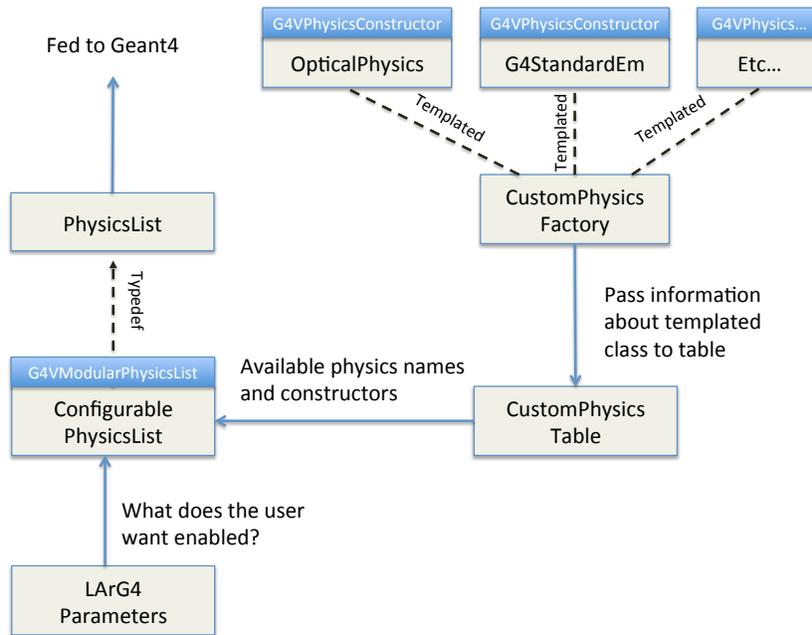


Figure 7: Illustration of the relationship between classes involved in LArSoft physics list handling

No code should ever be added to the physics list handling service explicitly, but new physics constructors can easily be added at any time by creating a class derived from `G4VPhysicsConstructor` and registering it using a `CustomPhysicsFactory` object in its `.cxx` file. For an example, see the implementation of the physics constructor and its corresponding factor object in `LArG4/OpticalPhysics.cxx`.

The default Geant4 physics constructors do not instinctively register themselves with the LArSoft physics list. Therefore, they are registered via physics factories which are defined in the `CustomPhysicsBuiltIns` supporting files.

The `TConfigurablePhysicsList` class inherits from the Geant4 `G4VModularPhysicsList` class. This means it can be populated with physics constructors provided by the `CustomPhysicsTable` and fed to Geant4. This process of passing the physics list to Geant4 is handled by the `G4Helper` object of `nutools`, which since the time of coding has been pulled outside of the LArSoft code-base. However, population of the physics list with the required physics constructors happens in the `TConfigurablePhysicsList`, which obtains all the enabled physics constructor from the `CustomPhysicsTable` and then registers them using the `RegisterPhysics` method of its base class, `G4VModularPhysicsList`.

5.3 The OpticalPhysics Constructor and Associated Physics Processes

An example of a physics constructor is the `larg4::OpticalPhysics` class. This class contains all of the processes and particle definitions relevant to the full optical simulation. Each process is briefly described in the following subsections. The parameters for Geant4 processes are set by the `larg4::MaterialPropertyLoader` class, which itself obtains them from the `util::LArProperties` service, to be described in more detail in section 5.4. In what follows I will specify parameters by their names in the `LArProperties` service, though in some special cases the Geant4 names which these properties are provided to the geant4 material tables as may be different. An interested user can look to the material property loader implementation for a mapping.

5.3.1 Scintillation

The full optical physics list utilizes the default Geant4 `G4Scintillation` process. The parameters required are: “FastScintEnergies”, “FastScintSpectrum”, “SlowScintEnergies”, “SlowScintSpectrum” which define the shape of the scintillation energy spectrum; “ScintResolutionScale”, a Fano factor like parameter; “ScintFastTimeConstant”, “ScintSlowTimeConstant”, the time constants of the fast and slow scintillation light in nanoseconds; “ScintBirksConstant”, a quenching factor for Birks law charge density quenching; “ScintYield”, the total number of photons per MeV produced; and “ScintYieldRatio”, the ratio of photons produced with a fast scintillation time constant to those produced with a slow one.

The scintillation spectrum for the fast and slow components is assumed to be identical and is based on the digitized spectrum from [7], shown in figure 8. The fast and slow time constants and scintillation yield information are taken from [2] and the references therein, and are given in table 11.

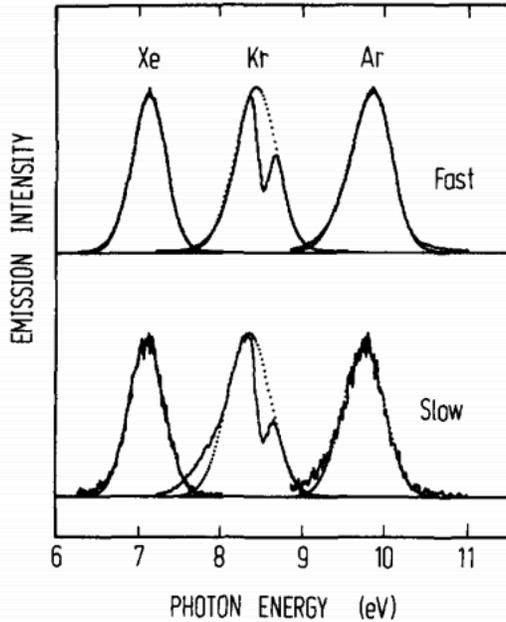


Figure 8: Scintillation spectrum of fast and slow components of noble liquids, from [7]

Scintillation quenching is implemented by the `G4EmSturation` process, which is called as a helper object to `G4Scintillation`. We use a Birks constant of $0.069 / (\text{cm MeV})$, which was extracted from [1]. Scintillation quenching in LArSoft is a topic which has not been exercised or tested at the present time, and will be the subject of more serious future investigations in order to ensure an accurate model of the quenching behavior.

The total scintillation yield is often dropped by a factor of the global PMT quantum efficiency, and then the PMT assembly efficiency set to 1 downstream. This enhances the simulation speed by preventing the undetected fraction of photons from being stepped.

Note that for the The parameters supplied through the `LArProperties` for per-particle-type scintillation apply only to the fast optical simulation. For the full optical simulation we use a single scintillation yield and fast / slow ratio to represent all particles, which is approximately what would be expected for a MIP. This feature could be extended reasonably straightforwardly in the future.

Parameter	Value
ScintResolutionScale	1
ScintFastTimeConstant (ns)	6
ScintSlowTimeConstant (ns)	1500
ScintBirksConstant (1/ cm MeV)	0.0179
ScintYield (photons/MeV)	24000*0.03
ScintYieldRatio	0.25

Table 11: Default parameters for the G4Scintillation process

5.3.2 Cherenkov Light

Cerenkov light is produced with the default Geant4 Cerenkov process, `G4Cerenkov`. The only settable parameter for Cerenkov light production is the refractive index of the material, which is given as a spectrum over a set of energy bins. In LArG4 we set the refractive index of argon over the range visible to standard optical detectors, from around 100 to 700nm. The curve from which this data was extracted comes from references [8, 3, 9] and is shown in figure 9.

