Simulation Tutorial

Anyssa Navrer-Agasson and Joe Bateman

10th LArTPC Software Workshop - 27th October 2025

What you will be learning?

1. What a FHiCL file is and some basic syntax.
2. How to write your first FHiCL file to begin simulating particles.
3. How you can use `lar` to run your simulation.
4. To use Geant4 and `lar` to propagate your particles.
5. To simulate the detector response.

By the end of this tutorial, you should have generated a sample of 10 1μ1p events up to the detector simulation stage. This is a sample you'll be using in subsequent tutorials.

This tutorial is heavily inspired by the previous tutorials of Rob Derby, Luis Mora Lepin, Marina Reggiani-Guzzo and Aran Borkum. Thank you!

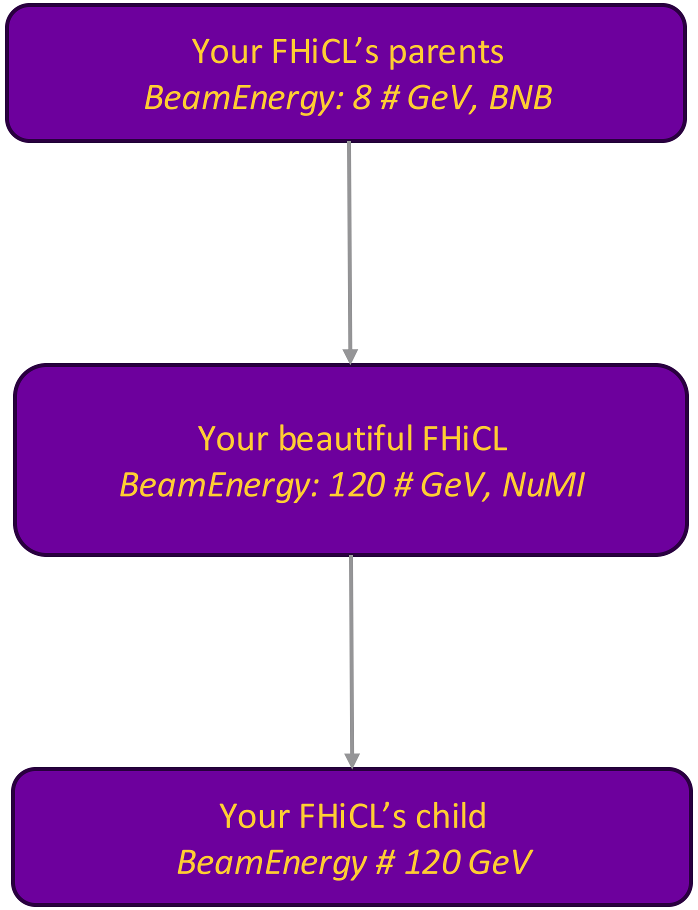
What is a FHiCL?

What is a FHiCL?

- FHiCL (pronounced “**fickle**”) stands for **Fermilab Hierarchical Configuration Language** to configure software built on `art`, such as **LArSoft**.
  - **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 FHiCLs?

- 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.



What will you need?

- This tutorial will require a workspace with **sbndcode**, and the workshop branch **uk_larsoft_workshop_2025** checked out and set up.
  - Refer back to Rachel's tutorial for more details!
- Starting from a fresh environment:
  - Start the container using:

```
source /scratch/LAR25/useful/container.sh
```

  - You likely made a setup script in the previous session (`setup.sh`) in your LArSoft directory. Use **`source setup.sh`** to run the script , or use the following commands:

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

(a copy of this script can also be found here: **/scratch/LAR25/setup/setup.sh)**

FHiCL Syntax

- 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 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
```

FHiCL Syntax

- 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:
  - Tables can be spliced together using `@table::name`.
  - Spliced tables inherit the entries each table called, in addition to any newly defined entries.

