

# Changing LArSoft Framework to CD ART

Brian Rebel  
September 2010

# Why Change?

- The major reason to change is that we get on the same page with future intensity frontier experiments - makes it easier for postdocs, students to transition from one to another
- Also gain support from Computing Division in upkeep of framework, how it plays with new releases from ROOT etc
- The new framework has several other desirable features
  - Designed for multithread running, should we ever need it
  - Better type safety and memory management
  - Desirable features such as configurable services (more later)
  - Ability to navigate the output ROOT file and draw distributions from data members
- Information on using the new framework is available at <https://cdcvs.fnal.gov/redmine/projects/larsoftsvn/wiki/>

# Basic Concepts

The namespace `edm` is short for "Event Data Model". It is the namespace that contains the handles to both information stored in an event and configurations for jobs. Objects that are stored in an event are collectively known as data products. They can either be added to an event using an `edm::EDProducer` derived module or they can be retrieved and operated on using an `edm::EDAnalyzer` module. Once an object has been stored in the event, its data cannot be altered.

## `edm::EDProducer`

---

This is a type of module that makes data products and stores them in the `edm::Event`. The module takes an `edm::ParameterSet` in the constructor and uses that to configure that module. The user must supply the implementation for the `edm::EDProducer::produce()` method to create and store data products.

## `edm::EDAnalyzer`

---

This is a type of module that analyzes data products but cannot write them in an `edm::Event`. The module takes an `edm::ParameterSet` in the constructor and uses that to configure that module. The user must supply the implementation for the `edm::EDProducer::analyze()` method to analyze data products.

## `edm::Event`

---

The `edm::Event` is the primary way to access products made by `EDProducer` type modules. The accessing is done by creating a

It also provide the user with information about an event such as the run, event number, etc through methods like

`edm::Event::run()`

The `edm::Event` can also be used to access products by asking it to return an `edm::Handle`.

# Basic Concepts

## edm::Handle

---

An edm::Handle is what is returned to a Module when a data product is requested. The request can either be from a edm::EDProducer that is attempting to get objects stored in a previous reconstruction or analysis step, or it can be from a edm::EDAnalyzer that is attempting to do some analysis task using the information in the object. For example, to get the data product mp::MyProd from the event, one should do

```
1 edm::Handle< std::vector<mp::MyProd> > mplistHandle;  
2 evt.getByLabel("moduleprocesslabel",mplistHandle);
```

where evt is an object of type edm::Event discussed below. The edm::Event::getByLabel method takes two arguments, the first is the name of the process associated with the module that produced the list of mp::MyProd objects, and the second is the edm::Handle that is to be associated to the list.

edm::Handles look like a pointer in the code in that the data members of the object being handled are accessed using the "->". For example, to get the size of the list one can do

```
1 mplistHandle->size();
```

## edm::ParameterSet

---

This object keeps track of which parameters are to be set by the user at run time for a module or edm::Service. It can interpret several data types including

- bool
- int
- unsigned int
- std::vector<int>
- std::vector<unsigned int>
- std::string
- std::vector<std::string>
- double
- std::vector<double>

Other types are available, but the above list should serve almost all a user's needs.

# Basic Concepts

## edm::Service

---

The `edm::Service` is a templated object that behaves like a singleton, except that it is owned and managed by the framework. A service can be used within any module. Services can be configured using the job configuration file. A typical example of the use of a service is the detector `Geometry`. The `Geometry` is needed in just about every module, but you don't want to keep making instances of the `Geometry`. Additionally, the different detectors may have to set different parameter values that should be handled in the job configuration.

## edm::TFileService

---

This is a specialized service that connects up to the file where histograms made by modules are to be stored. It provides a mechanism for making `TObjects` to be stored in that file and managing the memory for those objects.

## MessageLogger and MessageService

---

The `MessageLogger` provides several levels of messages that can be used to print information to an output log. The levels most likely to be useful are `edm::LogDebug`, `edm::LogInfo`, `edm::LogWarning` and `edm::LogError`. These are listed in order of severity. The `MessageService` can be configured to set the number of messages printed and to send each class of message to a different output stream.

It can be used as follows:

```
1 if(x > 2) edm::LogWarning("XTooBig") << "x = " << x << " is too big";
```

## edm::Exception

---

The `edm::Exception` can be used to cause the framework to end a job gracefully if some predetermined bad thing happens. The use of the `edm::Exception` can be configured to skip a module, or skip to the next event, run, etc. Different exception classes can be set to do different things.