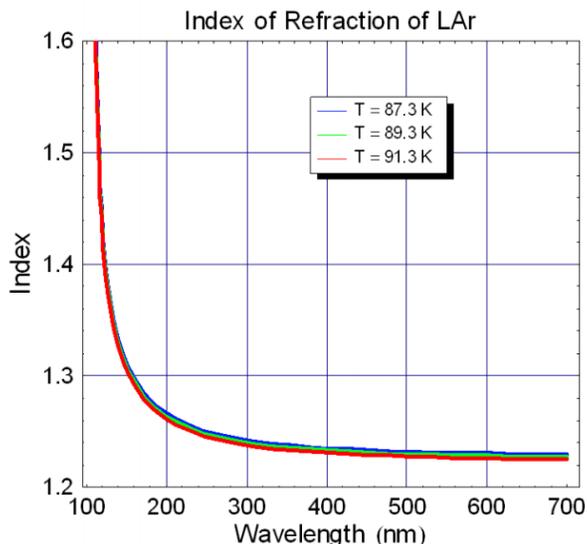


Figure 9: Refractive index of liquid argon as a function of wavelength

5.3.3 Rayleigh Scattering

The rayleigh scattering length in argon at 128nm is theoretically calculated to be 90cm. LArSoft contains the calculated rayleigh spectrum over the wavelength range 110-500nm, in order to ensure correct scattering lengths for Cerenkov as well as scintillation photons. The rayleigh scattering length as a function of photon energy is taken from [10] (and references therein) and shown in figure 10.

5.3.4 Surface Reflection and Absorption

In LArG4 we use a simplified reflection process relative to the default Geant4 `OpBoundaryProcess`, due in part to lack of data for reflectivities at argon / material boundaries. This process is implemented by the `OpBoundaryProcessSimple` class, which inherits from the Geant4 class `G4VDiscreteProcess` and is registered by the `OpticalPhysics` physics constructor. This process produces a certain fraction of

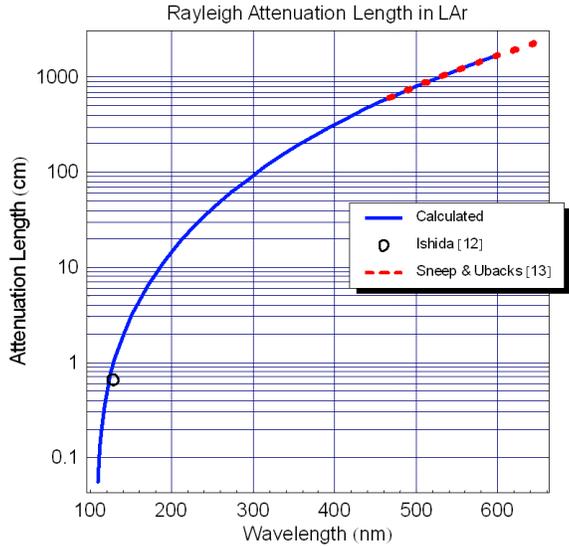


Figure 10: Calculated rayleigh scattering length in liquid argon, with data overlaid. Plot taken from [10]

reflected photons at an interface between argon and the specified material, of which some percentage are reflected specularly (equal angle of incidence to angle of reflection) and the remainder are reflected diffusely (isotropic reflection from the surface). Both the reflection coefficient and diffuse fraction are supplied as functions of photon energy in the LArProperties service configuration.

At present only stainless steel reflections are incorporated, using a constant reflection coefficient of 0.25 with 0.5 diffuse fraction. These are the values of the reflection coefficients at 128nm, taken from [2]. A better model of the energy dependence of this reflectivity, as well as the reflectivity of other materials in the geometry should be implemented at some point. However, most of the photons we care about are at 128nm, and most surfaces are black to these - so the precise details of all component reflectivities should not ultimately be very important.

Table 3
Measured optical parameters for aluminium and stainless-steel, materials at the boundaries of the LAr volume (T600 detector).

	Reflectivity (%) →	Diffusive Refl. (%)	Specular Refl. (%)
Aluminum	23	100	0
Stainless-steel	25	50	50

Table 12: Reflectances at 128nm taken from [2]

5.3.5 Bulk Absorption

There are no natural absorption processes in high purity argon, so the theoretical absorption length at 128nm is infinite. The best published limit says that the absorption length in pure argon is >20cm [5]. Experimentally, it is known to be >90cm, from an unpublished ICARUS analysis, which has its ultimate sensitivity limit at the 128nm rayleigh scattering length of 90cm [4]. Bulk absorption is

expected from impurities such as nitrogen, oxygen and water. In MicroBooNE, the oxygen and water levels will be maintained at ppb levels to prevent a detrimental impact to free electron lifetime, and at these concentrations we can expect them to contribute no significant light absorption. Nitrogen, however, is not monitored, and will be present at several ppm levels. This may be enough to cause some attenuation of 128nm light over long path lengths. At present we assume the path length to be long but not infinite, choosing a value of 20m which is flat over all photon wavelengths. We hope to replace this with measured data taken in Fermilab test stands during spring-summer 2013.

5.4 Custom Material Property Handling

User defined properties relating to the physics of liquid argon are supplied in LArSoft via the LArProperties service. Geant4 physics processes require these parameters to be set in a `G4MaterialPropertiesTable` for each volume in its logical volume store. This task is performed by the `MaterialPropertyLoader` class, which provides an interface between the LArProperties service and the Geant4 volume store. The `MaterialPropertyLoader` reads both wavelength dependent and wavelength independent parameters from the LArProperties service and constructs the Geant4 material properties table for argon. It was coded with an eye to generalizability, and with a small modification it would be easy to also supplying other material property tables, if necessary.

Once all required material properties have been catalogued in the `MaterialPropertyLoader`'s internal data containers, the logical volume store is requested from geant4 and every volume is checked for its material name. Those matching the required material are assigned the relevant `G4MaterialPropertiesTable`. After this has been done, Geant4 physics processes being called can access the parameters they require by querying the properties of the logical volume in which they are stepping.

5.5 Sensitive Detectors, SimPhoton Logging and Parallel Worlds

The geometry information in the `geo::Geometry` service is extracted from the detector `gdml` independently from the detector geometry which exists in LArG4. Hence, whilst the `Geometry` service of LArSoft calculates the `OpDetID` <-> `gdml` volume mapping at the beginning of the job, the LArG4 geometry does not implicitly know which elements are sensitive and what their IDs should be.

The singleton object `OpDetLookup` is responsible for maintaining a table which maps a named Geant4 physical (placed) volume to an optical detector ID. When a sensitive optical volume is designated in LArG4, it is registered with this table to find which ID it should have. The matching is based on the coordinates of the center of the volume element, which should match between LArG4 and `geo::Geometry`.

The sensitive detector handling in LArG4 is implemented according to a standard Geant4 design scheme. Sensitive detector elements are associated with logical volumes in the Geant4 geometry. When a particle steps in such an element, it triggers the `ProcessHits` method of the sensitive detector. The sensitive detector object for optical systems in LArG4 is the `OpDetSensitiveDetector` object. The action taken by this object upon detecting a photon is to figure out which element of the optical system this photon stepped in using the `PhysicalVolume` information extracted from the `g4Step`, and then add a “detected” `MCTruth` photon object to the singleton `OpDetPhotonTable`.

