LArSoft Simulation Tutorial

Anyssa Navrer-Agasson and Robert Darby
UK LArSoft Workshop 2024, Edinburgh, 28/10/2024

(Tutorial based on slides made by Aran Borkum and Marina Reggiani-Guzzo, thanks a lot!)

Goals of this tutorial

- Learn the basic syntax used in FHiCL files
- Write a functional FHiCL file to simulate particles
- Learn the basics of the lar command
- Run the LArTPC standard simulation

What is a FHiCL file?

FHiCL **(pronounced Fickle)** stands for Fermilab Hierarchical Configuration Language. Let's discuss the meaning of some of these concepts:

- **Hierarchical:** A FHiCL file can inherit parameters from a "parent" FHiCL file. This is a key feature given that LArSoft is object-oriented.

- **Configuration:** A FHiCL is a configuration file that lets us choose what to run and how to run it.

- **Language:** FHiCL files have a specific syntax that we should follow. This syntax is quite similar to the one used in JSON, we will see some examples later.

With time, you will learn to love FHiCL files. They allow us to avoid hard-coded values in whatever LArSoft module you are using. You can change any parameter without the necessity of recompiling everything again (Thus saving precious time). Also, any parameter defined will persist through the entire chain of FHiCL files that you run. For instance, you might want to generate some particles and carry on with the simulation until you can obtain something you can actually analyse (Typically in the form of a beautiful ROOT n-tuple)

NOTE: The extension of FHiCL files is **.fcl**

The FHiCL file syntax

The basis of the FHiCL file syntax is name-values pairs that you can use to declare different parameters.

1.- Parameters

```
pi: 3.14159
beam: "NuMI"
output_name: "my_beautiful_file.root"
```

Comments can be done in a C++ (// my comment) or Python (# my comment) way

```
pi: 3.14159  // This is a comment
beam: "NuMI"  # This is also a comment
output_name: "my_beautiful_file.root"
```

2.- Sequences

Sequences can be declared with square brackets and comma delimiters.

```
seq1: [1, 2, 3]   // A sequence of integers
seq2: [3.14, 2.17, "Gaussian"] // Sequence of floats and words
seq3: [1, 3.14, [2, "Gaussian"], 5] // Sequences are also
allowed as elements
```

Note: The starting index for any sequence is zero

Overriding of a sequence element is also allowed:

```
seq2[2]: "Uniform"  // "Gaussian has been changed by "Uniform"
```

3.- Tables

Tables are declared using curly braces:

```
tab1:{
   pi:3.1415
   beam: "NuMI"
   seq1: [3.14, 2.17, "Gaussian"]
}
```

Similar to sequences, we can also override a table value:

```
tab1.beam: "BNB"  // Now the beam is set to be “BNB”
```

We can refer to an entire table using **@local::var**

```
tab2: @local::tab1  // Now tab2 is the same as tab 1
```

One table can be spliced into another one by using **@tab::tab_name**

```
tab3:{
   @tab::tab1
   time: 1.9
}
```

This table is equivalent to

```
tab3:{
   pi:3.1415
   beam: "NuMI"
   seq1: [3.14, 2.17, "Gaussian"]
   time: 1.9
}
```

Prologs

Prologs contain parameter values that can be accessed by other files. Basically, we create a dictionary of possible parameter-value pairs that we can choose from. For instance, we can declare the energy of the beam that we want to use:

```
BEGIN_PROLOG
    bnb: 8        // 8 GeV beam
    numi: 120     // 120 GeV beam
END_PROLOG

BeamEnergy: @local::numi
```

In this example, BeamEnergy will have the value declared for numi in the prolog. This works if we are working with just one FCHiL file. Usually, this is not the case, in a real-case scenario you will be dealing with many parameters and multiple files, meaning that you might need to repeat the values for bnb and numi over and over. Lucky us, the hierarchical structure of FHiCL files is here to save us!

