

UNIVERSITY OF CAMBRIDGE

µBooNE

SHORT-BASELINE NEAR DETECTOR SBND

DUNE

...and more!

LArSoft Simulation Tutorial

Magnus Handley

University of Cambridge

11th UK LArSoft Workshop: Edinburgh 2026

What are we learning?

*This tutorial is based on the work of **many** previous people: thank you to (at least) Joe Bateman, Anyssa Navrer-Agasson, Rob Derby, Luis Mora Lepin, Marina Reggiani-Guzzo and Aran Borkum!*

1) What a FHiCL file is and basic syntax
2) How to write a basic FHICL to simulate your first particles
3) How to use the **lar** command to run your simulation
4) How to run Geant4 using LArSoft to simulate particle propagation
5) How to run the detector simulation using LArSoft

By the end of this workshop, you will have simulated 10 1μ1p 'neutrino-like' events, each with an additional muon to mimic a cosmic ray. You will use this sample in subsequent tutorials looking at reconstruction and analysis!

What are we learning?

*This tutorial is based on the work of **many** previous people: thank you to (at least) Joe Bateman, Anyssa Navrer-Agasson, Rob Derby, Luis Mora Lepin, Marina Reggiani-Guzzo and Aran Borkum!*

1) What a FHiCL file is and basic syntax
2) How to write a basic FHICL to simulate your first particles
3) How to use the **lar** command to run your simulation
4) How to run Geant4 using LArSoft to simulate particle propagation
5) How to run the detector simulation using LArSoft

By the end of this workshop, you will have simulated 10 1μ1p 'neutrino-like' events, each with an additional muon to mimic a cosmic ray. You will use this sample in subsequent tutorials looking at reconstruction and analysis!

There's a lot we're covering as we go along, so just remember…


DON'T
PANIC

What will you need?

You need a working version of **sbndcode** with the **uk_larsoft_workshop_2026** branch checked out, set up, and compiled: see Lucy's tutorial from this morning!



Starting in a fresh terminal:

- Start a container:

```
source /Workshop/setting_up/setup_container.sh
```

Setup your environment – either:

- Using the setup script we made earlier:

```
source setup.sh
```

- Or if you didn't make one earlier (run from your larsoft dev area):

```
source /cvmfs/sbnd.opensciencegrid.org/products/sbnd/setup_sbnd.sh
source localProducts_*/setup
mrbsetenv
mrbslp
```

Fickle FHiCLs: What, Why, and How?



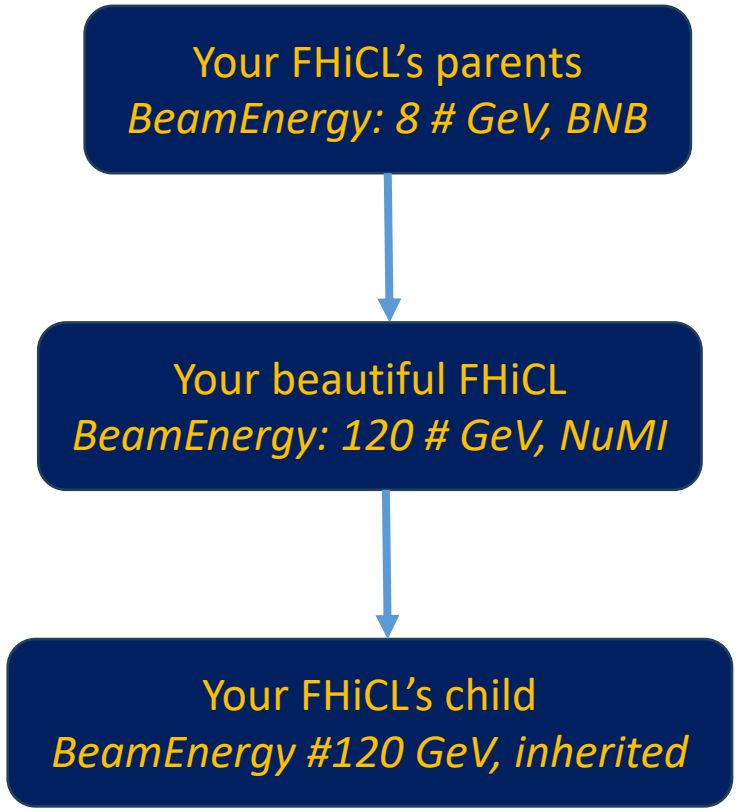
What is a FHiCL?

- FHiCL (pronounced “fickle”) stands for *Fermilab Hierarchical Configuration Language*
- These are files which tell LArSoft (and other software built on **art**) what to do
  - Hierarchical refers to how FHiCL files inherit variables from “parent” FHiCLs
  - Configuration describes how FHiCLs can modify the variables of your simulation tools without needing to recompile
- Language – FHiCLs have their own specific syntax, taking elements from JSON, C++ and Python
- They appear across the simulation chain and are identified by the .fcl extension

Why do we FHiCL?

There are plenty of other configuration file formats we could use (JSON, XML, YAML etc.) so why did Fermilab develop its own?

- FHiCLs allow us to avoid hardcoded variables in each LArSoft module you use.
- You can inherit and modify variables from parent FHiCL files, without needing to recompile
- Parameters will persist across a chain of FHiCLs and from simulation through reconstruction to analysis code.
- Common parameters can be stored in one place and reused easily by many different daughter FHiCLs without needing to redefine.