There is a subtlety involved in the sensitive detector scheme, in that the photon detectors exist in a Geant4 parallel world. This prevents particles other than photons from being detected by these objects, and also allows one to prevent photons from seeing the TPC charge voxels which exist in LArG4, giving a great increase in simulation speed. Finally, it avoids problems associated with trying to differentiate between different placements of the same logical volume which have associated sensitive detectors. The implementation of a parallel world geometry for repeated placements of a similar sensitive logical volume is a standard Geant4 design pattern, and more information can be found in the Geant4 manual.

The object `OpDetReadoutGeometry` is a derived class of `G4VUserParallelWorld`. At the beginning of the job, LArG4 registers an instance of the readout geometry with `G4Helper`. This readout geometry object seeks all appropriately named sensitive elements in the Geant4 geometry, and for each one

found, makes a placement of a new logical volume in the parallel world. This logical volume has a unique name and is associated with the optical sensitive detector object. It is also registered with the `OpDetLookup` object, which generates a mapping between the unique volume name and its detector ID, which is calculated from volume center coordinates extracted from the Geometry service. The parallel world scheme in LArG4 is illustrated in figure 17.

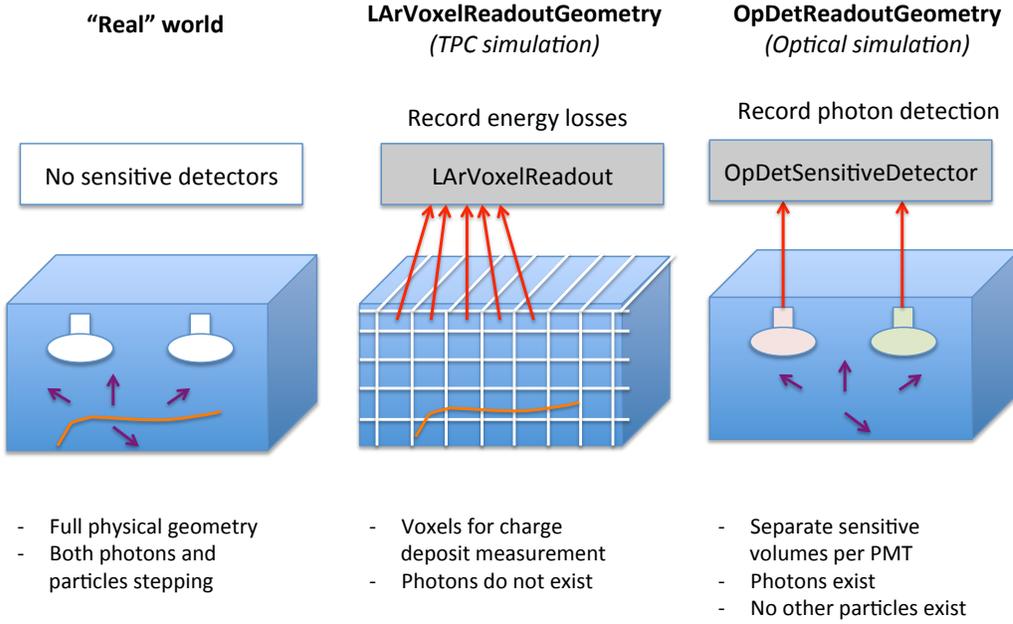


Figure 11: Cartoon of the parallel geometry scheme in LArG4

In the `LArG4/PhysicsList.cxx` file, the `G4ProcessManager` for optical photons is instructed to ensure that optical photons step in the readout geometry as well as the global geometry. No other particles have this property. Hence the `OpDetSensitiveDetector` object will only be activated for optical photons.

The `OpDetPhotonTable`, which accepts detected photon objects (`"sim::OnePhoton"`'s) from the sensitive detectors is read out at the end of event processing by the LArG4 module. The contents of the table are stored in the event. This table accumulates detected photons per optical channel for both full simulation, that is, Geant4 stepped photons, as well as fast simulated library-lookup photons - see section 6 for more information.

6 The Fast Optical Simulation in LArG4

6.1 Configuring and Running the Fast Optical Simulation

The argon properties relevant for the fast optical simulation have significant overlap with those relevant to the full optical simulation, and are also supplied by the `LArProperties` service. The physics constructor which controls the fast optical simulation is called `FastOptical`, is defined by the `larg4::FastOpticalPhysics` class. An example `fcl` script configuring and running the fast optical simulation can be found in `EventGenerator/prodsingle_fastoptical.fcl`.

The `FastOpticalPhysics` constructor list is very similar to the full optical physics constructor, which is described in detail in section 5. The primary difference is that the `G4Scintillation` process is replaced with a larsoft process, `OpFastScintillation`. As with `G4Scintillation`, this process is also called per step of a charged particle, but instead of generating Geant4 trackable optical photons it samples a likely

detected photon response from a pre-built photon library, and adds the relevant detected photons to the `OpFastScintillationTable`.

Cerenkov light cannot be simulated with the fast lookup table since it has a directionality in its production. Therefore, the default Geant4 Cerenkov process can still be enabled in the `FastOptical` physics list by setting the parameter `EnableCerenkovLight`. This results in the stepping of optical photons, and as such causes a significant increase in computational time. Depending on the task at hand, the user can decide whether or not Cerenkov emission is likely to make a significant enough contribution to warrant simulating it photon-by-photon.

Unlike the `G4Scintillation` process, the `OpFastScintillation` process can accept a different scintillation yield per particle type. This feature is enabled if the `ScintByParticleType` flag in `LArProperties` is set to true. At present the allowed particle types are muon, proton, kaon, electron and pion. These yields and fast/slow ratios are taken from [6] (see figure 8) and [2].

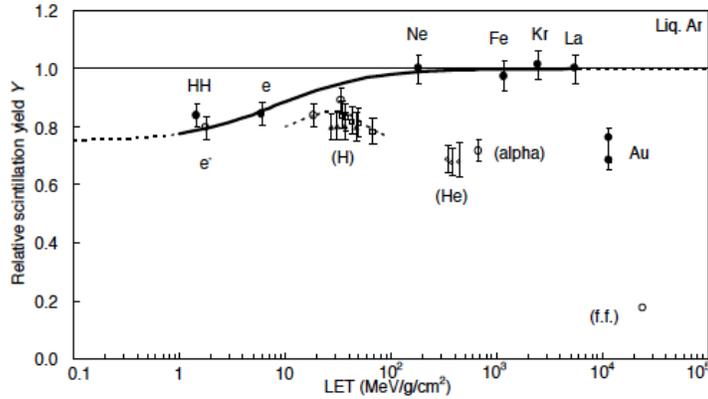


Figure 12: The scintillation yields of different particle types in LAr

6.2 The Mechanics of the Fast Simulation, and the PhotonPropagation Package

The processes of building and querying the fast simulation library are handled by the `PhotonPropagation` package. The central object of this package is the `PhotonVisibilityService`, which provides other modules with an interface through which to interact with the photon library. The photon library data itself is stored in a standalone `PhotonLibrary` object, which has dedicated methods for input and output of the library to disk in a specified external root file.

6.2.1 The PhotonLibrary Object

The `OpFastScintillation` process models the production of some number of photons along a particle step, determined by the particle type and its scintillation yield, as some degree of quenching if Birks constant is specified. Given the location of this step in the detector, a predictable fraction of these photons, on average, will arrive at each optical detector. The fast scintillation process looks up this fraction in the vicinity of the step, and multiplies it by the number of photons produced. This number is poisson fluctuated to obtain a number of photons detected at each optical detector from this charge deposit. The shortness of the steps taken by each particle in Geant4 means that in practice this number is almost always 0 or 1.

