MANCHESTER 1824
The University of Manchester

Simulation Tutorial

Marina Reggiani-Guzzo

UK LArSoft Workshop 2022 - Lancaster, 08/11/2022

What are you going to learn in this tutorial?

1. Understand how configuration files (FHiCL) work in LArSoft and how to write one
2. Understand how to run a simulation
3. Simulate some events up to the detector response stage
   a. Generation (Gen) → Propagation (LArG4) → Detector Simulation (DetSim)

Tutorial heavily inspired by the last years' ones! Thank you for the previous speakers!

Is your environment set-up?

This tutorial requires a workspace with a compiled **sbndcode** with the branch **uk_larsoft_workshop_2022** checked out

(see David's tutorial)

Once your workspace is set up, you are ready for this tutorial… let's go?

FHiCL files

What is a FHiCL?

FHiCL = Fermilab Hierarchical Configuration Language

Is it possible to inherit FHiCLs and their parameters when you are writing your own FHiCL file.

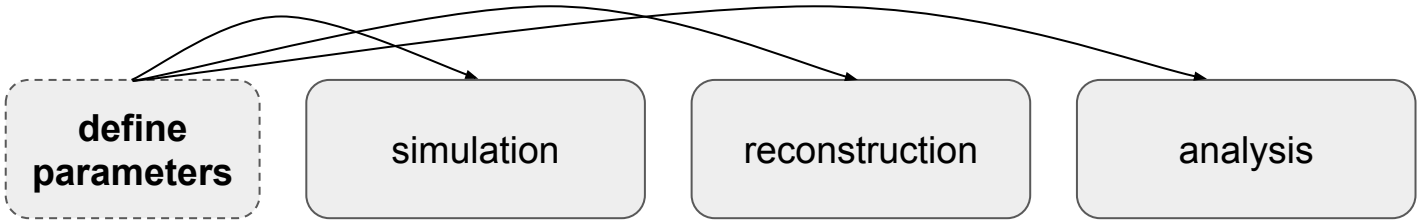
A FHiCL (a.k.a. fcl) enables us to configure and run LArSoft modules

A FHiCL is written in a specific syntax (similar to JSON). We will see its structure later on in this tutorial, stay strong!

You can easily identify a FHiCL file by its extension **.fcl**

Why use a FHiCL file?

- The hierarchical feature of a FHiCL file allows us to define parameters only once that will persist throughout the entire simulation/reconstruction/analysis code



- But if necessary you can easily change a parameter on the fly without having to recompile anything (for instance, if you want to run a quick test). We are going to see how it works shortly…

FHiCL Language

Quick introduction on the **syntax** of FHiCL.

Everything is based on **name-value pairs**

```
n: 1
pi: 3.14159
charge: “positive”
```

Comments can be done with # or //

```
n: 1              # this is a comment
pi: 3.14159       // this is a comment as well
charge: “positive”
```

FHiCL Language - Sequences

Sequences are defined by square bracketed **[ ]** with comma delimiters

```
seq1: [1, 2, 3, 4]              # basic sequence
seq2: [1, 2, "word"]            # sequence with numbers and words
seq3: [1, [2, 3], 4, "word"]    # mix of everything
```

You can also overwrite any of the entries

```
seq1[1]: 10          # 2 changed to 10, now seq1: [1, 10, 3, 4]
```

Remember that the sequence entries start from “zero”

FHiCL Language - Tables

Tables are enclosed in curly braces { }

```
tab1:
{
    a: 123
    b: “this is a table”
    seq1: [1, 2, 3, 4, 5]
}
```

And overwriting works similar to before

```
tab1.a: 246    # overwrite 123 with 246
```