How do we FHiCL? (Syntax 1)

Parameters:

- You can define integers, floats and strings as parameters
- Both C++ (//) and pythonic (#) style comments can be used to make your variables clearer.

```
pi: 3.14159 // This is a C++-style comment
beam: "BNB" # This is a python-style comment
output_name: "hello_world.root"
```

Sequences:

- Sequences can take the same and mixed types of input, including other sequences!
- You can also override individual elements of a sequence.

```
Seq1: [1, 2, 3] // Sequences can be single type
Seq2: [1, [2.3, “Gaussian”]] // Or mixed type
Seq1[2]: 3 // And elements can be overridden
```

How do we FHiCL? (Syntax 2)

Tables:

- We can define a table using curly braces and can assign entries to a mix of types.
- Entries can be overwritten similarly to a sequence.

Splicing tables:

- Sometimes we want to include the parameters of one table into another – this is done with *splicing*.
- Tables can be spliced together using @table::name, allowing all the entries of the table called name to be added at that point.

```
// We can declare a table using curly brackets
tab1:{
     pi: 3.14159
     beam: "BNB"
     seq1: [1, [2.3, "Gaussian"], 4]
}
// Entries can be overwritten by name
tab1.beam: "NuMI"
// Tables can also be spliced together
tab2: {
     @table::tab1
     n: 100
}
beam_name: tab2.beam // Inherited from tab1
```

How do we FHiCL? (Syntax 3)

Defining a prolog

```
BEGIN_PROLOG
bnb: 8 // 8 GeV beam
numi: 120 // 120 GeV beam
END_PROLOG
BeamEnergy: @local::numi
```

Prologs:

- We can define sets of parameters we might want to reference later, or in another file.
- If we want to access the parameters saved in a prolog in a different file, we can access them using #include: e.g. if we want the parameters saved in **beam_config.fcl**, we can get them using **#include beam_config.fcl**
- If we want a parameter defined elsewhere in the same file, or included through a prolog, we can access it using @local::<var>.

Reminder: if any parameter gets defined multiple times, the later definition always overrides the earlier ones!

beam_config.fcl

```
BEGIN_PROLOG
bnb: 8 // 8 GeV beam
numi: 120 // 120 GeV beam
END_PROLOG
BeamEnergy: @local::numi
```

your_wonderful_fhicl.fcl

```
#include "beam_config.fcl"
BeamEnergy: @local::numi
```

Creating Your First FHiCL



Setting Up Your FHiCL

Now that we know a bit about how they work, it's time to make one of our own!

- Make a new directory in your home directory to put your FHiCL in and give it a sensible name (e.g. simulation_tutorial)

- Using your favourite text editor, create and open a new file called **sim_tutorial_gennon0_T0.fcl**

```
cd $HOME
mkdir simulation_tutorial
cd simulation_tutorial
emacs –nw sim_tutorial_gennon0_T0.fcl
```

Basic Structure

FHiCL files that run things (as opposed to ones that only contain config parameters) all have the same basic structure:

- Include: Import other FHiCLs you need
- Process name: Name this set of modules.
- Services: Define the simulation-specific services required.
- Source: Define your FHiCL inputs.
- Physics: Declare and configure the modules that will be run.
- Outputs: Specify outputs of the FHiCL.

```
#include

process_name:

services: {

}

source: {

}

physics:{

}
outputs:{

}
```

Basic Structure: Includes

Like we saw earlier, these allow you to inherit definitions from other files

For today let's add the following as the include block:

```
// experiment specific configurations
#include "simulationservices_sbnd.fcl"
#include "messages_sbnd.fcl"
// configuration files containing prologs
#include "singles_sbnd.fcl"
#include "rootoutput_sbnd.fcl"
```

So overall, our file now looks like the one on the right:

```
// experiment specific configurations
#include "simulationservices_sbnd.fcl"
#include "messages_sbnd.fcl"
// configuration files containing prologs
#include "singles_sbnd.fcl"
#include "rootoutput_sbnd.fcl"

process_name:

services: {

}

source: {

}

physics:{

}
outputs:{

}
```

Basic Structure: Process Name

Process name:

- This defines the name for the collection of modules this FHiCL will run.
- This name should be unique, as the same process cannot be run multiple times over the same art-root file.
- The aim of this tutorial is to generate single particles, so you should call this process **SingleGen**

```
// experiment specific configurations
#include "simulationservices_sbnd.fcl"
#include "messages_sbnd.fcl"
// configuration files containing prologs
#include "singles_sbnd.fcl"
#include "rootoutput_sbnd.fcl"

process_name: SingleGen

services: {

}

source: {

}

physics:{

}
outputs:{

}
```

Basic Structure: Services

Services:

- The services table defines the simulation-specific services that are generally needed.
- This could include detector geometry, physical properties or file management.

For this tutorial, we need to add the SBND-specific simulation service, and a root output:

```
...

process_name: SingleGen

services:{
     @table::sbnd_simulation_services
     TFileService:{
          fileName:"hist_prod_single_sbnd.root
     }
}

source: {

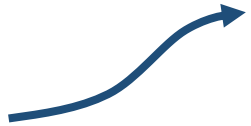
}

...
```

Basic Structure: Source

Source:

- This table contains all information regarding the input data this FHiCL will take
- In our case, we are at the start of the simulation so we add **module_type: EmptyEvent** so LArSoft doesn't expect to receive any already created data



```
...

source:{
     module_type: EmptyEvent
     timestampPlugin:{
          plugin_type: "GeneratedEventTimestamp"
     }



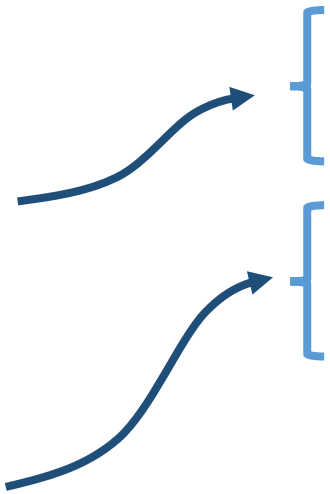

}

...
```