The `phot::PhotonLibrary` object represents the entire table of visibilities (defined as photons detected per photon produced) for each optical detector at each voxel in the detector. As well as providing

Parameter	Type	Purpose
RIndexEnergies	vec<double>	Energy bin limits of the refractive index
RIndexSpectrum	vec<double>	Values of refractive index per energy bin
RayleighEnergies	vec<double>	Energy bin limits of the rayleigh scattering coefficient
RayleighSpectrum	vec<double>	Rayleigh scattering length per energy bin (cm)
AbsLengthEnergies	vec<double>	Energy bin limits of the bulk absorption spectrum
AbsLengthSpectrum	vec<double>	Absorption length per energy bin (cm)
ReflectiveSurfaceEnergies	vec<double> >	Energy bin limits of the reflection spectra (same for all materials)
ReflectiveSurfaceNames	vec<double> >	Names of reflective surfaces - must match GDML specification
ReflectiveSurfaceReflectances	vec<vec<double> >	Table of reflectances, [material1 per energy], [material2 per energy] ...]
ReflectiveSurfaceDiffuseFractions	vec<vec<double> >	Table of diffuse vs specular reflectance fractions, format as above
EnableCerenkovLight	bool	Whether to enable Cerenkov production as well as scintillation
ScintResolutionScale	double	Internal parameter for G4ScintillationProcess, not used in LArSoft (=1)
ScintFastTimeConstant	double	Time constant (in ns) of fast scintillation component
ScintSlowTimeConstant	double	Time constant (in ns) of slow scintillation component
ScintBirksConstant	double	Birks constant for scintillation quenching
ScintYield	double	Total (fast + slow) yield of scintillation photons per MeV
ScintYieldRatio	double	Ratio of fast to slow scintillation intensity
ScintByParticleType	bool	If true, each specified particle uses its own scintillation yield and yield ratio. If false, use the global yield and ratio specified by ScintYield, ScintYieldRatio
MuonScintillationYield	double	Total (fast + slow) yield of scintillation photons for muons per MeV
MuonScintillationYieldRatio	double	Ratio of fast to slow scintillation intensity for muons
ProtonScintillationYield	double	Total (fast + slow) yield of scintillation photons for protons per MeV
ProtonScintillationYieldRatio	double	Ratio of fast to slow scintillation intensity for protons
KaonScintillationYield	double	Total (fast + slow) yield of scintillation photons for kaons per MeV
KaonScintillationYieldRatio	double	Ratio of fast to slow scintillation intensity for kaons
ElectronScintillationYield	double	Total (fast + slow) yield of scintillation photons for electrons per MeV
ElectronScintillationYieldRatio	double	Ratio of fast to slow scintillation intensity for electrons
PionScintillationYield	double	Total (fast + slow) yield of scintillation photons for pions per MeV
PionScintillationYieldRatio	double	Ratio of fast to slow scintillation intensity for pions

Table 13: Parameters relevant to optical physics in LArProperties service

Branch	Type	Meaning
Voxel	Int_t	Voxel ID for this entry
OpChannel	Int_t	Optical channel ID for this entry
Visibility	Float_t	The visibility of this voxel to this optical channel

Table 14: Branches of the PhotonLibraryData TTree, stored to disk

methods to look up and set values in the table, the photon library object has I/O methods which can be used to convert the table to and from a flat TTree stored in an external root file. The storage method outputs the library through the TFileService to the global file service output file for the current job. The loading method opens the root file from a string specified location.

The TTree stored to file is in the top directory of the root file, and is named be named PhotonLibraryData. It contains one entry per voxel per channel, and the branches are as shown in table

6.2.2 The PhotonVisibilityService

The PhotonVisibilityService can be called anywhere in LArSoft by using a service handle, as

```
art :: ServiceHandle < phot :: PhotonVisibilityService > pvs
```

If the PhotonVisibilityService is enabled in the services block of the job configuration, at the beginning of the job it will create an instance of the PhotonLibrary object. This object will immediately be filled with photon library data from the path specified by the “LibraryFile” parameter. The location of the file should be specified relative to some directory in the \$FW_SEARCH_PATH environment variable. The default location for the photon library is in the directory PhotonPropagation/LibraryData/.

LArSoft modules can then interact with this PhotonLibrary through methods of the PhotonVisibilityService. In particular, methods are implemented to allow the user to query the visibility per optical detector at a particular voxel number or at a particular 3D point. The names of these methods are self explanatory, so I refer the reader to the source code. These methods are called by the OpFastScintillation process in LArG4 to determine how many detected photons should be given to each optical detector for a given light production intensity.

As mentioned in section 2.3, the PhotonVisibilityService is also responsible for maintaing the global voxelization scheme for the optical simulation. This can be accessed with the

If the LibraryBuildJob parameter is set to true, the visibility service will use the lightsource event generator and sim photon counter modules to coordinate a library building job, a process which is described in the next section. As such, the PhotonVisibilityService is a central pillar of the fast optical simulation.

Some module require access to the PhotonVisibilityService for features such as the global voxel scheme, but do not require access to the photon library. In jobs involving only such modules, job startup times can be improved by setting the DoNotLoadLibrary parameter to true. This prohibits any access to optical library data later in the job.

The full list of configurable parameters of the PhotonVisibilityService are listed below.

6.3 The Library Building Process

6.3.1 Configuring and Running a Library Build Job

The configuration script

```
EventGenerator/prodsingle_buildopticallibrary.fcl
```

Parameter	Type	Purpose
NX, NY, NZ	int	The number of voxels in the X, Y and Z directions for global vox scheme
XMin, XMax, YMin, YMax, ZMin, ZMax	double	Boundaries of the custom voxel region for global vox scheme
UseCryoBoundary	bool	Whether to use a custom region (0) or full cryostat (1)
LibraryFile	bool	Location of root file containing photon library data
LibraryBuildJob	bool	If set to true, LightSource and SimPhotonCounter set up for library build
DoNotLoadLibrary	bool	If true, skip library load from file at job startup

Table 15: Parameters of PhotonVisibilityService

Shows how to set up a library build job. The script has a few vital components. First, the PhotonVisibilityService configured in library building mode

```
services.user.PhotonVisibilityService: @local::standard\_photonvisibilityservice\_buildlib
```

And the LightSource event generator, LArG4 with full optical simulation, and SimPhotonCounter modules are present in the job sequence. Under these conditions, the PhotonVisibilityService will coordinate the production of photons voxel by voxel with the LightSource generator. These photons are simulated, with all optical physics including reflection, rayleigh scattering and absorption accounted for, by the LArG4 full optical simulation. Those photons reaching the optical detectors create sim::Photons objects representing detected photons. The SimPhotonCounter counts the number of photons detected at each optical detector.

With the PhotonVisibilityService configured in library building mode, the SimPhotonCounter produces an output TTree in the correct format for library building, containing the visibility fraction (photons detected / photons produced) for each optical detector at each voxel, using the TFileService output. This TTree, if spanning every voxel in the detector, is the complete photon library object for the fast simulation. In general, each job in a library building batch will only simulate some subset of the voxels in the detector. Then the output TFiles must be zipped together into a single library file. Scripts which can do this are described in the next section.