`edm::Exception` can be used as follows:

```
1 if(x > 2) throw edm::Exception("XTooBig") << "x = " << x << " is too big";
```

# Basic Concepts

## `edm::Ptr<T>` and `edm::PtrVector<T>`

---

The `edm::Ptr<T>` is a template class that acts like a ROOT TRef. It provides a linkage between objects written into different areas of the event (and output file). For example, the `recob::Wire` object holds an `edm::Ptr<raw::RawDigit>` pointing to the `raw::RawDigit` it corresponds too. The `edm::Ptr<T>` behaves like a pointer (ie you access the methods of the T using the "->"). It also has functionality to return the actual pointer to the object in question using `edm::Ptr<T>::get()` or to check if the `edm::Ptr<T>` is pointing to a legitimate object using `edm::Ptr<T>::isNull()`.

An `edm::PtrVector<T>` is a vector of `edm::Ptr<T>` objects. It provides the basic functionality you would expect from a `std::vector`, including iterators, `size()`, `begin()` and `end()` methods. It is useful when storing the connections of many objects of the same type in an object, for example `recob::Hits` in a `recob::Cluster`.

# Status of the Switch

- The following packages have been switched over:

SimulationBase	Simulation	Geometry
EventGeneratorBase	EventGenerator	Utilities
RawData	RecoBase	LArG4
DriftElectrons	DetSim	

- These are the packages necessary to test the simulation chain
- Have successfully run jobs to produce events from GENIE, CRY and Single Particle simulations