Basic Structure: Source

Source:

- This table contains all information regarding the input data this FHiCL will take
- In our case, we are at the start of the simulation so we add **module_type: EmptyEvent** so LArSoft doesn't expect to receive any already created data
- maxEvents: 10 sets the default number of events. A value of -1 would mean all events from an input file, and this can also be overwritten at runtime
- **firstRun** and **firstEvent** set the default start values for run and event



```
...

source:{
     module_type: EmptyEvent
     timestampPlugin:{
          plugin_type: "GeneratedEventTimestamp"
     }
     maxEvents: 10
     firstRun: 1
     firstEvent: 1

}

...
```

Basic Structure: Physics

Physics:

- This table declares and configures the modules that will be run over the input.
- These are split into producers, analyzers, and filters.

Producers add information to a root file without needing an input. Our single-particle event generator (which we declare in the **generator** field) is a producer.

These modify the input file (in our case an empty empty event, as defined in **source**)

```
physics:{
     producers:{
          rns: { module_type:
                    "RandomNumberSaver"
          }
          generator: @local::sbnd_singlep
     }
}
```

Basic Structure: Physics

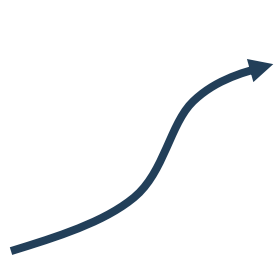
Physics:

- This table declares and configures the modules that will be run over the input.
- These are split into producers, analyzers, and filters.

Analyzers perform analysis based on an input file without modifying it.

Filters remove files we aren't interested in: these modify the input file.

Our workflow is producer-only, so we leave analyzers and filters empty.



```
physics:{
     producers:{
          rns: { module_type:
                    "RandomNumberSaver"
          }
          generator: @local::sbnd_singlep
     }
     analyzers: { }
     filters: { }



}
```

Basic Structure: Physics

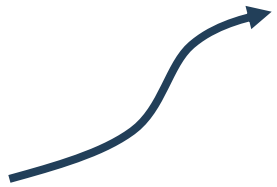
Physics:

- This table declares and configures the modules that will be run over the input.
- These are split into producers, analyzers, and filters.

We then need to configure the modules

simulate declares the order to run producers

stream1 defines the art-root output stream of the FHiCL



```
physics:{
     producers:{
          rns: { module_type:
                    "RandomNumberSaver"
          }
          generator: @local::sbnd_singlep
     }
     analyzers: { }
     filters: { }
     simulate: [rns, generator]
     stream1: [out1]


}
```

Basic Structure: Physics

Physics:

- This table declares and configures the modules that will be run over the input.
- These are split into producers, analyzers, and filters.

More configuration:

- **trigger_paths** lists everything which modifies an event (producers and filters)
- **end_paths** lists everything which doesn't modify an event (analyzers and outputs)



```
physics:{
     producers:{
          rns: { module_type:
                    "RandomNumberSaver"
          }
          generator: @local::sbnd_singlep
     }
     analyzers: { }
     filters: { }
     simulate: [rns, generator]
     stream1: [out1]
     trigger_paths: [simulate]
     end_paths: [stream1]
}
```

Basic Structure: Outputs

Outputs:

- This table declares where the output of the FHiCL should go
- Notice that the output has the same name (out1) as was defined in stream1 earlier
- The lines in out1 inherit the rootoutput table and define the name of the output file (but this can be overwritten at execution)

```
...

outputs:{
     out1:{
          @table::sbnd_rootoutput
          fileName:
          "prodsingle_sbnd_%p-%tc.root"
     }
}
```

Putting Everything Together: Your First FHiCL!

Your completed FHiCL should look like this!

```
// experiment specific configurations
#include "simulationservices_sbnd.fcl"
#include "messages_sbnd.fcl"
// configuration files containing prologs
#include "singles_sbnd.fcl"
#include "rootoutput_sbnd.fcl"


process_name: SingleGen


services:{
        @table::sbnd_simulation_services
        TFileService:{
                fileName:"hist_prod_single_sbnd.root"
        }
}


source:{
        module_type: EmptyEvent
        timestampPlugin:{
                plugin_type: "GeneratedEventTimestamp"
        }
        maxEvents: 10
        firstRun: 1
        firstEvent: 1
}
```

```
physics:{
        producers:{
                rns: { module_type:
                "RandomNumberSaver"
                }
                generator: @local::sbnd_singlep
        }
        analyzers: { }
        filters: { }
        simulate: [rns, generator]
        stream1: [out1]
        trigger_paths: [simulate]
        end_paths: [stream1]
}


outputs:{
        out1:{
                @table::sbnd_rootoutput
                fileName:
                "prodsingle_sbnd_%p-%tc.root"
        }
}
```

A copy of this file can be found here: $MRB_SOURCE/sbndcode/sbndcode/Workshop/TPCSimulation/sim_tutorial_gennon0_T0.fcl

Task 1: Running Your First FHiCL

It’s time to run our FHiCL! If you have sbndcode setup properly, you will have access to the **lar** command. We run our FHiCL using the -c (--config) option:

```
lar -c sim_tutorial_gennon0_T0.fcl
```

If all goes well, this should run and finish with a summary message: “Art has completed and will exit with status 0 ”.