6.3.2 Library building with the grid

The library building job which has been run for MicroBooNE required 220,000 hours of cpu time. This is obviously a job for the grid, rather than a single machine. Scripts for organizing grid jobs are present in the directory

```
PhotonPropagation/LibraryBuildTools
```

The file prodsingle_buildopticallylibrary.fcl is the base fcl file used to configure the library building job. The script OpticalLibraryBuild_grid.sh copies this file to the remote grid location with the name thisjob.fcl, and modifies it by adding lines to the end. In particular, the job is configured to run over a specified subset of voxels, and deposit the specified number of photons in each, with the lines:

```
echo "physics.producers.generator.FirstVoxel : $FirstVoxel" >> thisjob.fcl
echo "physics.producers.generator.LastVoxel : $LastVoxel" >> thisjob.fcl
echo "physics.producers.generator.N : $NPhotonsPerVoxel" >> thisjob.fcl
```

If the submission is configured properly then each job in the batch runs over a different set of voxels, such that ultimately all voxels are simulated once.

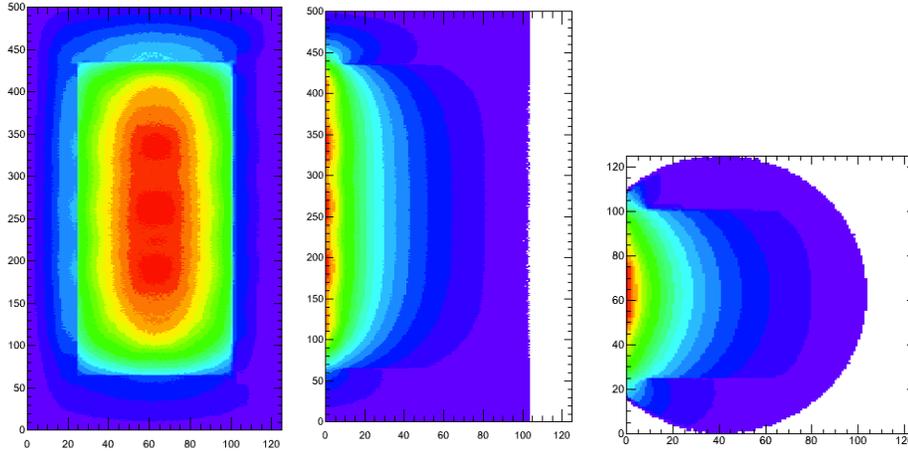


Figure 13: XProjection, YProjection and ZProjection for the MicroBooNE Optical library

The actual command used to submit this job to the grid is supplied for reference in the `SubmitCommand.sh` file. **DO NOT USE THIS SCRIPT BLINDLY.** It is included in the repository as a helpful reference, but running it directly without modifying the relevant file paths may overwrite existing data, or in the best case, attempt to write files to areas the user does not have permissions for.

After all grid jobs have completed, there will be a few thousand output directories copied back to some specified location. The simple script `CopyScript.sh` accesses each of these directories and copies the library output root files into one specified directory. Once this has been done, the `CombineIntoLibrary.sh` script can be run to combine all the individual few-voxel files into one large library file containing data for every voxel. This script runs the `AssembleSingleFile.C` root script, which opens each file in the specified list (generated automatically by `CombineIntoLibrary.sh`) and copies the entries from the TTrees therein into one large TTree, stored in a specified output location.

6.4 The PhotonLibraryAnalyzer Tool

If a photon library file is loaded by the `PhotonVisibilityService`, the analyzer module `PhotonLibraryAnalyzer` can be used to produce visibility maps across the detector. This module only does work at the beginning of the job rather than event-by-event, running through each voxel and querying the sensitivity at each optical detector for that location.

The output is produced through the `TFileService`, and comprises a series of histograms. The XProjection, YProjection and ZProjection histograms show the visibility summed for every optical detector projected down one dimension. For example, the XProjection shows the total visibility summed for every X at each YZ point. The projection histograms are shown for the MicroBooNE optical library in figure 13. These histograms are useful in several ways - for example, at the time of running, there is a small coordinate system bug in the `geo::Geometry` service of LArSoft - this is visible in the Y offset of these projections, and will be fixed in the near future.

As well as producing projections, the `PhotonLibraryAnalyzer` produces series of histograms showing slices through the 3D volume. The slices can be made in X, Y, or Z, and with one slice per voxel in that direction a complete 3D map of the library is represented. Figure 14 shows the X slices at the wireplane, the detector center and the cathode plane, with the z axis (color) showing the total visibility of each voxel summed over all optical detectors.

The `PhotonLibraryAnalyzer` module also produces `XInvisibles`, `YInvisibles` and `ZInvisibles` histograms, which show the number of voxels which are invisible to all optical detectors in the library projected in X, Y and Z. If the statistics of the library generation are high enough, these should not

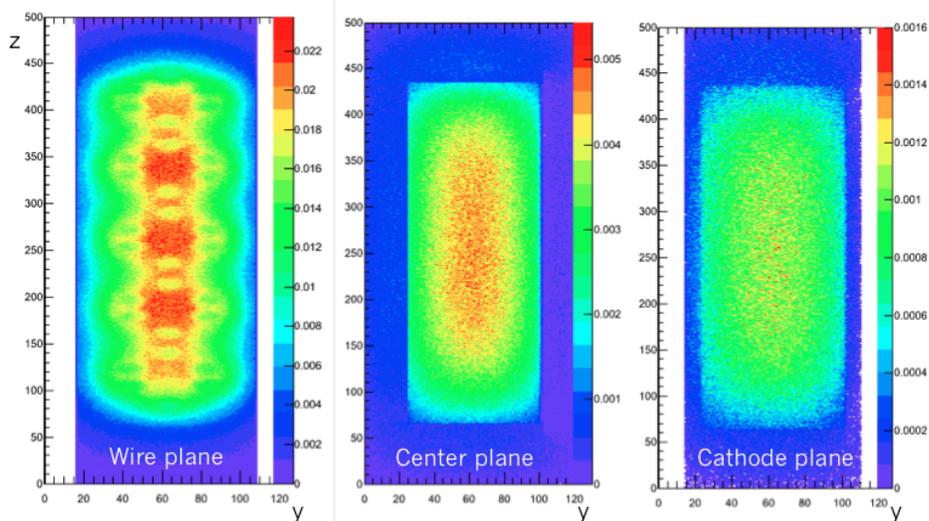


Figure 14: projX histograms for the x slices at the wireplanes, in the center of the TPC and at the cathode plane for MicroBooNE

exist except at legitimately invisible points in the detector (for example, behind the cathode plane or outside the cryostat). Finally, the VisByN histogram shows how many of the optical detectors have a record of the light from each voxel. If the library statistics were high enough to sample the far tails of detectability for each optical detector, this histogram would have all events in bins at 30 (the total number of optical detectors) and 0. In a library built with lower statistics, some voxels will not have recorded visibility data for some optical detectors, and this histogram will not be so strongly peaked at 0 and 30.

7 Digitization and Basic Reconstruction

The `OpticalDetector` package in LArSoft contains digitization modules to turn MCTruth detected photons from the optical simulation into feasible optical detector signal pulses, and reconstruction modules to extract reconstructed quantities from these pulses and make hypotheses about the spatial origin of the light detected in an event. This package is under active development, and this section describes what currently exists, and may be subject to significant change in the future. As such there is less technical detail here than elsewhere in the note, but we include an overview for completeness.