- Comparisons of output for several modules in next slides

# Making the Switch

## Converting Data Objects from FMWK

---

The main difference between FMWK and the CD framework is that the CD framework does not save TObjects so that data objects do not inherit from TObject. Thus, the main difference between the FMWK objects and the CD framework objects is that in the latter framework the definition and implementation files do not have the following:

in a .h file remove

```
#include "TObject.h"
```

```
: public TObject
```

```
ClassDef(ObjName,1)
```

in a .cxx file remove

```
ClassImp(ObjName)
```

- Notice that the majority of data objects have been converted in RecoBase, RawData, SimulationBase, Simulation and LArG4
- Have to make 2 additional files to register objects written to file, details on the wiki

# Making the Switch

## Converting a Module from FMWK

---

It is very straight forward to convert a module from the FMWK framework into the CD developed framework.

There are only a few easy steps to doing the conversion, as illustrated using the DriftElectrons module.

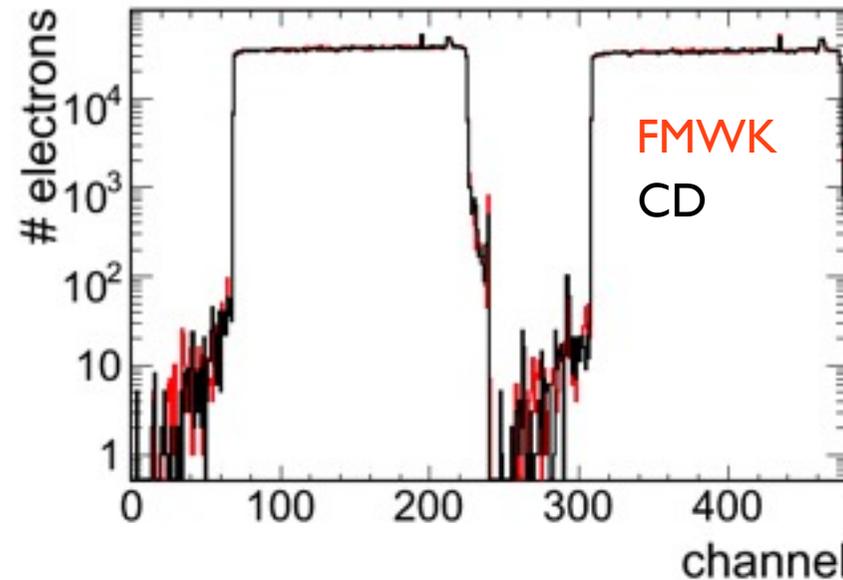
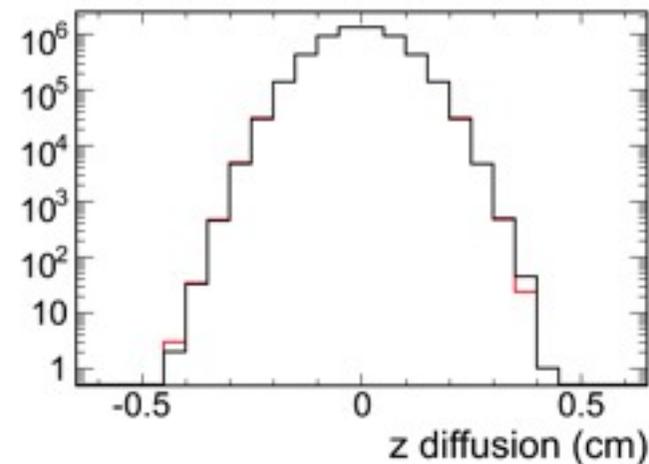
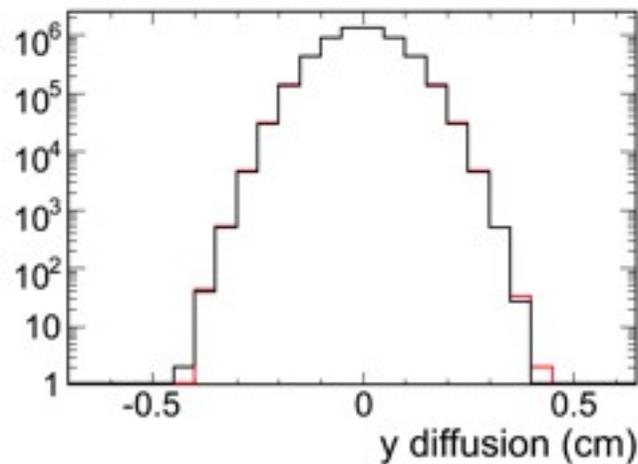
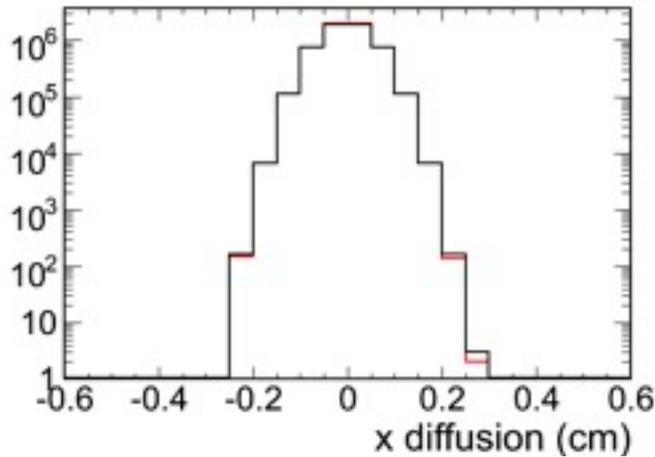
The .h file must be changed so that the module inherits from the proper type of edm module and the Update method must be removed.