You can also override some of the default parameters defined in your FHiCL:

- -n (--nevts): specify the number of events to process.
- -o (--output): set the name of the output file.
- -s (--source): specify which file to take as an input

You can get a full list of options available using `lar --help`

Modifying Your FHiCL

But what are we *actually* simulating?

LArSoft successfully ran our FHiCL, but what is it really doing?

Let's look at what the FHiCLs we import contain:

- We need to find where they live: we can do this using the script **find_fhicl.sh** which should be available if you have larsoft set up, but can also be found at $MRB_SOURCE/sbndcode/sbndcode/Workshop/TPCSimulation/find_fcl.sh

- `source find_fhicl.sh singles_sbnd.fcl`

This will return the path to the FHiCL in your LArSoft install directory, which you can open using your preferred text editor.

Take a look inside and see if you can spot which particle is being simulated (you may need to identify another FHiCL to investigate!).

singles_sbnd.fcl

Opening the file, you'll see a prolog like this:

In the FHiCL we can see some of the parameters that we can modify, such as:

- Initial momentum: P0
- Initial Time: T0
- Initial Position: X0, Y0, Z0

But this still doesn't tell us which particle is being simulated! We need to go further up the FHiCL chain…

```
#include "singles.fcl"


BEGIN_PROLOG
sbnd_singlep: @local::standard_singlep
# Particle generated at this time will appear in main drift
window at trigger T0.
physics.producers.generator.T0: [ 1.7e3 ] # us
physics.producers.generator.P0: [ -1.0 ] # GeV/c
physics.producers.generator.SigmaP: [ 0.0 ] # GeV/c
physics.producers.generator.PDist: 0
physics.producers.generator.X0: [ 150.0 ] # cm
physics.producers.generator.Y0: [ 150.0 ] # cm
physics.producers.generator.Z0: [ -50.0 ] # cm
physics.producers.generator.Theta0XZ: [ 15.0 ] # degrees
physics.producers.generator.Theta0YZ: [ -15.0 ] # degrees
physics.producers.generator.SigmaThetaXZ: [ 0.0 ] # degrees
physics.producers.generator.SigmaThetaYZ: [ 0.0 ] # degrees
END_PROLOG
```

singles.fcl

Find our FHiCL using:

```
source find_fhicl.sh singles.fcl
```

Take a look at what it contains! What we're looking for is the Monte Carlo PDG (Particle Data Group) code of the particle being simulated

We can find the name of the particle the code corresponds to using the list outlined here:

- https://pdg.lbl.gov/2024/reviews/rpp2024-rev-monte-carlo-numbering.pdf

singles.fcl

```
BEGIN_PROLOG
#no experiment specific configurations because SingleGen is
detector agnostic
standard_singlep:
{
module_type: "SingleGen"
ParticleSelectionMode: "all" # 0 = use full list, 1 = randomly
select a single listed particle
PadOutVectors: false
# false: require all vectors to be same length
# true: pad out if a vector is size one
PDG: [ 13 ] # list of pdg codes for particles to make
P0: [ 6. ] # central value of momentum for each particle
SigmaP: [ 0. ] # variation about the central value
PDist: "Gaussian" # 0 - uniform, 1 - gaussian distribution
X0: [ 25. ] # in cm in world coordinates, ie x = 0 is at the wire
plane
# and increases away from the wire plane
Y0: [ 0. ] # in cm in world coordinates, ie y = 0 is at the center
of the TPC
Z0: [ 20. ] # in cm in world coordinates, ie z = 0 is at the
upstream edge of the TPC and increases with the beam direction
T0: [ 0. ] # starting time
```

```
SigmaX: [ 0. ] # variation in the starting x position
SigmaY: [ 0. ] # variation in the starting y position
SigmaZ: [ 0.0 ] # variation in the starting z position
SigmaT: [ 0.0 ] # variation in the starting time
PosDist: "uniform" # 0 - uniform, 1 - gaussian
TDist: "uniform" # 0 - uniform, 1 - gaussian
Theta0XZ: [ 0. ] # angle in XZ plane (degrees)
Theta0YZ: [ 3.3 ] # angle in YZ plane (degrees)
SigmaThetaXZ: [ 0. ] #in degrees
SigmaThetaYZ: [ 0. ] #in degrees
AngleDist: "Gaussian" # 0 - uniform, 1 - gaussian
}
random_singlep: @local::standard_singlep
random_singlep.ParticleSelectionMode: "singleRandom"
#randomly select one particle from the list
argoneut_singlep: @local::standard_singlep
microboone_singlep: @local::standard_singlep
microboone_singlep.Theta0YZ: [ 0.0 ] # beam is along the z axis
microboone_singlep.X0: [125] # in cm in world coordinates, ie x = 0
is at the wire plane
microboone_singlep.Z0: [50] # in cm in world
Coordinates
END_PROLOG
```