7.1 Optical Digitization

7.1.1 The OpMCDigi digitization module

The OpMCDigi module loads the SimPhoton collection from the event and generates a digitized OpDet-Pulse for each channel. The signal shape for each optical detector is given by a linear superposition of waveforms, one starting at the arrival time of each detected photon. The single PE waveform is obtained from the OpDigiProperties service. If the “QE” parameter is set to less than 1, some fraction of all MC truth photons are discarded from the pulse at random.

After summing waveforms at the relevant photon times, the final pulse is rounded to an integer valued ADC signal. The signal in each bin is allowed to fluctuate either to the value above or below the true value, with a probability for each set such that the expectation value after many samplings would be equal to the true value of the summed pulse.

Parameter	Type	Purpose
InputModule	string	The name of the module which produced SimPhotons - usually LArG4
QE	double	The fraction of photons which will create pulses in each optical detector
SaturationScale	double	The maximum pulse height in the ADC
Dark Rate	double	Rate in Hz of single PE dark noise

Table 16: Parameters of the OpMCDigi module

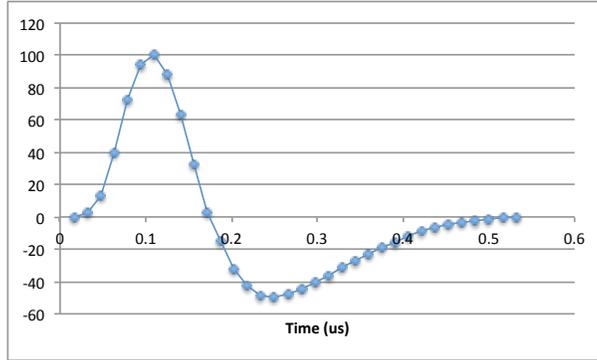


Figure 15: Pulse shape used for optical digitization, as supplied by the OpDigiProperties service

The pulse can be set to saturate at a particular ADC count (“SaturationScale”), which in the MicroBooNE electronics we expect to occur at around 2000 ADC counts. Noise can be injected in the form of single PE pulses at a specified dark noise rate, which will be randomly distributed over the sampling interval and have a number drawn from a poisson distribution determined by the specified noise rate and digitized frame length. The parameters of this module are listed in table 16

7.1.2 The OpDigiProperties Service

This service supplies parameters and waveform shape information to various optical digitization and reconstruction modules. The service is configured with a start and end time for the data frame in microseconds and a sampling rate in MHz. The number of samples per digitized opdet signal will then be the product of the window size and the sampling rate.

A single PE pulse shape to use for digitization is provided in a text file. The pulse currently used is extracted from real PMT readout electronics data for the MicroBooNE Bo PMT test stand, and is shown in figure 15. This pulse can be scaled by a constant (“PERescale”) when reading into the OpDigiProperties service. For example, the pulse shown below represents a signal of height ~ 5 PE, so it is scaled by a factor of 0.2 in order to represent a true single PE pulse. The parameters supplied to the OpDigiProperties service are listed in table 17.

Parameter	Type	Purpose
SampleFreq	string	ADC sampling frequency (MHz)
TimeBegin	double	Time of beginning of digitized frame (relative to MC t=0)
TimeEnd	double	Time of end of digitized frame (relative to MC t=0)
WaveformFile	string	Location of file providing single PE pulse shape
PERescale	double	Factor to scale waveform file by to obtain 1PE pulse

Table 17: Parameters of the OpDigiProperties service

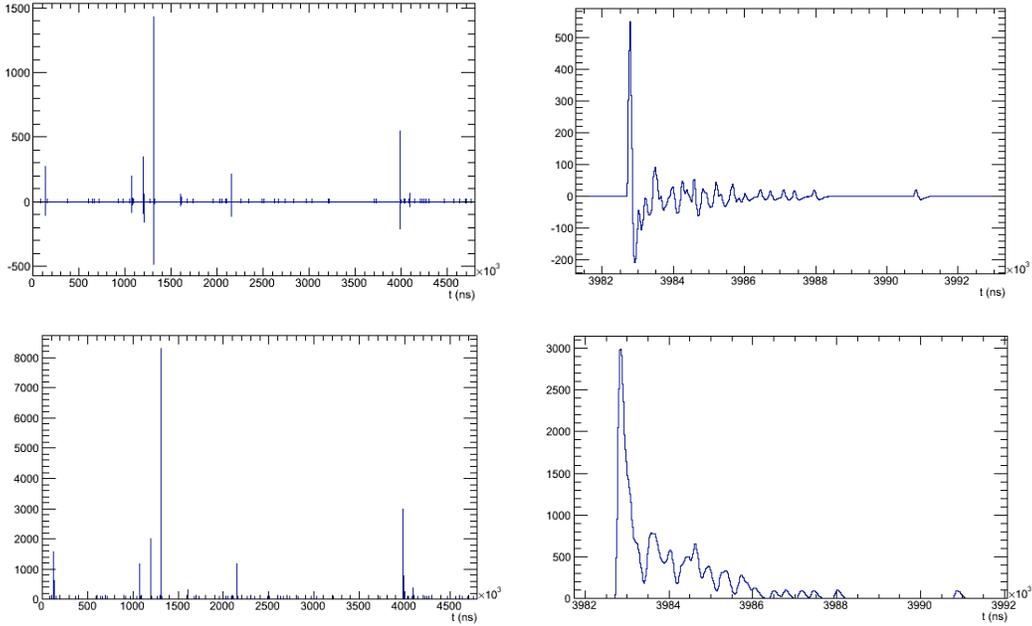


Figure 16: OpDigiAna waveform output for an overlaid cosmic / neutrino event. Top: bipolar pulse over full frame and zoomed into one peak. Bottom: unipolar pulse over full frame and zoomed into the same peak.

7.1.3 The OpDigiAna analysis module

The OpDigiAna module reads the digitized waveforms from the event and stores them as histograms through the TFileService. The only parameter required by OpDigiAna is the input module name, which is used to find the OpDetPulse objects. The pulses are output in two formats : first, as a bipolar pulse, which mirrors the actual pulse simulated and stored in the event. Second, as a unipolar pulse, which is the cumulative sum of the bipolar bins up to each time. Forming a unipolar pulse in this way is utilized in the optical hit finding algorithm. It relies on the fact that the bipolar pulse has a total integral of zero, and so the unipolar pulses for each photoelectron will return to the baseline value. Since the bipolar pulses add linearly, so do the unipolar cumulative pulses.

An example waveform is shown in figure 16. This waveform was generated in an event with several cosmic overlaid with a neutrino event. Each subevent has a distinct signal spike, and zooming in reveals smaller scale structure.