In the .cxx file the line calling the macro to declare a module is removed and of course the proper header files must be included. The Update method must be removed and the parameters are set using the edm::ParameterSet passed into the constructor. The produces() method of the base edm::EDProducer must be called with the type of object(s) to be stored in the event. The last major change is how objects from previous modules are gotten out of the edm::Event, as might be expected.

A new file, the \_plugin.cc file is needed to call the macro declaring the module to the framework.

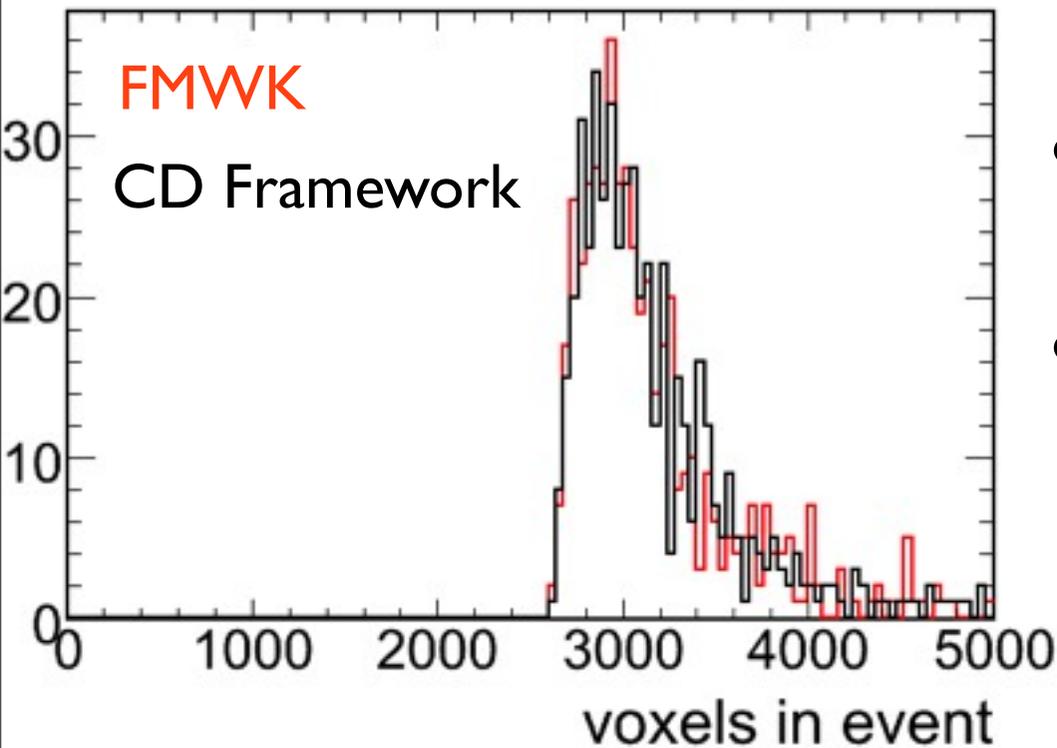
- An example is on the wiki

# Comparing Electron Drift

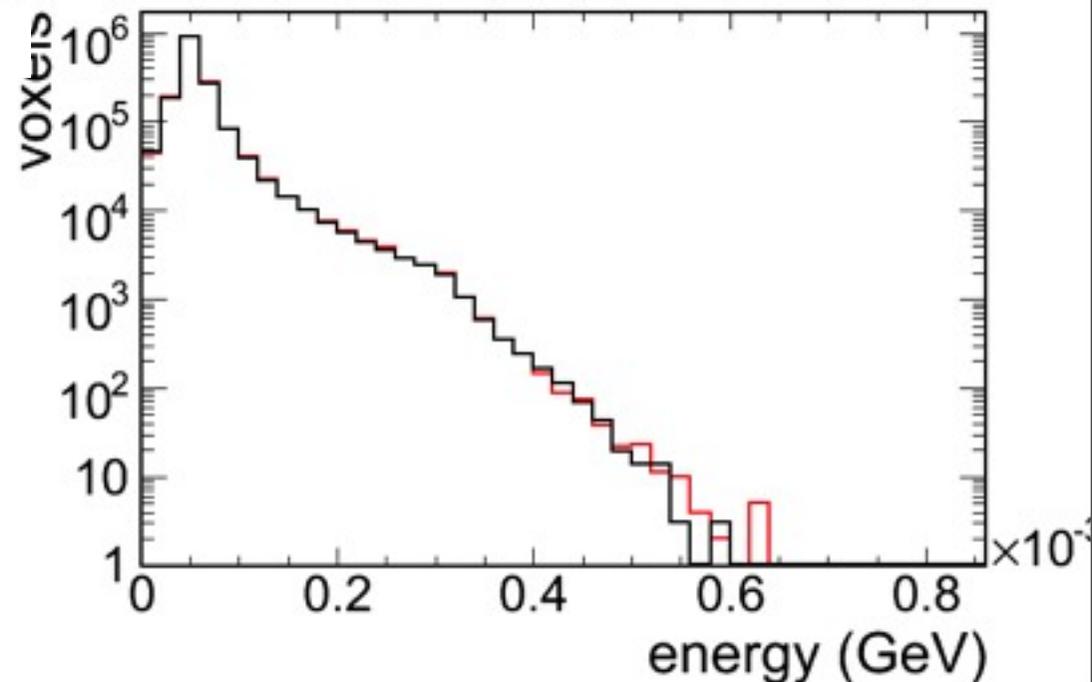


- Produced 500 single 6 GeV muons with each version of the software
- Made several sanity check plots to ensure that the two versions of the software were doing the same thing
- Plots to left show diffusion in each direction
- Plot above shows number of electrons created for each channel