singles.fcl

```
BEGIN_PROLOG
#no experiment specific configurations because SingleGen is
detector agnostic
standard_singlep:
{
module_type: "SingleGen"
ParticleSelectionMode: "all" # 0 = use full list, 1 = randomly
select a single listed particle
PadOutVectors: false
# false: require all vectors to be same length
# true: pad out if a vector is size one
PDG: [ 13 ] # list of pdg codes for particles to make
P0: [ 6. ] # central value of momentum for each particle
SigmaP: [ 0. ] # variation about the central value
PDist: "Gaussian" # 0 - uniform, 1 - gaussian distribution
X0: [ 25. ] # in cm in world coordinates, ie x = 0 is at the wire
plane
# and increases away from the wire plane
Y0: [ 0. ] # in cm in world coordinates, ie y = 0 is at the center
of the TPC
Z0: [ 20. ] # in cm in world coordinates, ie z = 0 is at the
upstream edge of the TPC and increases with the beam direction
T0: [ 0. ] # starting time
```

```
SigmaX: [ 0. ] # variation in the starting x position
SigmaY: [ 0. ] # variation in the starting y position
SigmaZ: [ 0.0 ] # variation in the starting z position
SigmaT: [ 0.0 ] # variation in the starting time
PosDist: "uniform" # 0 - uniform, 1 - gaussian
TDist: "uniform" # 0 - uniform, 1 - gaussian
Theta0XZ: [ 0. ] # angle in XZ plane (degrees)
Theta0YZ: [ 3.3 ] # angle in YZ plane (degrees)
SigmaThetaXZ: [ 0. ] #in degrees
SigmaThetaYZ: [ 0. ] #in degrees
AngleDist: "Gaussian" # 0 - uniform, 1 - gaussian
}
random_singlep: @local::standard_singlep
random_singlep.ParticleSelectionMode: "singleRandom"
#randomly select one particle from the list
argoneut_singlep: @local::standard_singlep
microboone_singlep: @local::standard_singlep
microboone_singlep.Theta0YZ: [ 0.0 ] # beam is along the z axis
microboone_singlep.X0: [125] # in cm in world coordinates, ie x = 0
is at the wire plane
microboone_singlep.Z0: [50] # in cm in world
Coordinates
END_PROLOG
```

singles.fcl

We can learn a few things from this file:

- **module_type: SingleGen** refers to a module in already in LArSoft, which this FHiCL calls. The module is a C++ file with functions called over the input event, and you can learn more about it here.
- We now have the full list of parameters defined in the **standard_singlep** table, including the PDG sequence. By default, singles.fcl will produce a **single 6 GeV muon** but, as you may recall, this can be overridden.

We can also see all the parameters a FHiCL has defined (including inherited parameters!) using the command:

```
fhicl-dump sim_tutorial_gennon0_T0.fcl > sim_tutorial_gen.txt
```

(the output is very long so we save the full list of parameters to a text file)

Modifying Your Output

Example 1: If we want to replace the muon with an electron, leaving everything else untouched:

sim_tutorial_gennon0_T0.fcl:

```
physics.producers.generator.PDG: [ 11 ] // overwriting the muon with an electron
```

Example 2: What if we want to produce multiple particles in a single event? We could try adding another PDG code to the sequence:

sim_tutorial_gennon0_T0.fcl:

```
physics.producers.generator.PDG: [ 11, 13 ] // electron and muon
```

Try it! Does this work the way you expect?

Modifying Your Output

…nope! Clearly something is wrong!

We're now missing entries for several parameters.

We can fix this by:

1) Assigning a second entry for each parameter
2) Setting PadOutVectors: True, which repeats single entries for all particles defined in the PDG sequence

Running **sim_tutorial_gennon0_T0.fcl:**

```
%MSG-s ArtException: SingleGen:generator@Construction 20-Oct-2026 07:40:40 CDT
ModuleConstruction
cet::exception caught in art
---- SingleGen BEGIN
       The P0,
       SigmaP,
       X0,
       Y0,
       Z0,
       SigmaX,
       SigmaY,
       SigmaZ,
       Theta0XZ,
       Theta0YZ,
       SigmaThetaXZ,
       SigmaThetaYZ
       T0,
       SigmaT,
       vector(s) defined in the fhicl files has/have a different size than the PDG
       vector
       and it has (they have) more than one value,
       disallowing sensible padding and/or you have set fPadOutVectors to false.
---- SingleGen END
%MSG
Art has completed and will exit with status 65.
```

Modifying Your Output

We can also combine these two solutions and only modify specific variables.

Example 3: We want to produce two particles, a 0.7 GeV electron and a 0.8 GeV muon, that are otherwise identical. We could instead modify our FHiCL to include:

sim_tutorial_gennon0_T0.fcl:

```
physics.producers.generator.PadOutVectors: true
physics.producers.generator.PDG: [ 11, 13 ] // electron and muon
physics.producers.generator.P0: [ 0.7, 0.8 ]
```

This will now run and produce the particles and energies that we expect.

Simulation Workflow

Simulation Workflow

The simulation chain in LArTPC experiments generally follows the same three-stage structure:

1) **Particle generation:**

    Deciding which primary particles are produced, along with their initial properties

2) **Particle propagation:**

    Simulating the physics of how those original particles move through the argon

3) **Detector simulation:**

    Simulating how the detector responds to the charge and light generated inside the detector

Particle Generation

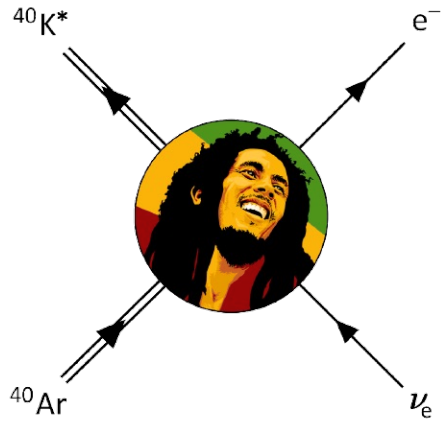
Typical LArSoft generators include:

- **Single particle gun:** Like the FHiCL we just created, this generator simulates single particles by specifying time, position and kinematics.
- **GENIE:** Typical neutrino interaction generator, provided an input flux.
- **MARLEY:** Low energy neutrinos, such as supernova and solar neutrinos.
- **TextFileGen:** Simulate particles based on an input text file (ie. in the hepevt format), such as the output of some BSM generators.