The best practice is to declare all the parameters using a prolog in an independent file. For instance, the previous code block can be contained in a FHiCL file called **beam_config.fcl**. Then, using **#include** we can import those values into a different FHiCL file (let's say **your_working_file.fcl**) as follows:

```
#include beam_config.fcl

BeamEnergy: @local::numi
```

Some of the reasons why keeping an independent file with the prolog are:

- Shorter FHiCL files in case you need to declare a lot of parameters
- Information is unified
- Tidier codes

[First Task] Creating your own FHiCL file

So far we have reviewed the FHiCL syntax and how to declare a set of parameters using a prolog. Now, we will learn how to write a FHiCL that can actually be run with LArSoft.

Start by creating a new (work) directory in your home folder and using your favourite text editor create a new file named **sim_tutorial_gen_non0_T0.fcl**

```
cd $HOME
mkdir simulation_tutorial
emacs -nw sim_tutorial_gen_non0_T0.fcl
```

The FHiCL files that you can run with LArSoft have a basic structure with specific fields that have to be present and filled in the right way. The structure is:

```
#include

process_name:

services:
{


}
source:
{


}
physics:{


}
outputs:{


}
```

Add each of these sections to your FHiCL. Now, as we start filling the file we'll look into the details of each section.

1.- Include statements:

In general, FHiCL files start by adding #include statements. This statement is used to tell the FHiCL from which FHiCL it will inherit. For our particular example, we should add:

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

2.- Process name:

Defines an overall name for the collection of modules that the FHiCL will run. This name has to be unique, the same process cannot be run multiple times over the same art-root file. For instance, if you want to run a reconstruction step on top of a previous reconstruction process (Reco) you can define a new process called: **process_name: Reco2**

The module that we want to use in this tutorial generates single particles, it is called SingleGen, so the next line that should be in your FHiCL is:

```
process_name: SingleGen
```

3.- Services:

The services table contains all of the simulation specific services that are generally used. For instance, detector geometry, physical properties, file management, etc.

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

The first table we included here loads SBND-specific configurations. The elements of this table are contained in the FHiCL included at the beginning **simulationservices_sbnd.fcl**.

4.- Source:

All the information regarding the input that the FHiCL takes will be contained here.

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

- module_type: EmptyEvent tells the FHiCL to start with an empty event (This file is actually the starting point of the simulation, so it should not expect anything). If the FHiCL were to accept a ROOT file instead, we use module_type: ROOTInput.

- maxEvents: 10 sets the number of events to simulate by default. If you set this value to -1 it will process all the events contained in the input file (if any). Later we will learn how to override this value at the moment of executing the file with LArSoft.

- firstRun and firstEvent provide the default starting values for the run and event numbers.

5.- Physics

The scope of this section is to declare and configure the modules that will be run over the input event.

```
physics:
{

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

What is each of this code lines doing?

- producers: Modules that go here add information to the ART-ROOT file -> Modifies the input file
- analyzers: Perform analysis using the input ART-ROOT file without modifying it.
- filters: Remove files that we are not interested -> Modifies the input file
- simulate: Declare in what order to run the producers
- stream1: Define the output stream
- trigger_paths: list everything that **will modify** the event (producers, filters)
- end_paths: list everything that **will not modify** the event (analyzer, outputs)

6.- Outputs

Declare where the output of this FHiCL will go:

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

Note that the “out1” has the same name as the value included in the physics table as “stream1”. The other lines do:

- @table::sbnd_rootoutput: Included at the beginning from “rootoutput_sbnd.fcl”
- fileName: Specify the default name of the output value. This can be overridden at the moment of executing the FHiCL with LArSoft.

Now with all these basic ingredients your file will 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"
```

```
    }
}
```

Note: A copy of this file can be found here:

```
$MRB_SOURCE/sbndcode/sbndcode/Workshop/TPCSimulation/sim_tutorial_gen_non0_T0.fcl
```

Great! We have a fully working FHiCL file that can be run with LArSoft. This file will actually simulate some particles for you. But first…

What are we simulating?

We have a working FHiCL, but what particle and with what properties we are simulating?

Let's dive into what FHiCL files we are importing to our own file. To do this, we will use the magic file **find_fcl.sh.**

NOTE: *A copy of this script lives here*

```
$MRB_SOURCE/sbndcode/sbndcode/Workshop/TPCSimulation/find_fcl.sh
```

You can copy and paste the script to your working directory.