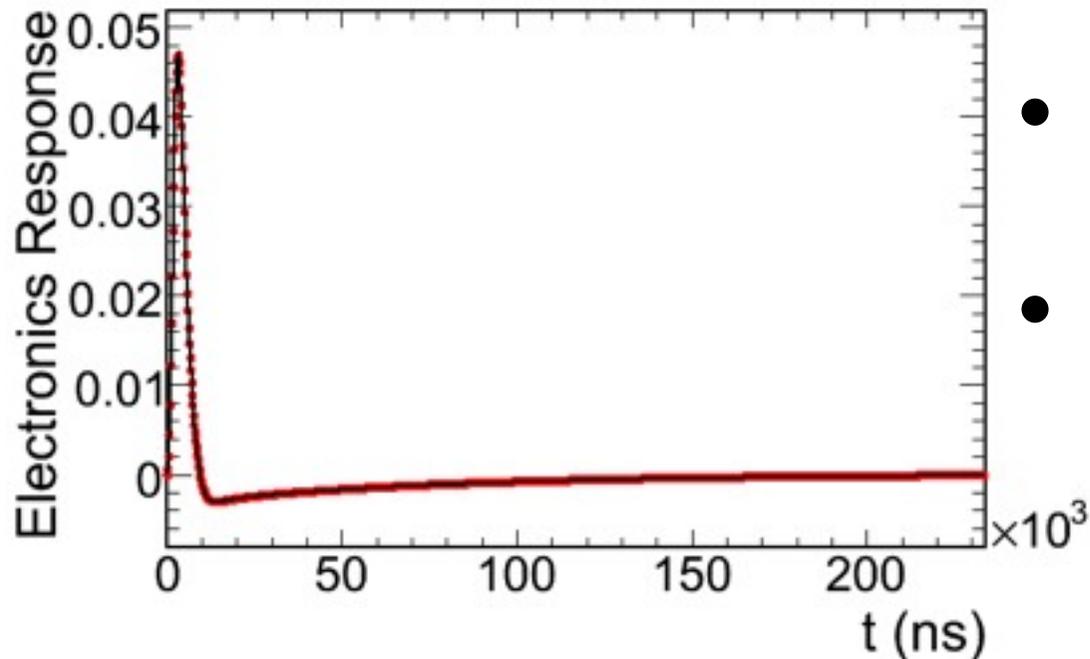
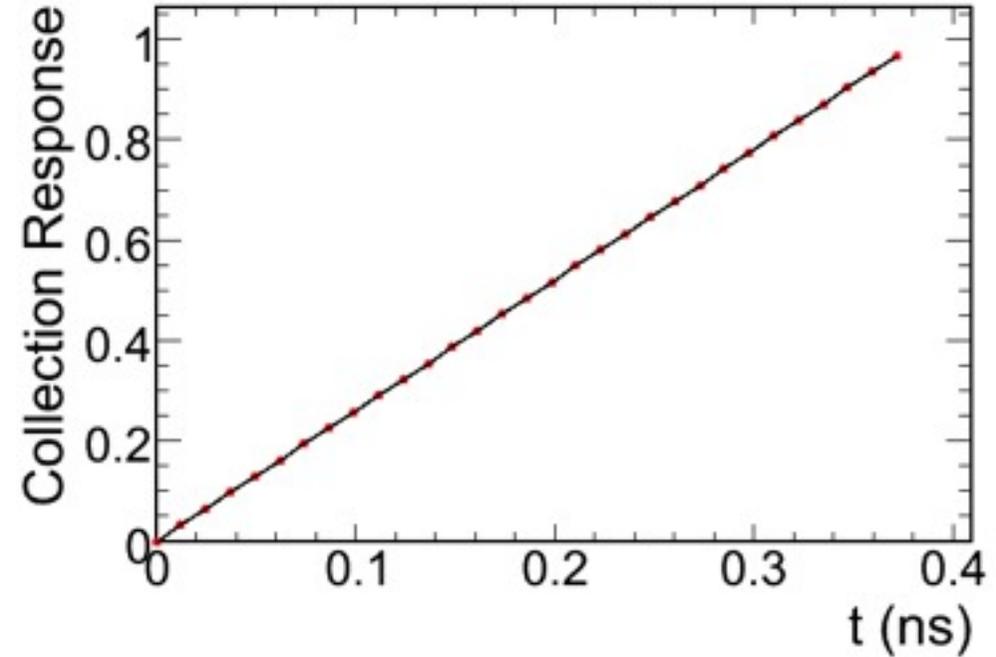
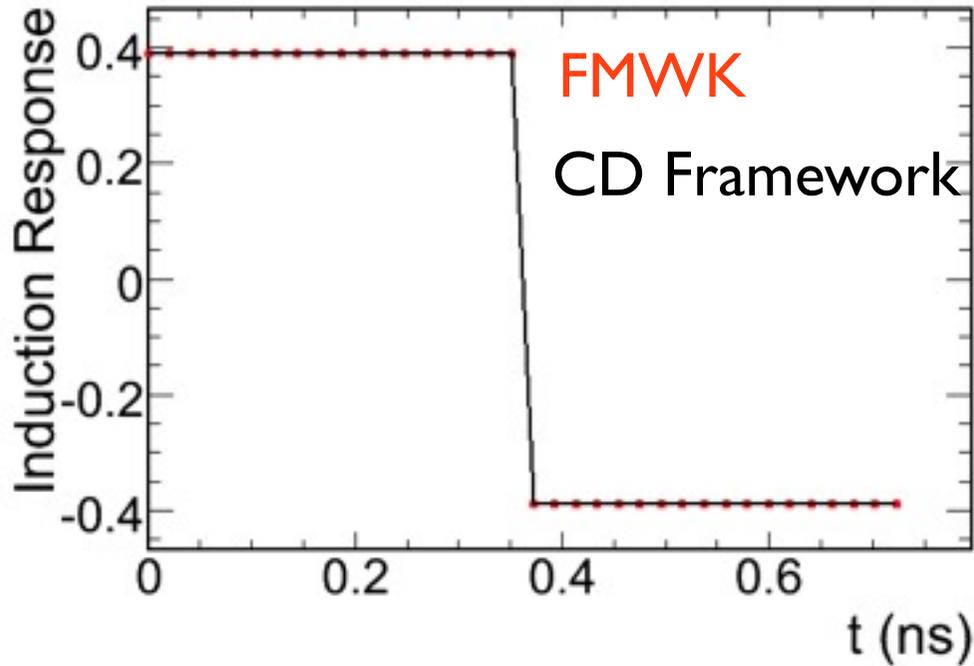
# Comparing Output of LArG4