After this stage, the ART-ROOT file will typically include a number of simb::MCTruth objects

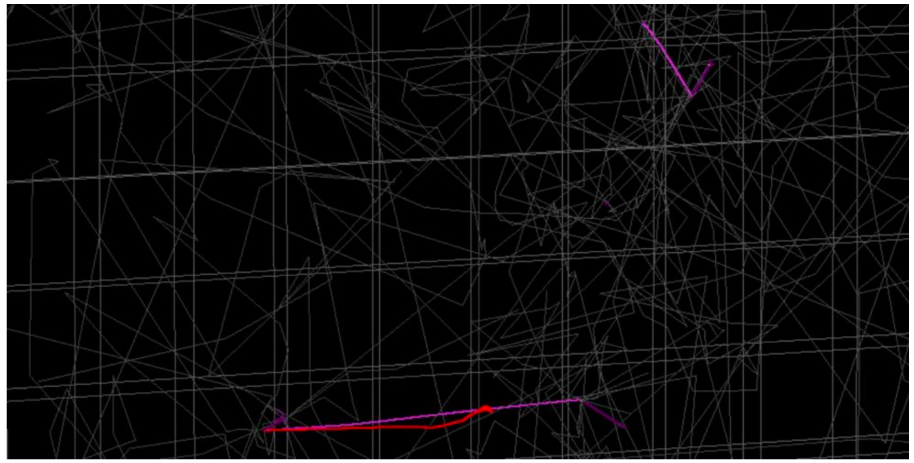
Genie
UNIVERSAL NEUTRINO GENERATOR & GLOBAL FIT

CORSIKA



Particle Propagation

- Output products of the generation stage are now propagated through the detector volume using Geant4.
- This handles the simulation of ionization, decays, argon interactions and electromagnetic showering as particles traverse the argon
- After this stage, the ART-ROOT file will typically include a large number simb::MCParticle objects.



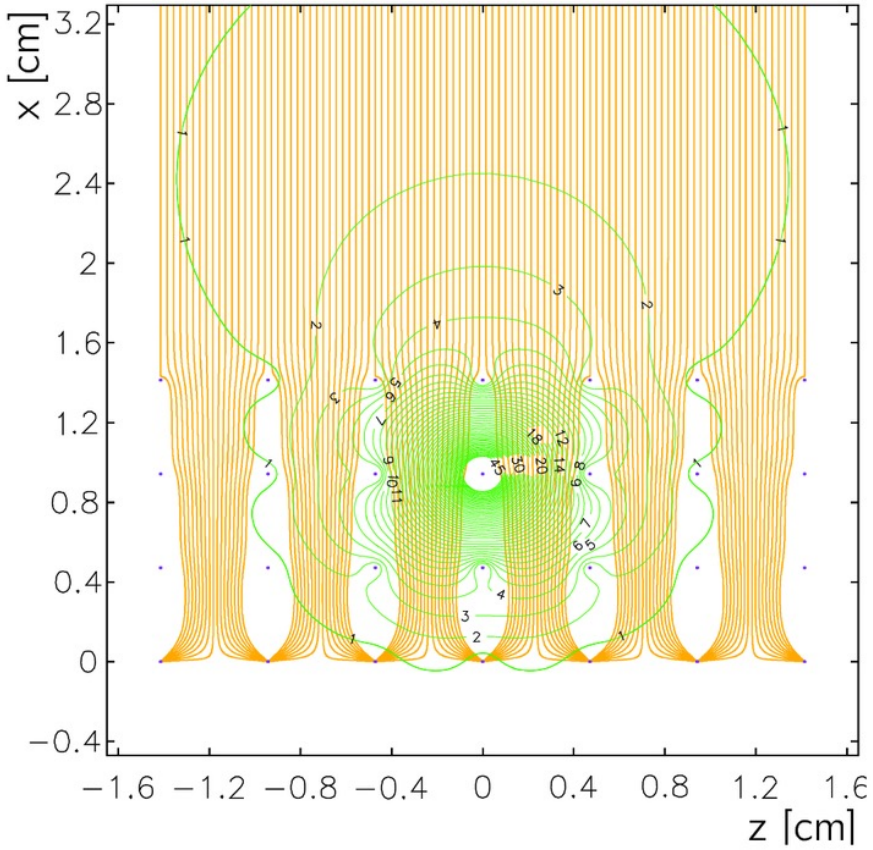
Detector Simulation

This stage simulates the detector response:

- Wire response to charge across each plane.
- PMT response to light – more details ~~tomorrow~~ later today!

We now have additional recob::wire and raw::OpDetWaveform objects in our ART-ROOT file, the simulated detector response to our input.

ProtoDUNE SP electron drift map



Simulation Workflow

Each of these stages has an associated FHiCL to use. For this tutorial, these will be:

1) Particle generation:
   sim_tutorial_gennon0_T0.fcl, the FHiCL you made earlier.

2) Particle propagation
   g4_workshop.fcl, included in your LArSoft build.

3) Detector simulation
   detsim_workshop.fcl, also included in your LArSoft build.