Take your time and have a look at each of the files included in your FHiCL by doing:

```
source find_fcl.sh singles_sbnd.fcl
```

If your environment is properly set you should get something similar to:

```
/home/user28/larsoft_workdir/localProducts_larsoft_v09_91_02_prof_e26/sbndcode/v09_91_02/fcl/singles_sbnd.fcl
```

Open this file using your favourite editor **(hopefully something used by cool people as emacs)**, you will see the following:

```
#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
```

In this particular fcl we have access to some of the parameters that we can customize. For instance the energy (P0), initial time (T0) and starting position (X0,Y0,Z0).

Unfortunately, we still do not have information about what particle we are simulating. What we are looking for is the Monte Carlo PDG code of the particle.

NOTE: *You can check the full list of PDG codes here:* https://pdg.lbl.gov/2007/reviews/montecarlorpp.pdf

----

Let's dive a bit deeper. Now run

```
source find_fcl.sh singles.fcl
```

Again, if everything is correctly set, you should get the following path:

```
/cvmfs/larsoft.opensciencegrid.org/products/larsim/v09_91_02/job/singles.fcl
```

Open the file, you should see this:

```
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
```

Three things are worth our attention here:

First, the **module_type: "SingleGen"** refers to the actual LArSoft module that this FHiCL executes. The **module itself is a C++ file** with the respective functions to be executed over the input event. You can find the details of the module here:

https://github.com/LArSoft/larsim/blob/develop/larsim/EventGenerator/SingleGen_module.cc

We won't go into the details on how this C++ has to be written, you will actually to this later in the workshop. For now, you need to know that the module_type variable has to be the name of the .cc file (before the _module.cc).

Second, the table standard_singlep contains the full list of parameters that the "SingleGen" module accepts. Among these parameters, **PDG** is the one that tells you what particle we are simulating. In this FHiCL file, this value is hardcoded to 13 which corresponds to a muon with an energy of 6 GeV.

Now you might wonder, What if I want to simulate something different? Other particles, other kinematics and so on…

Overriding Parameters

Let's say that instead of a muon, now we want an electron (PDG number 11). To do this we need to replace the respective original value contained in **singles.fcl**. Notice how the table declared in **singles.fcl** is passed to **singles_sbnd.fcl** and eventually to our own fcl. If you correctly keep track of how the original table is inherited in our FHiCL and if you recall the way to override a parameter contained in a table, you will realize that we need to write the following at the end of our file.

```
physics.producers.generator.PDG: [11]   # This replaces the hardcoded muon by an electron
```

—---

NOTE: *An alternative way to obtain a list of the parameters that you can override is to use the command fhicl-dump:*

```
fhicl-dump sim_tutorial_gen_non0_T0.fcl > sim_tutorial_gen.txt
```

This command will print the set of configured parameters in the entire hierarchy to a text file.

—--

What if we want more than one particle? For instance, for an electron and a muon, you should write

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

Do not forget to provide values for all the particles declared.

NOTE: *If you try to run the FHiCL with this modification you will get the following error:*

```
%MSG-s    ArtException:       SingleGen:generator@Construction
23-Oct-2023 18:02:23 BST  ModuleConstruction
cet::exception caught in art
---- SingleGen BEGIN
 The 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.
```

This is pointing that some parameters have missing entries. To avoid this issue you need to set PadOutVectors to true.

```
physics.producers.generator.PadOutVectors: true
```

This will fill any vector not declared by you with default values in order to match the size of the vectors PDG and P0.

The Simulation workflow

In general, the full simulation chain in LArTPC experiments starts with the simulation of primary particles inside the detector, propagation of secondaries and simulation of the detector response.

1.- Particle generation

Single particle gun: The FHiCl file we just created is used to simulate single particles by specifying their time, position and kinematics.

GENIE: Used to simulate neutrinos provided an input flux

CORSIKA: For cosmic ray simulation

MARLEY: Supernova and solar neutrinos

TextFileGen: This is a special module used to generate to simulate particles from an input text file. This is typically the case in BSM generators.

After running any of these generators, the output ART-ROOT file will contain a list of **simb::MCParticle** objects.

2.- Propagation of particles

The propagation of any particle inside the detector volume is done using Geant4. This software handles the simulation of ionisation processes, decays, collisions with argon atoms, showers, etc.

After running this step, a new list of **simb::MCParticle** objects will be added to the input ART-ROOT file.