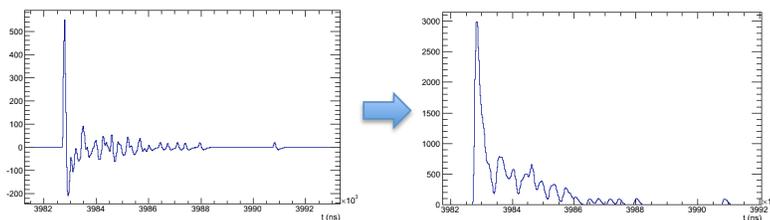
7.2 The Hit Finder Reconstruction Module

7.2.1 The OpHitFinder module

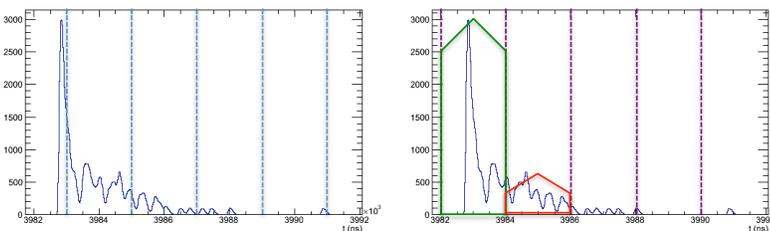
The OpHitFinder module reads in digitized pulses from the event in the form of raw::OpDetPulse objects. It first turns bipolar pulses into unipolar pulses by producing the cumulative pulse as a function of bin time. If the cumulative pulse at a given time is less than a supplied zero suppression threshold, it is assumed to be zero. This is to prevent instabilities from noise and rounding errors from becoming magnified as the accumulated pulse is calculated over long times, in cases where the integral of the bipolar pulses deviates from being exactly zero.

Once the unipolar pulse has been calculated, it is binned broadly into regions of a specified time, usually $1 - 2\mu s$. If one such bin contains an isolated peak, such that the ratio of the bin content either side to this bin content is less than a specified isolation fraction, this bin is used to see a hit. The

1. Invert pulse to unipolar



2. Broad binning and find isolated hit seeds



3. Walk from start of seed to find peak. Walk fixed time past peak to collect late light.

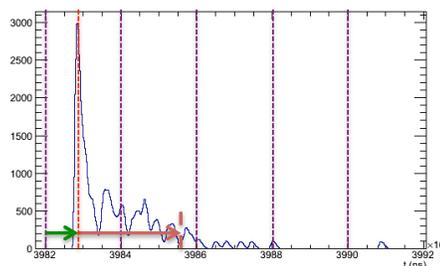


Figure 17: Cartoon of the optical hit finding process

coarse binning is performed twice, the second time with a half bin offset, to prevent missing pulses which straddle a bin division.

Once a seed bin has been found, we walk along the original finely binned unipolar waveform from the start of this bin to find the peak time. Walking along the pulse stops when we have gone one broad bin width from the highest bin found so far. The peak therefore forms a type of “anchor point” around which the pulse is integrated. The pulse therefore usually walks us into the next bin of the broad histogram, which is by design in order to collect the pulse late light in the total area.

The area, peak time and maximum amplitude of the pulse are stored in an OpHit. The number of PE in the pulse is calculated by dividing the area by a specified calibration constant, which represents the area of 1PE as a unipolar pulse. If this is measured in a detector, it is important to invert the pulse to unipolar with the method described before calculating the 1PE area. A cartoon showing this process in action is shown in figure 17.

7.2.2 The OpLowIntensityHitFinder module

The OpLowIntensityHitFinder module is an alternative hit finder, developed to deal specifically with low intensity pulses. The hit finding method finds seeds at peaks in the optical detector signal and then fits gaussians to these pulses. After this first fit, a second fit is performed where the widths and scales of all the individually determined gaussians are varied to account for overlaps. The hit parameters are extracted from the final set of fitted, superposed gaussians. This module has not been extensively tested with full light yield simulations, due to its much higher CPU time requirements for long waveforms.

Parameter	Type	Description
InputModule	string	The module name which produced the digitized pulses
PEArea	double	The integrated area of a unipolar pulse
IsolationFrac	double	The maximum fractional size of the bins either side of a hit
SamplesPerBin	int	The coarse-graining width to bin signals into for hit finding
HitThreshold	double	The minimum number of PE required in a coarse bin to form a hit
ZeroSupThresh	double	The zero suppression threshold used when inverting a bipolar pulse (see text)

Table 18: Parameters supplied to the OpHitFinder module

7.2.3 The OpHitAna analyzer

This analyzer module produces histograms showing the time structure and magnitude (in number of PE) of the OpHits found.

7.3 Flash Finding

7.3.1 The OpFlashFinder module

The purpose of the OpFlashFinder module is to associate OpHits across different PMTs into collections of hits nearby in time which represent distinct optical subevents. The algorithm for doing so is described in the following.

Of all the hits read from the event, the hit with the largest number of PE is selected. All hits which are within a specified flash time width of this hit are collected into a subevent. The multiplicity of hits in this subevent must be greater than a supplied multiplicity condition in order to proceed to create a flash object.

Next, subevents which are near in time and have non-overlapping sets of optical detector channels are merged. The average time of the final subevents is calculated and the number of photoelectrons per optical detector is acquired. Using the known positions of the optical detectors, looked up from the optical geometry, and the known number of PE per detector, the position of the centre-of-mass of the light detected, and the geometrical width of the detected flash, is calculated in the YZ directions and in the UVW directions. The central time of the flash is compared to a user specified beam window to determine whether the flash occurred in time with a beam neutrino event. All the above information is stored into a `recob::OpFlash` object.

7.3.2 The OpFlashAna analyzer

The OpFlashAna analyzer extracts data from the OpFlashes in an event and produces histograms. These histograms include geometrical maps of the flash locations in YZ (the width and center being extracted from the stored flash object, and the flash drawn as a gaussian in these two dimensions), the flash time vs intensity, and the number of PE per flash per optical detector, amongst other plots.

8 Conclusion

In this note I have described the code which has been implemented in LArSoft for both full and fast optical simulation chains, as well as helper modules and services which facilitate optical physics simulations and analyses.

The code base for the simulation routines (largely in LArG4) is likely fairly stable, and this note should remain relevant and useful into the future. Optical digitization and reconstruction routines are in a state of flux and are likely to change significantly once real data begins to be processed from MicroBooNE.

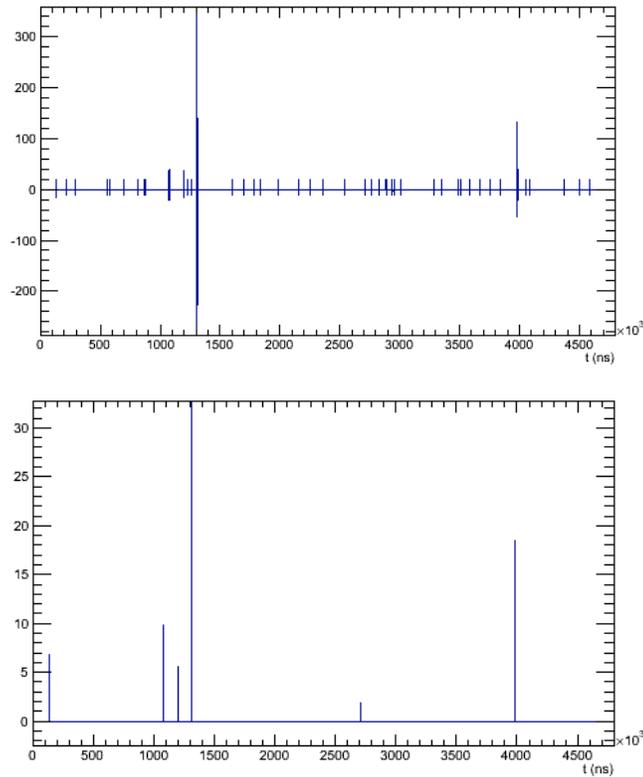


Figure 18: The timing hits found for a particular optical detector in an overlaid neutrino + cosmic event. Left: The digitized pulse from OpDigiAna, Right: the reconstructed hit time structure from OpHitAna. Height of spikes gives #PE. Both plots are cut off vertically to show detail. Note that due to the very fine binning on this scale it is difficult to see the area of the peaks in the digitized signal, although large pulses have visible large initial spikes from prompt light.