Each experiment will have its own standard version of these simulation FHiCLs. In the case of SBND, they are called **standard_g4_sbnd.fcl** and **standard_detsim_sbnd.fcl**

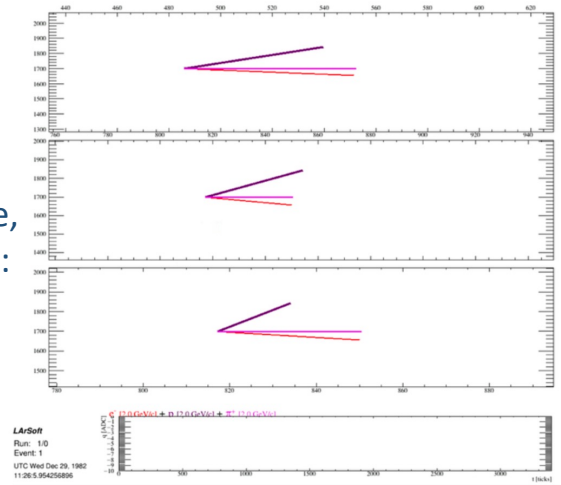
Checking Your Outputs

Once you've run through the simulation chain, it can be useful to inspect your generated events by eye. Run an **event display** using:

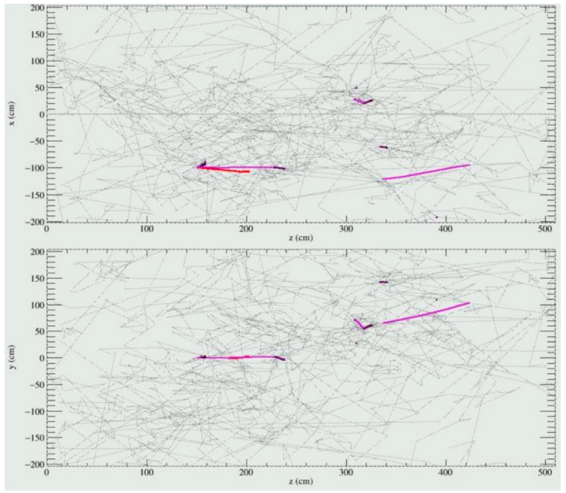
```
lar -c evd_sbnd.fcl -s your_detsim_output_file.root
```

There are three view options using **evd_sbnd.fcl:**

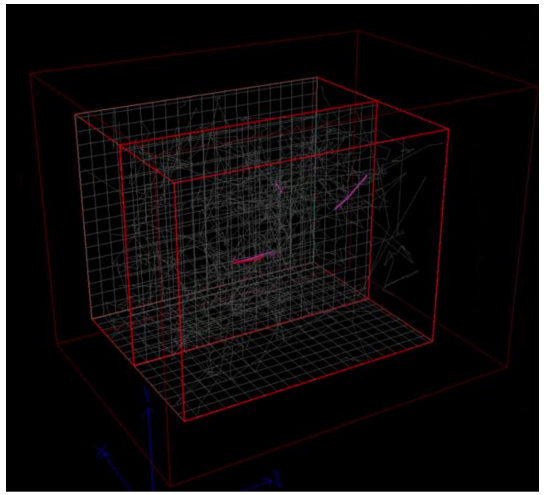
Time vs wire, charge view:



Ortho3D:



Display3D:



If each stage has run correctly, you should be able to see the paths your particles traced through the detector, each labelled by colour.

Checking Your Outputs

If you are working on a Fermilab GPVM in future and want to look at an event display, it's easiest to set up a VNC to view it.

- Useful instructions on how to do this are available on the SBN software wiki.
- Alternative event display tools are also available, such as TITUS for SBN, but are beyond the scope of this tutorial.

Running The Simulation

Task 2: Simulating 10 Events*

- Now we've gone through the simulation chain from generation to detector response, it's your turn to simulate some events.
- To start, try simulating a single particle for each event.
- Remember that the output of each step will be the input of the next!

*The detsim stage runs *very* slowly on the guacamole server and will probably be a bottleneck for you – feel free to generate and run geant4 over up to 10 events, but limit yourself to a maximum of three events when running the detector simulation

Simulating 10 Events*

Solution:

- In sim_tutorial_gennon0_T0.fcl:
  - physics.producers.generator.* sequences should contain one entry each.

- In the terminal:

```
lar -c sim_tutorial_gennon0_T0.fcl -n 10 -o output_gen.root
lar -c g4_workshop.fcl -s output_gen.root -o output_g4.root
lar -c detsim_workshop.fcl -s output_g4.root -o output_detsim.root -n 3
```

- Try using evd_sbnd.fcl to look at what you've made!

*Remember to limit the number of events you run detsim over to avoid waiting forever!

Main Task: Simulating “neutrino-like” 1μ1p with a “cosmic” μ

The main goal of this tutorial is to simulate 10 events, mimicking a neutrino interaction which produced one muon (13) and one proton (2212) coming from the same vertex, along with a separate muon coming in from above the detector mimicking a cosmic ray.

To do this, you’ll need to modify sim_tutorial_gennon0_T0.fcl to meet the following requirements:

- “Neutrino induced” muon: momentum = 0.7 GeV/c ; theta_xz = -10 deg ; theta_yz = 0 deg
- “Neutrino induced” proton: momentum = 0.7 GeV/c ; theta_xz = 35 deg ; theta_yz = 10 deg
- “cosmic” muon: momentum = 0.8 GeV/c; theta_xz = 30 deg ; theta_yz = -110 deg
- Start position for the “neutrino interaction” (x,y,z) = (-100,0,150) cm
- Start position for the “cosmic muon” (x,y,z) = (-120, 250, 300) cm
- Set all variations (vertex position, momentum, angles and time) to 0
- Set all distributions to “uniform” (vertex position, time and angle)
- Disable particles coming from the same vertex using SingleVertex: False

If you’re running low on time, a FHiCL meeting these requirements can be found here:

$MRB_SOURCE/sbndcode/Workshop/TPCSimulation/.solutions/sim_tutorial_gen_non0_T0_complete.fcl

Main Task: Particle Gun

- Once you've modified sim_tutorial_gennon0_T0.fcl, use lar with your FHiCL:

```
lar -c sim_tutorial_gennon0_T0.fcl -n 10 -o output_gen.root
```

- You can check the data products produced using:

```
lar -c eventdump.fcl -s output_gen.root -n 1
```

- Hopefully, you'll see the expected simb::MCTruth object in the output!