3.- Detector simulation

This is the last step of the simulation covered in this tutorial. In this stage, the detector response to the charge and the light generated inside the detector is simulated.

After this step, an object called **raw::RawDigit** will be added to the input ART-ROOT file. This object contains the collection of digitised charge vs time of each wire.

How to interact with LArSoft

In order to do anything with LArSoft we need to use the command **lar.** Typically the syntax of this command is:

```
lar -c your_fhicl.fcl -s input_file.root -o output_file.root -n 5
```

Let’s check what each of the parameters given to lar do:

- -c: tells LArSoft the file you want to run
- -s: tells LArSoft what your input file to use (Only if the FHiCL requires one)
- -o: The name of the output file. Please be sure that the name of the output is meaningful
- -n: How many events you want to simulate, set it to -1 to run over all the events contained in the input file.

There are more parameters that can be passed to lar, to check in detail you can run **lar -h**

Cool, now we now the syntax of the command lar, we can actually run the full simulation chain. As discussed in the previous section there are three simulation steps, each of which is run by a specific FHiCL file. You can find each of these files here:

```
$MRB_SOURCE/sbndcode/sbndcode/Workshop/TPCSimulation/
```

The list of files is:

- Particle Gun: sim_tutorial_gen_non0_T0.fcl
- Propagation: g4_workshop.fcl

- Detector simulation: detsim_workshop.fcl

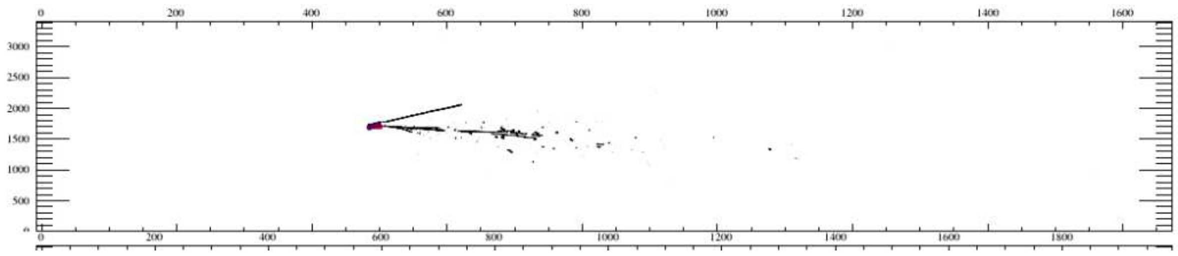
NOTE: *Each experiment has its own version for some of these FHiCL. For instance, in SBND you can find **standard_g4_sbnd.fcl** and **standard_detsim_sbnd.fcl**.*

We can also run the SBND event display to inspect by eye our generated events through the following FHiCL:

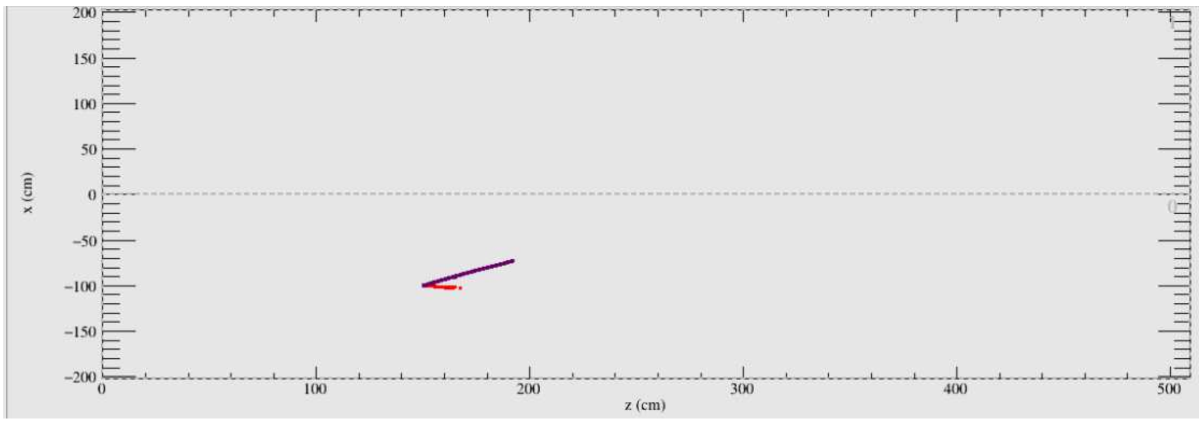
```
lar -c evd_sbnd.fcl -s your_detsim_output_file.root
```