Where possible I will attempt to keep this note up to date with future developments in LArSoft optical physics functionality, so check back on the LArSoft svn page often.

References

- [1] S. Amoruso et al. Study of electron recombination in liquid argon with the ICARUS TPC. *Nucl.Instrum.Meth.*, A523:275–286, 2004.
- [2] M. Antonello et al. Analysis of liquid argon scintillation light signals with the icarus t600 detector. Technical Report ICARUS-TM/06-03, 2006.
- [3] A. Bideau-Mehu, Y. Guern, R. Abjean, and A. Johannin-Gilles. Measurement of refractive indices of neon, argon, krypton and xenon in the 253.7 to 140.4 nm wavelength range. dispersion relations and estimated oscillator strengths of the resonance lines. *Journal of Quantitative Spectroscopy and Radiative Transfer*, 25(5):395 – 402, 1981.
- [4] Flavio Cavanna. Private Communication.
- [5] P Cennini, J.P Revol, C Rubbia, F Sergiampietri, A Bueno, M Campanelli, P Goudsmit, A Rubbia, L Periale, S Suzuki, C Chen, Y Chen, K He, X Huang, Z Li, F Lu, J Ma, G Xu, Z Xu,

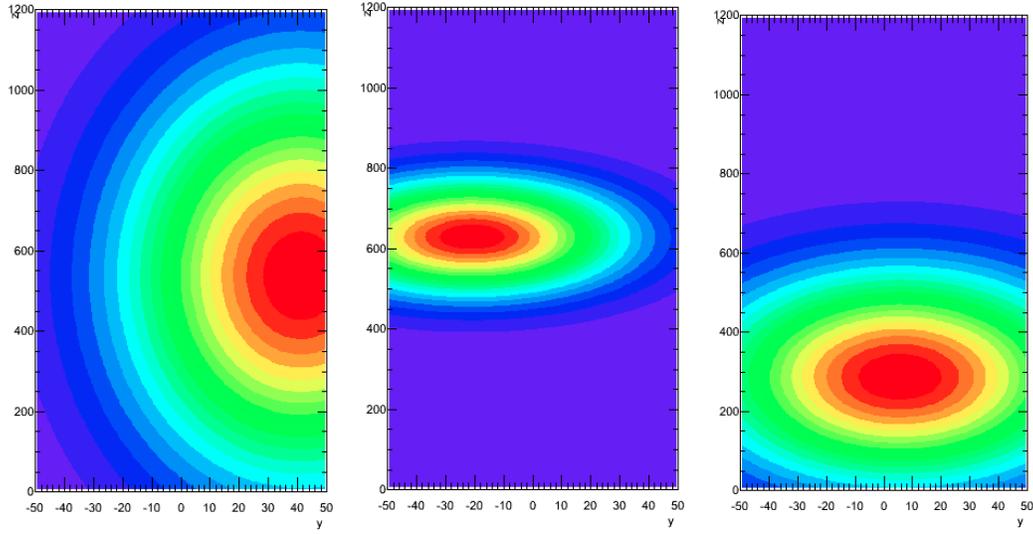


Figure 19: The YZ positions of flashes corresponding to three different subevents in the same overlaid cosmic and neutrino event

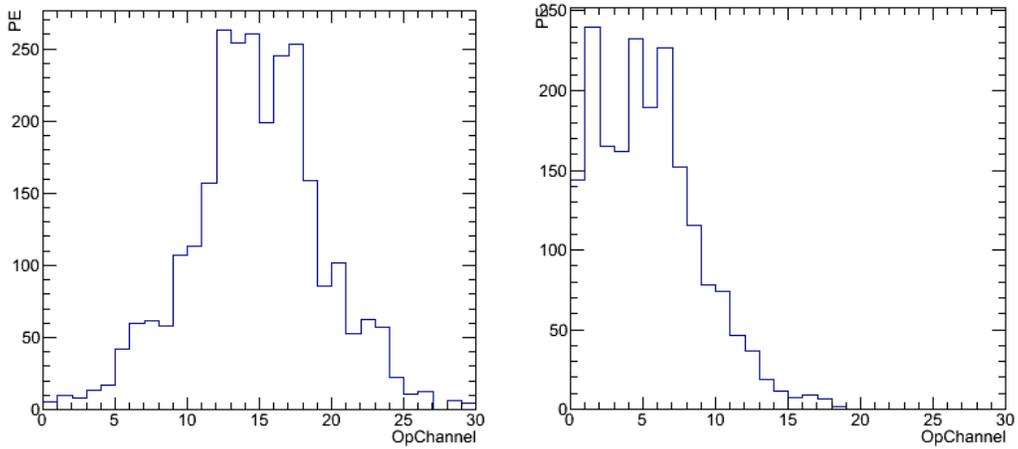


Figure 20: Number of photoelectrons counted in each optical channel for two flashes

- C Zhang, Q Zhang, S Zheng, F Cavanna, D Mazza, G.Piano Mortari, S Petrera, C Rossi, G Manocchi, P Picchi, F Arneodo, I.De Mitri, O Palamara, D Cavalli, A Ferrari, P Sala, A.Borio di Tigliole, A Cesana, M Terrani, C Zavattari, S Baibussinov, A Bettini, C Carpanese, S Centro, D Favaretto, D Pascoli, A Pepato, F Pietropaolo, S Ventura, P Benetti, E Calligarich, S Campo, S Coco, R Dolfini, B Ghedi, A.Gigli Berzolari, F Mauri, L Mazzone, C Montanari, A Piazzoli, A Rappoldi, G.L Raselli, D Rebuzzi, M Rossella, D Scannicchio, P Torre, C Vignoli, D Cline, S Otwinowski, H Wang, and J Woo. Detection of scintillation light in coincidence with ionizing tracks in a liquid argon time projection chamber. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 432(23):240–248, 1999.
- [6] Tadayoshi Doke, Akira Hitachi, Jun Kikuchi, Kimiaki Masuda, Hiroyuki Okada, and Eido Shibamura. Absolute scintillation yields in liquid argon and xenon for various particles. *Japanese Journal of Applied Physics*, 41(Part 1, No. 3A):1538–1545, 2002.
- [7] E. Morikawa et al. Argon, krypton, and xenon excimer luminescence: From the dilute gas to the condensed phase. *J. Chem. Phys.*, 91:1469, 1989.
- [8] A. C. Sinnock and B. L. Smith. Refractive indices of the condensed inert gases. *Phys. Rev.*, 181:1297–1307, May 1969.
- [9] R. K. Teague and C. J. Pings. Refractive index and the lorentz–lorenz function for gaseous and liquid argon, including a study of the coexistence curve near the critical state. *The Journal of Chemical Physics*, 48(11):4973–4984, 1968.
- [10] Craig Thorn. Catalogue of liquid argon properties. MicroBooNE docdb 412-v4, Oct 09.