Entire tables can be referred using **@local::var**

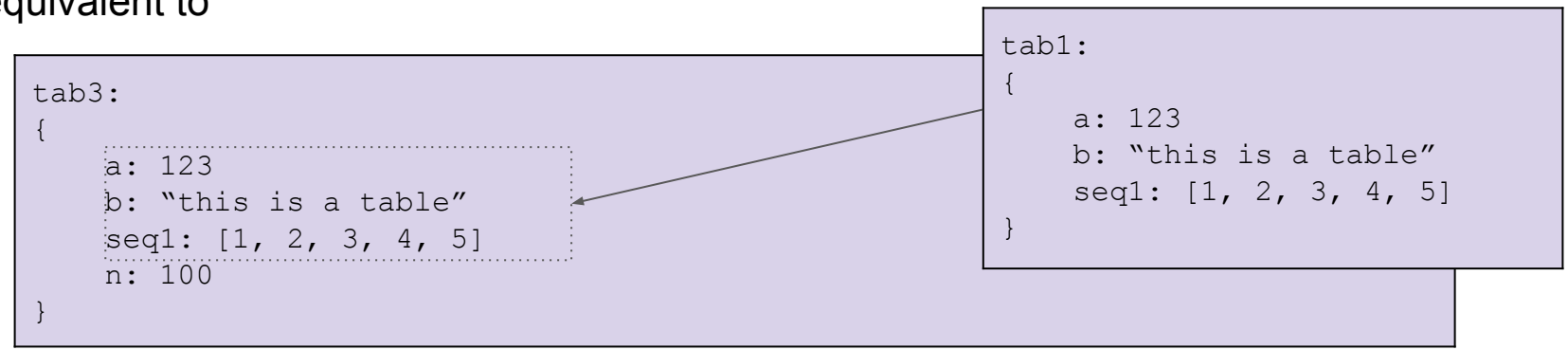
```
tab2: @local::tab1    # tab2 is now the same as tab1
```

FHiCL Language - Splicing

You can splice tables together using a reference **@table::tab_name**, for instance:

```
tab3:
{
    @table::tab1
    n: 100
}
```

Is equivalent to



FHiCL Language - Prologs

Prologs contain configurations that can be accessed in other fcl files. The idea is to have a dictionary of possible values we can choose from. For instance, one can define the possible beam energies as:

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

And in your code you can simply choose which beam energy you want to use:

```
BeamEnergy: @local::numi  choose numi beam energy
```

How is it usually structured in a FHiCL?

One option is to have everything written in a single FHiCL file

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

BeamEnergy: @local::numi
```

But that is not how it is usually done, PROLOGs tend to be written in a separate FHiCL that is inherited by other FHiCL files:

beam_config.fcl

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

your_working_fhicl.fcl

```
#include "beam_config.fcl"

BeamEnergy: @local::numi
```

FHiCL - Configurations

- Instead of writing long files, and repeating all the information over and over again, we can write our **configurations** in one file and include it in another
- Guarantees that the information is unified
- Shorter, cleaner and tidier codes!

Writing your first FHiCL

FHiCL file formats

The FHiCL files you actually run have a very important structure and some fields that have to be there and need to be filled out properly. The overall structure is:

```
#include

process_name:

services: { }
source: { }
physics: { }
outputs: { }
```

The next slides will explain the role of each part and what kind of information they take. Do not worry in memorising anything now, focus on understanding the overall structure of this file.

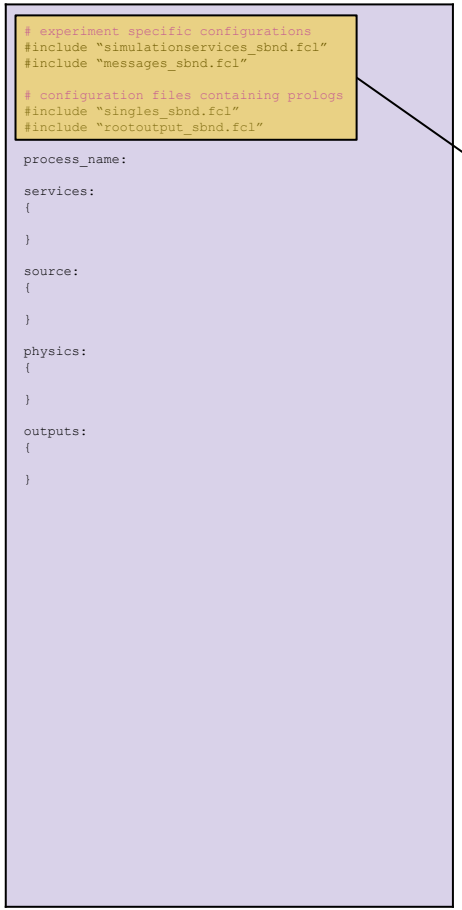
Writing your FHiCL: Skeleton