The SBND event display has three possible views:

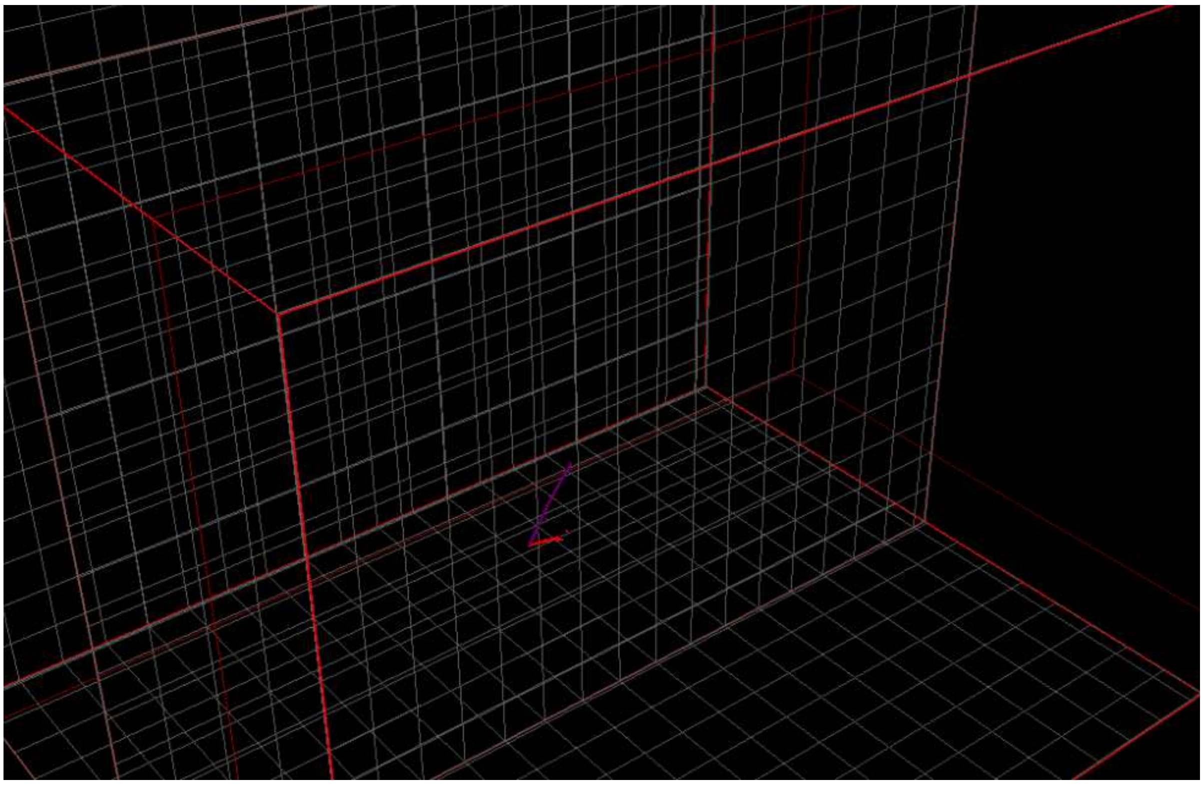
Time vs wire:



Ortho 3D:



Display 3D:



IMPORTANT: Running the event display without a VNC is extremely slow! This can actually happen if you are working in a FNAL GPVM. There is a way to setup a VNC in a FNAL machine but it goes beyond the scope of this tutorial, for more details go here:

https://cdcvs.fnal.gov/redmine/projects/sbndcode/wiki/Viewing_events_remotely_with_VNC

[Second Task] lar syntax:

How would you simulate 10 single particles up to the detector simulation? Bear in mind that the output of one step is the input of the next one.

Solution to the first task:

```
lar -c sim_tutorial_gen_non_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
```

Running your own Simulation

Equipped with all the machinery we learned so far, we will actually simulate the particles that we fancy with the parameters that we want. At the end, we will check how they look inside the SBND detector.