- Similar number of voxels created by each version
- Same distribution of energy/voxel

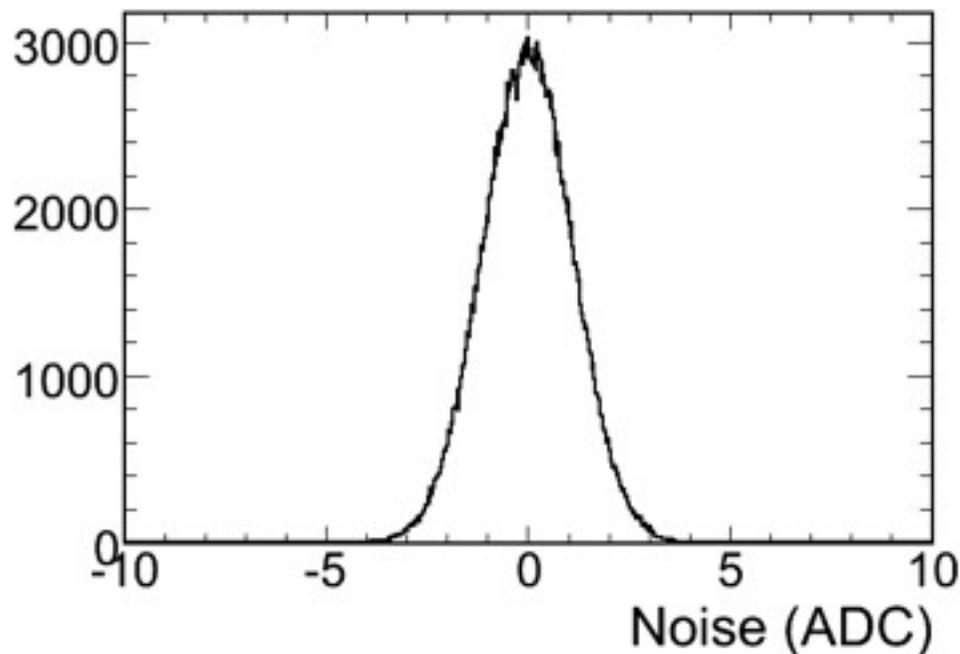
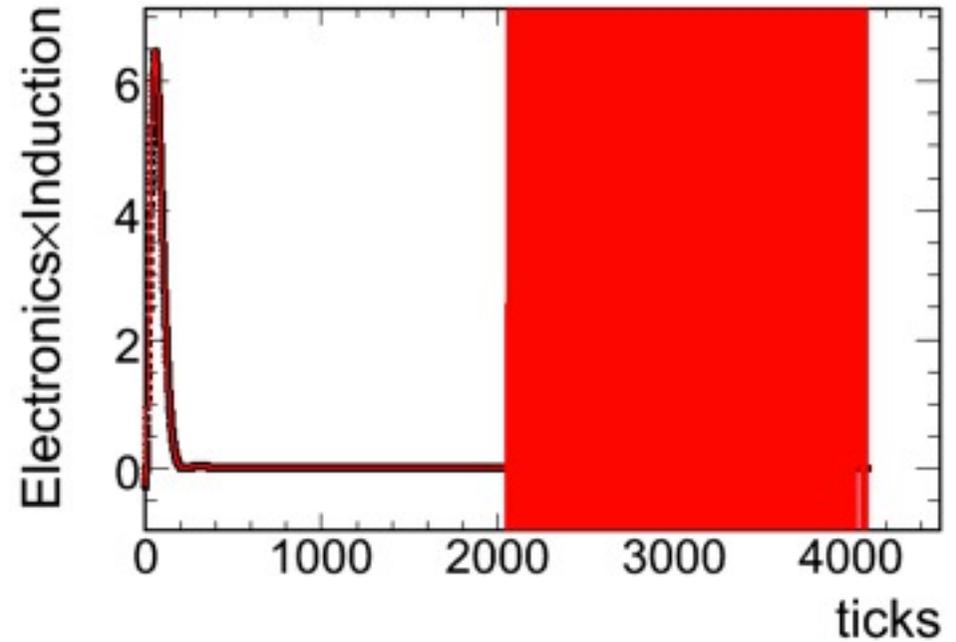
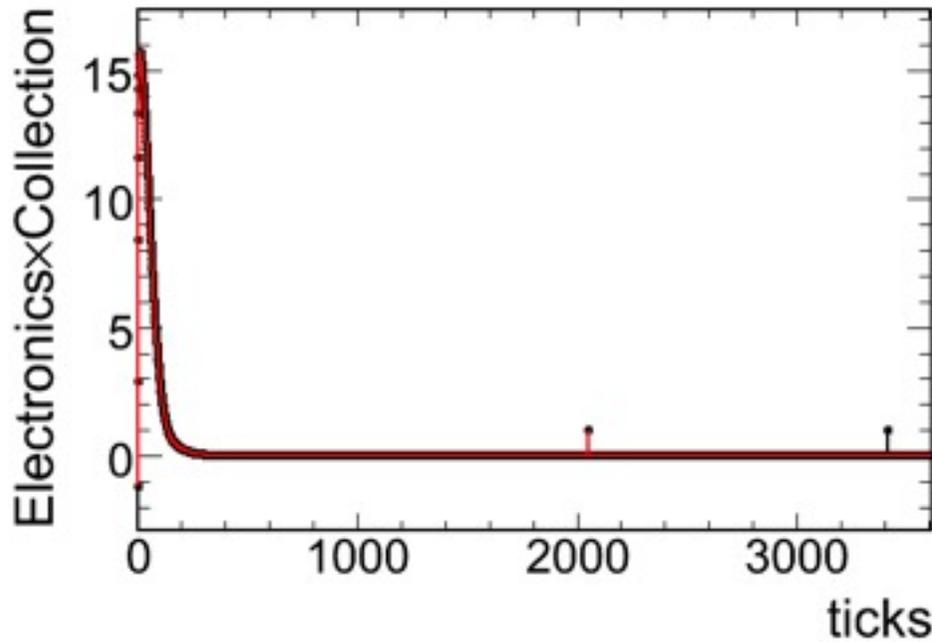