```
#includes

process_name:

services:
{

}

source:
{

}

physics:
{

}

outputs:
{

}
```

This is the skeleton of a FHiCL file

Let’s complete it together and understand the function of each part (you will have access to the final and complete file at the end)

Writing your FHiCL: **#includes**



Different experiments have their own files and configurations.

In general, FHiCL files start by inheriting include files:

```
# experiment specific configurations
#include "simulationservices_sbnd.fcl"
#include "messages_sbnd.fcl"

# configuration files containing prologs
#include "singles_sbnd.fcl"
#include "rootoutput_sbnd.fcl"
```

If you want to see what's inside a specific FHiCL file, you can find out its path by using this very useful command:

```
find_fhicl.sh simulationservices_sbnd.fcl
```

Writing your FHiCL: **process_name**

```
# 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:
{

}
```

Define overall name for the collection of modules you are running.

Must be unique, so you cannot have the same process name run multiple times on the same art-root file. If you are running some reconstruction but a process called Reco has already been run, define a new one called process_name: Reco2

The module that generates single particles is called **SingleGen**:

```
process_name: SingleGen
```

Writing your FHiCL: **services**

```
# 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:
{

}

physics:
{

}

outputs:
{

}
```

Services table contains all of the simulation specific services that are commonly used, for instance: detector geometry, physical properties, file managements…

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

Load in SBND specific configuration, table included from simulationservices_sbnd.fcl

Naming the output ROOT file

Writing your FHiCL: **source**

```
# 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:
{

}

outputs:
{

}
```

This is where we specify the **input information** (or source)

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

Start with an empty event. If we have an input ROOT file, specify **ROOTInput** here.

Default number of events to generate. Use **-1** if you want to run over all events in a file.

Default run and event number.

Writing your FHiCL: **physics**

```
# 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:
{

}
```

Define and configure modules that do work on the event

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

Add information to the ARTROOT file
➔ Modifies file

Perform analysis on the ARTROOT file
➔ No modification

Remove events we are not interested
➔ Modifies file

Writing your FHiCL: **physics**

```
# 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:
{

}
```

Define and configure modules that do work on the event

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

Defines the modules in the order you want to run them in

Define the output stream if you need it (configured later in the file)

Everything that modifies the event (filters/producers)

Everything that does not modify the event (analyzers and output streams)

Writing your FHiCL: **outputs**

```
# 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"
    }
}
```

Define where the output goes:

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

Note: same name used in **stream1: [ ]** in **physics: { }**

Included from **rootoutput_sbnd.fcl**

Specify the default output filename here. Check here to see more options on how to configure the name.

How do I choose what to simulate?

```
# 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"
    }
}
```

You can define a few initial properties:

- PDG: Particle ID
- P0, SigmaP, PDist: Momentum
- X0, Y0, Z0: Initial position
- And much more!

Full file here:
$MRB_SOURCE/sbndcode/sbndcode/Workshop/TPCSimulation/sim_tutorial_gen_non0_T0.fcl

Trust me, this is a complete and working FHiCL ready to simulate particles. But what is it simulating?

Find out what this file is simulating

Hints:
- Explore the input files (and their inputs)
- Use the command **find_fhicl.sh** (slide 14)

What am I simulating? (solution)

Exploring the include files…

```
# experiment specific configurations
#include "simulationservices_sbnd.fcl"
#include "messages_sbnd.fcl"

# configuration files containing prologs
#include "singles_sbnd.fcl"
#include "rootoutput_sbnd.fcl"
```

What am I simulating? (solution)

Exploring the include files…

```
# experiment specific configurations
#include "simulationservices_sbnd.fcl"
#include "messages_sbnd.fcl"

# configuration files containing prologs
#include "singles_sbnd.fcl"
#include "rootoutput_sbnd.fcl"
```

```
#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
```

This file contains some information about the particle (T0, momentum, initial position…) but not the particle ID. So the question remains: what are we simulating?

What am I simulating? (solution)

Exploring the include files…

```
# experiment specific configurations
#include "simulationservices_sbnd.fcl"
#include "messages_sbnd.fcl"

# configuration files containing prologs
#include "singles_sbnd.fcl"
#include "rootoutput_sbnd.fcl"
```

Remember about the hierarchical structure and… voilà!