Main Task: G4 and DetSim

The next stages will be the same as previously:

```
lar -c g4_workshop.fcl -s output_gen.root -o output_g4.root
lar -c detsim_workshop.fcl -s output_g4.root -o output_detsim.root
```

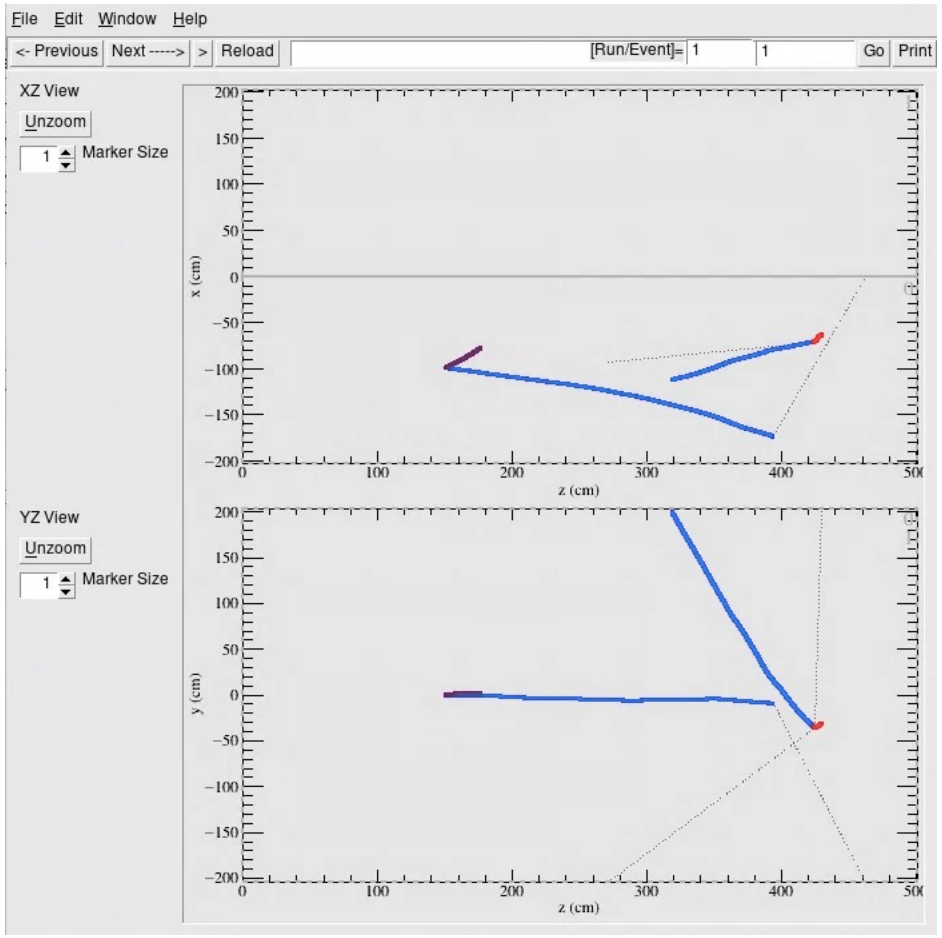
Once again, you can use eventdump.fcl to check that you have the products you expect in output_detsim.root.

Main Task: Visualising Events

You can also take a look at your events using the event display:

```
lar -c evd_sbnd.fcl -s output_detsim.root -n 10
```

If everything has gone correctly, you should see something like this (in Ortho3D view):



Once you're happy with your output, simulate another 10 events, adding a gaussian variation to theta_xz and theta_yz (keep the variation in the "cosmic-ray" muon small)

Bonus Tasks

If you manage to create a 10 event 1μ1p + μ sample matching the specifications outlined and gaussian angle variations, why not try the following:

- Simulate 1e1p events:
  - Repeat the steps of the main task, replacing the first muon (13) with an electron (11).
  - Can you distinguish the electron shower on the event display?

- Getting closer to a “real” event:
  - Try generating a couple of events with 1 muon and 1 proton, but now with 5 additional muons (cosmic rays) distributed randomly over the detector volume.
  - After running up to the detsim stage, what does your event look like now?

(Bonus) Bonus Tasks

Rather than using a particle gun to create our cosmic rays, we can use CORSIKA to generate muons for us.

- Generate 10 cosmic events using Corsika with this FHiCL:

  prodcorsika_cosmics_proton_sbnd.fcl

After the geant4 and detsim stages, does the event display look different to your previous tasks?

- Generate 10 neutrino and cosmic events using this FHiCL:

  prodgenie_corsika_proton_nu_spill_tpc_sbnd.fcl

How does this compare to your previous neutrino + cosmics sample? Take a look!

Contacts

You can ask any of us, but if you have particular questions about things in the simulation lecture or tutorial, you can find us on slack:

- @Magnus Handley
- @Adam Lister

There is also a dedicated channel for the simulation tutorial:

- #tpc_simulation

Backup samples

If you didn’t manage to make the standard samples, you can find some premade ones (with and without gaussian angle variations) in:

- **/Workshop/TPCSimulation/example_outputs**