# DetSim Comparison



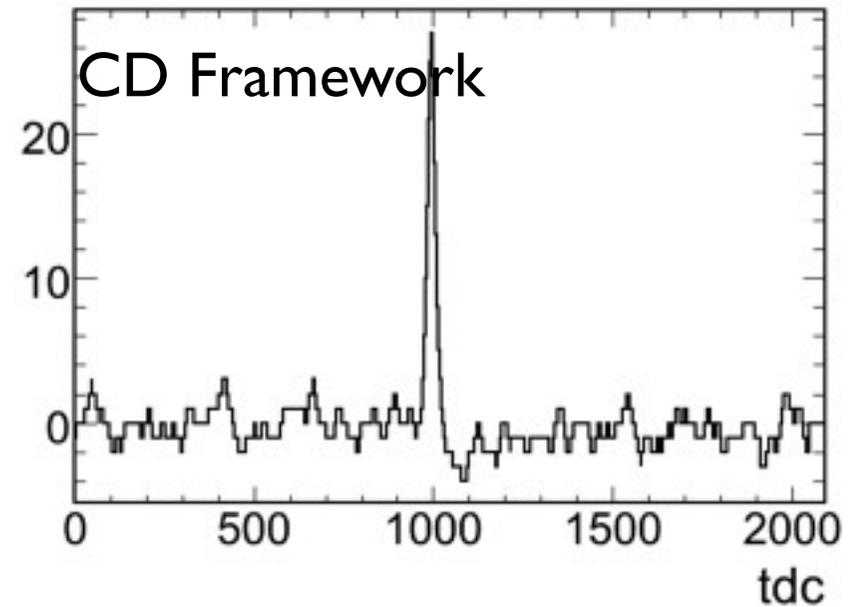
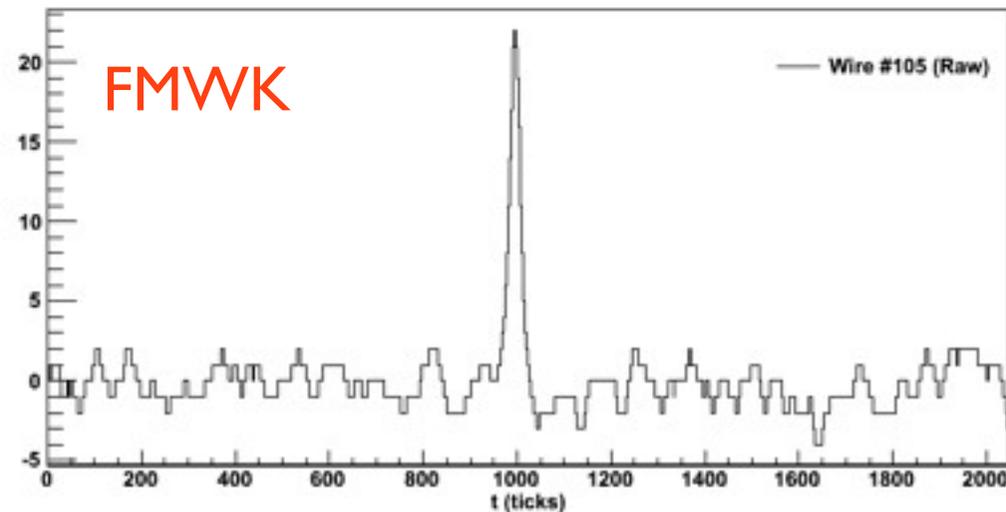
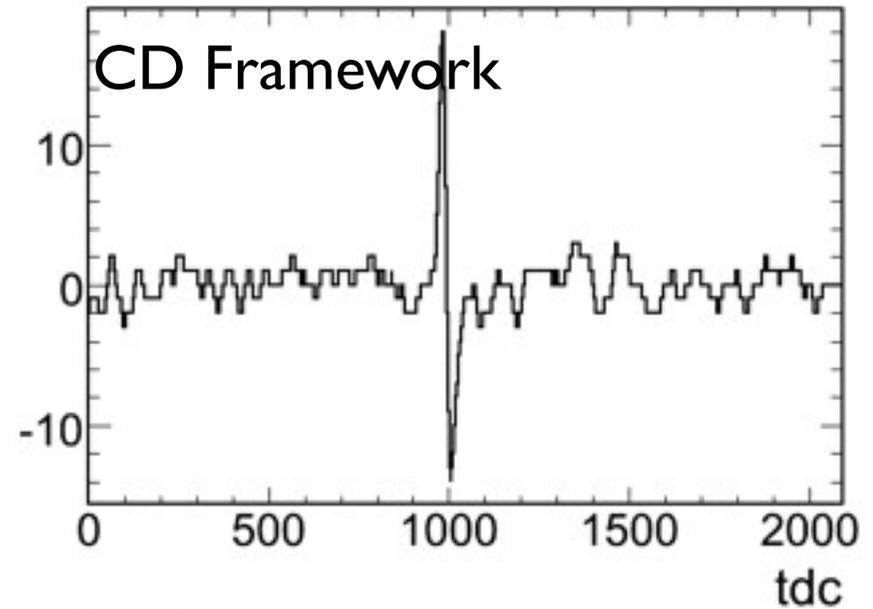
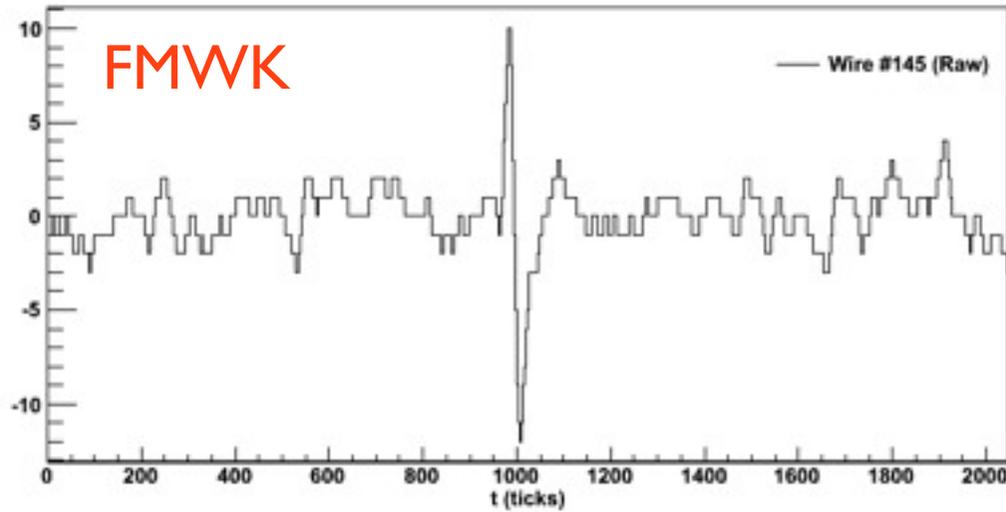
- The response files for the different planes and electronics are identical
- Not surprising as they come from the same ROOT file

# DetSim Comparison



- FFT Convolution of the electronics with the different plane response files is also identical, at least for the ticks that actually matter and are not just padding of a vector
- The noise distributions are also the same

# DetSim Comparison



- Signal shapes are the same for FMWK and CD framework produced tracks

# Next Steps

- New svn repository has been started with the code based on the new framework
- The new code base will be installed in blue arc areas used for submission to the FermiGrid (one area for each LAr experiment)
- Convert EventDisplayBase and EventDisplay to work in the new framework (likely me)
- Add functionality to update the configuration interactively, as you can do in the event display now (CD person)
- Setup the build system to acknowledge private and public releases (me and CD person)
- Get some early adopters to volunteer to convert their packages now (prefer those whose packages are early in the reco chain)
- Set a deadline for turning off the CVS
- Schedule a workshop for teaching average user how to use the new framework - propose Oct 7, 2010.