```
#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
```

We are simulating a muon, PDG = 13

```
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: requ
                                       # true:  pad
 PDG:                   [ 13 ]         # list of pd
 P0:                    [ 6. ]         # central val
 SigmaP:                [ 0. ]         # variation a
 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
```

What am I simulating? (solution 2)

Another possibility is to use the fhicl-dump command, that prints the entire set of configured parameters in the entire hierarchy:

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

The output can be long, so best put it into a text file where you can search easily

Output of the fhicl-dump command, saved in sim_tutorial_gen_dump.fcl

```
# Produced from 'fhicl-dump' using:
#   Input  : sim_tutorial_gen.fcl
#   Policy : cet::filepath_maker
#   Path   : "FHICL_FILE_PATH"

outputs: {
   out1: {
      compressionLevel: 1
      fileName: "prodsingle_sbnd_%p-%tc.root"
      module_type: "RootOutput"
      saveMemoryObjectThreshold: 0
   }
}
physics: {
   end_paths: [
      "stream1"
   ]
   producers: {
      generator: {
         AngleDist: "Gaussian"
         P0: [
            6
         ]
         PDG: [
            13
         ]
         PDist: "Gaussian"
         PadOutVectors: false
         ParticleSelectionMode: "all"
         PosDist: "uniform"
         SigmaP: [
            0
         ]
(...)
```

Input file

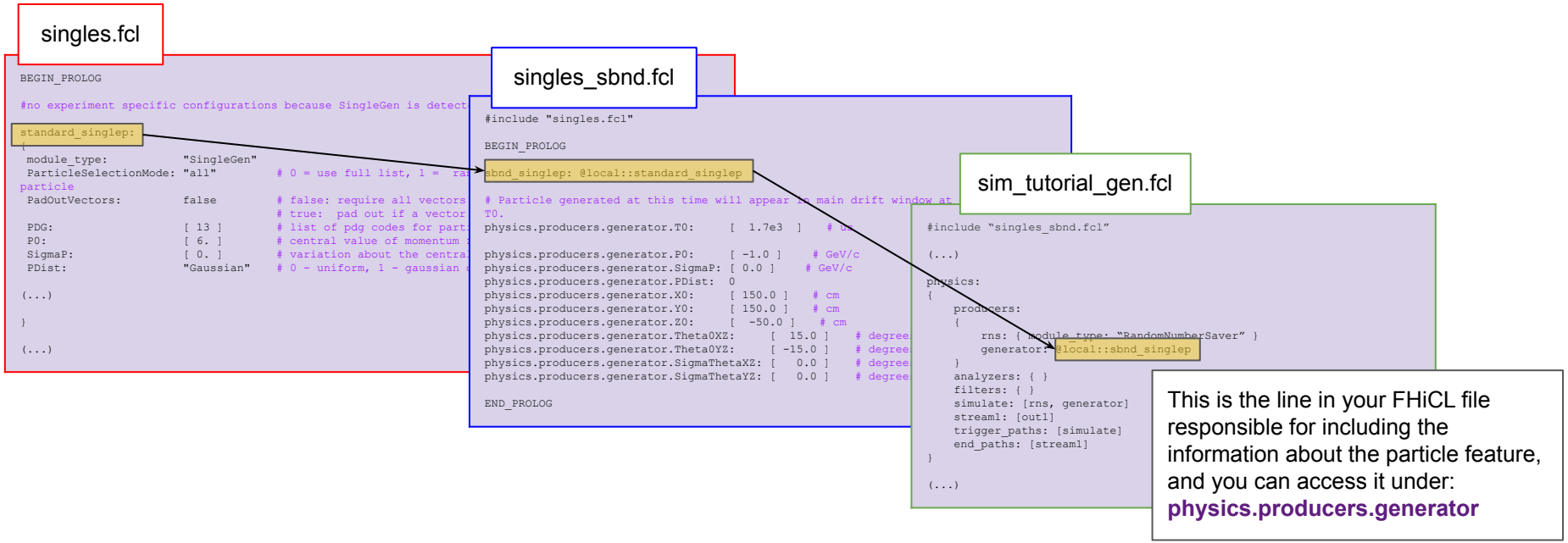
A muon is being simulated

The configured parameters defined in the previous slide will be displayed here.

What if I don’t want to simulate a muon?

One of the beauties of FHiCL files is that you can **override** parameters on the fly.

First, identify where the parameter is declared and define its “path”. Example: particle ID



Overriding parameters

You can overwrite a parameter in your FHiCL file (the ones displayed when you run fhicl-dump) after you define them by adding the following line at the end of your FHiCL file:

```
physics.producers.generator.PDG: [ 11 ] # this is an electron
```

This is true for any other parameter and for N≠1 particles:

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

Make sure to include the initial parameters for all particles

Modify the particle ID in your FHiCL and check the new fhicl-dump output. Do it for an electron and for a pair of particles. (~10 min)

Understanding the Simulation Flow

Generating Particles

The first step is to generate the particles we want to simulate, there are a few options:

1. **Single particle gun**: specify the topology of the collection of particles you want to study

2. **Generators**: use an input flux and/or physics model to simulate interactions
   a. **GENIE**: for generating neutrinos
   b. **CORSIKA**: for cosmic rays
   c. **MARLEY**: for supernova and solar neutrinos
   d. People doing BSM either write custom generators or develop additions to GENIE

This tutorial focuses on the single particle gun (developed FHiCL file so far in the tutorial). This generates a particle with some initial parameters: start position, start momentum, PDG code, energy…

The output of this is a list of simb::MCParticle objects.

Geant4

The next step is to propagate the particles through the detector, this is done with Geant4

Geant4 simulates all the physical processes that happen in the detector:

- Collisions with argon atoms
- Ionisation processes
- Showers
- Decays
- Etc…

The output is a also list of **simb::MCParticle** objects.

Detector Simulation

The last simulation step covered in this tutorial is the detector simulation. This step returns the detector response to the charge and light generated in the detector during the propagation.

The output of this stage if the raw::RawDigit which is a collection of digitised charge vs time from a wire.

How to interact with LArSoft

lar commands

To do anything with LArSoft you need to use the **lar** command.

This is how you run a simulation. For the SingleGen stage, for instance, there is no input file.

A typical LArSoft command looks like:

```
lar -c your_fhicl.fcl -s inputFile.root -o outputFile.root -n 5
```

The most important flags you can pass to a lar command are:

- -c (--config): the FHiCL file you are running
- -s (--source): the source file (a ROOT file made by some previous stage)
- -n (-nevts): the number of events to run (use **-n -1** if you want to run over all events)
- -o (--output): the name of the output art-root file (overrides the default)
- --nskip: the number of events to skip

Detailed usage can be found with **lar -h**

Running the full simulation chain

As explained, the full simulation chain consists of three stages (generator, propagation, detector simulation), and as you might imagine, there is a FHiCL responsible for each of them:

Path: **$MRB_SOURCE/sbndcode/sbndcode/Workshop/TPCSimulation/**

- Generator: **sim_tutorial_gen_non0_T0.fcl**
- Propagation: **g4_workshop.fcl**
- Detector Simulation: **detsim_workshop.fcl**

For SBND:
- standard_g4_sbnd.fcl
- standard_detsim_sbnd.fcl

Note that the generator output file is used as an input for the propagation stage, and so on…

Hint: use the **-o** flag to name the output files in a way that makes sense to you

Based on the previous slide, what is the lar command for each of the steps above?

lar commands: solutions

Have you given a thought on the previous task?
If not, what are you doing here? Go back to the task!

A full simulation chain will look like this:

```
lar -c sim_tutorial_gen_non_T0.fcl -n 20 -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
```

Visualising Events

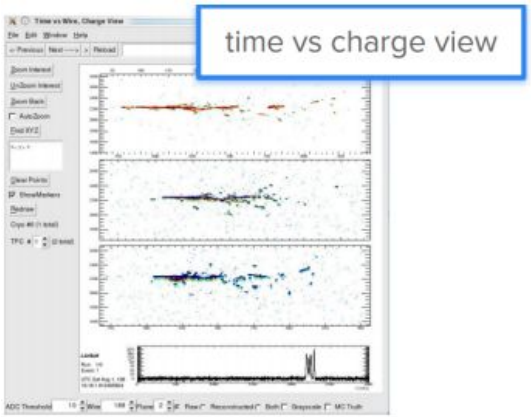
Event Display

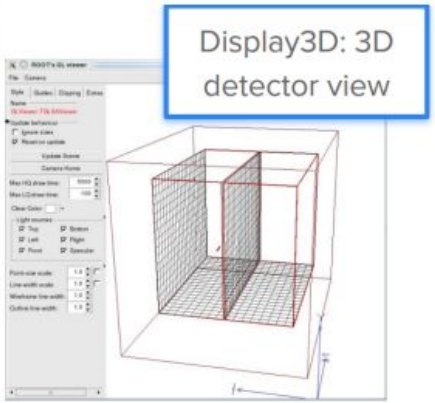
LArSoft has an event display that you can use to view the events and validate (by eye) your simulation.

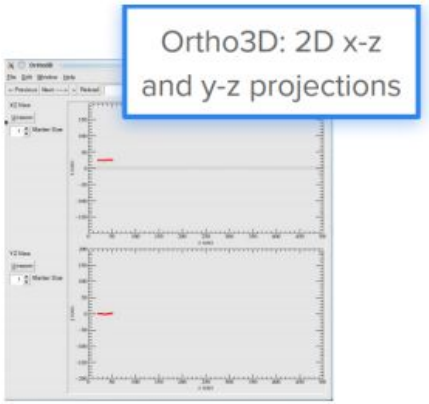
It can be very slow when not using VNC while working on the FNAL GPVMs

To run it use:

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







Running your own Simulation

Main task

- Copy the working FHiCL:
  $MRB_SOURCE/sbndcode/sbndcode/Workshop/TPCSimulation/sim_tutorial_gen_non0_T0.fcl
- Simulate 10 events with 1 muon and 1 proton with the following requirements:
  - Muon: momentum = 0.7 GeV/c ; theta_xz = -10 degrees ; theta_yz = 0 degrees
  - Proton: momentum = 0.8 GeV/c ; theta_xz = 35 degrees ; theta_yz = 10 degrees
  - Start position of both particles (x0, y0, z0) = (-100, 0, 150) cm
  - T0 of both particles = 1,600 ns
  - Set all variations (vertex position, momentum, angles, time) to 0
  - Set all distributions to “uniform” (vertex position, time, angle)
  - Set particles being created from the same vertex (SingleVertex:true)
- Run Geant4 over the produced particle file
- Run DetSim over the Geant4 file
- Run the Event Display over the DetSim file
- Repeat everything above with a gaussian variation to the angles

Bonus task

1. Repeat the main task and make 10 events with an electron instead of a muon

1. Generate 10 electron-neutrino events in the SBND active volume,
2. Run the Geant4 and Detsim stages
3. Open the events in the event display
4. Identify the electron shower and any other particles along with it

Extra material

- Quick Start Guide for FHiCL 3: The Fermilab Hierarchical Configuration Language

Solutions:

- **$MRB_SOURCE/sbndcode/sbndcode/Workshop/TPCSimulation/.solutions/**
  - **sim_tutorial_gen_non0_T0_complete.fcl** →FCL for the main task
  - **run_full_simulation.sh** → set of commands to run a full simulation using LArSoft

Vim

If you are not familiar with opening/editing a text file on the terminal, you can use **vim.**

- Open file: vim my_text.txt
- Edit file: esc + i (it will display -- INSERT -- on the bottom)
- Save file: esc + :w
- Save file and close file: esc + :wq
- Close file: esc + :q

Thanks!