First, we need to be sure that our development environment is properly set.

1.- [Recap] Workspace setup

If you are starting in a fresh terminal remember setup your working environment first. Probably you created an specific bash script for this in the previous session (**setup.sh**)

```
cd $HOME
cd larsoft_workd
source
/cvmfs/sbnd.opensciencegrid.org/products/sbnd/setup_sbnd.s
h
source localProducts_larsoft_v09_78_02_prof_e20/setup
mrbsetenv
mrbslp
```

fffff2.- [Main task] Simulate 1mu1p events

Go to your simulation directory:

```
cd $HOME
cd simulation_tutorial
```

Use your own FHiCL or copy the one living inside the sbndcode repo:

```
$MRB_SOURCE/sbndcode/sbndcode/Workshop/TPCSimulation/sim_t
utorial_gen_non0_T0.fcl
```

Simulate 10 events with 1 muon (13) and 1 proton (2212). First, you need to modify the necessary parameters in **sim_tutorial_gen_non0_T0.fcl** to match the following requirements:

- Muon: momentum = 0.7 GeV/c ; theta_xz = -10 deg ; theta_yz = 0 deg
- Proton: momentum = 0.7 GeV/c ; theta_xz = 35 deg ; theta_yz = 10 deg
- Start position for both particles (x,y,z) = (-100,0,150) cm
- Starting time T0 for both particles = 1600 ns
- Set all variations (vertex position, momentum, angles and time) to 0
- Set all distributions to “uniform” (vertex position, time and angle)
- Set particle being created from the same vertex: SingleVertex: True

1.- Particle gun

Then run the FHICL file using the command lar as follows:

```
lar -c sim_tutorial_gen_non0_T0.fcl  -o output_gen.root -n 10
```

Now check what products have been created in the ART-ROOT file **output_gen.root:**

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

Hopefully simb::MCTruth should be present ;)

================ GO BACK TO THE LECTURE FOR NOW ===================

2.- Geant4 and detector simulation

Run the g4 and detsim steps by doing:

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

Let’s check again what products have been added to the ART-ROOT file using the event dump FHiCL.

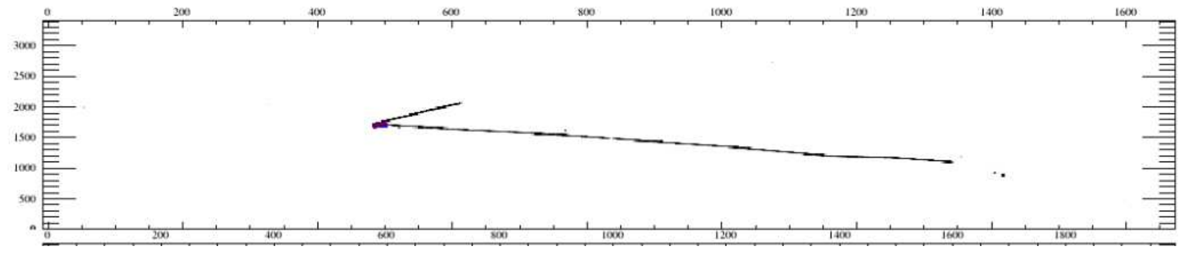
```
lar -c eventdump.fcl -s output_detsim.root -n 1
```

Can you spot any of the products mentioned in the simulation workflow section?

At this stage, visualize the simulated events using the SBND event display:

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

If everything is ok you should see something like this:



The solution to this task can be found here:

```
$MRB_SOURCE/sbndcode/sbndcode/Workshop/TPCSimulation/.solution
s
```

Following the same approach, simulate another 10 events but this time adding a Gaussian variation to the angles

Spare time?????? Move to task 3.-

3.- [Bonus task] Simulate 1e1p events

Repeat the same steps as in the 1mu1p task but change the PDG code of the muon by the PDG code of an electron (11).

- Can you identify the electron shower using the event display?

References:

Slides from previous versions:

Quick start to the FHiCL syntax

UK LArSoft Workshop (2022)

UK LArSoft Workshop (2021)

SBND Newbie material (Some of the info here might be outdated)