```
// 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
```

FHiCL Syntax

- Prologs:
  - We can use prologs to define useful configuration values that we may want to reference later, or in subsequent files.
  - If these configurations are defined in **`beam_config.fcl`**, we can use `#include` to inherit the prolog.
  - Then those configurations can be called using `@local::<var>`.

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

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

- You may be beginning to see why prologs are useful:
  - Instead of repeatedly defining parameters, we can define everything in higher level FHiCLs and inherit.
  - This leaves us with tidier files and information/parameters unified between workflows.

Creating a FHiCL file

Getting started

- Now we've reviewed some of the syntax, we'll learn how to write a FHiCL that can be run by LArSoft.
- To begin, make a new working directory in your home folder and use your preferred text editor to create a new file called `sim_tutorial_gennon0_T0.fcl`.

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

Basic structure

- FHiCL files that run with LArSoft have the same basic structure, with specific fields to be filled out :
  - **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

- **Include**
  - These behave as you have seen earlier, telling your FHiCL what files to inherit from.
- For this example, we should include the following FHiCLs:

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

```
#include

process_name:

services: {

}
source: {

}
physics:{

}
outputs:{

}
```

Basic structure

- **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`

```
process_name: SingleGen
```

```
#include

process_name:

services: {

}
source: {

}
physics:{

}
outputs:{

}
```

Basic structure

- **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 SBND-specific service configuration, and a root output:

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

```
#include

process_name:

services: {

}
source: {

}
physics:{

}
outputs:{

}
```

Basic structure

- **Source**
  - This table contains all information regarding the input that this FHiCL will take.
- **`module_type: EmptyEvent`** tells the FHiCL to begin with an empty event, as we are at the starting point of the simulation.

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

```
#include

process_name:

services: {

}
source: {

}
physics:{

}
outputs:{

}
```

Basic structure

- **`maxEvents: 10`** sets the default number of events to simulate. A value of **-1** will process every event in an input.
  - This can be overwritten at execution.
- **`firstRun`** and **`firstEvent`** dasg set the default start values for run and event.

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

```
#include

process_name:

services: {

}
source: {

}
physics:{

}
outputs:{

}
```

Basic structure

- **Physics**
  - This table declares and configures the modules that will be run over the input.
  - These are split into **`producers`**, **`analyzers`** and **`filters`**.
- **`Producers`:** modules here add information to the art-root file
  - **Modifies** the input file.

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

```
#include

process_name:

services: {

}
source: {

}
physics:{

}
outputs:{

}
```

Basic structure

- **Analyzers:** Perform analysis using the input file **without modifying** it.
- **Filters:** Removes files we aren't interested in.
  - **Modifies** the input file.

```
#include

process_name:

services: {

}
source: {

}
physics:{

}
outputs:{

}
```

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

Basic structure

- Configuring modules:
  - **`simulate`**: Declare the order to run producers.
  - **`Stream1`**: Define 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]
```

```
#include

process_name:

services: {

}
source: {

}
physics:{

}
outputs:{

}
```

Basic structure

- Configuring modules:
  - **trigger_paths** : lists everything that will modify an event (producers & filters)
  - **end_paths**: lists everything won'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]
}
```

```
#include

process_name:

services: {

}
source: {

}
physics:{

}
outputs:{

}
```

Basic structure

- **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`.
- The lines in `out1` inherit the `rootoutput` table and define the name of the output file (overwritable at execution).

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

```
#include

process_name:

services: {

}
source: {

}
physics:{

}
outputs:{

}
```

Your first FHiCL

- With all these building blocks in place, your FHiCL file 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"
     }
}
```

Note: 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**

- Now you have a simple FHiCL to simulate, how do we run it? If you've setup sbndcode in your container, you will have access to the `lar` command. To use it with your FHiCL, using the –c (--config) option:

  `lar -c sim_tutorial_gennon0_T0.fcl`

- If your FHiCL is formatted correctly, once the command has finished running you will see an event summary and the "Art has completed and will exit with status 0" message.
- You can also use options to override some of the 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 by using **`lar --help`**

Modifying your FHiCL

What is being simulated?

- This FHiCL can now be run by LArSoft, but what is it simulating?
- We can begin to work that out by looking at the FHiCLs we import into `sim_tutorial_gennon0_T0.fcl` using **`find_fcl.sh`**
  - A copy of the script that you can copy to your home directory lives here:

    `$MRB_SOURCE/sbndcode/sbndcode/Workshop/TPCSimulation/find_fcl.sh`
  - Copy the script into your working directory and use it to find `singles_sbnd.fcl`:

    `source find_fcl.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 what 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 energy: P0
  - Initial time: T0
  - Initial position: X0, Y0, Z0
- However, we still don't know what particle we're simulating.

```
#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

- We can look further back by opening `singles.fcl`:

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

- What we're looking for is the Monte Carlo PDG 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

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 other 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.
- **Note: this isn't the only way to get a list of parameters defined in a FHiCL**:

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

will write 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
```

- But would that work as we might expect?

Modifying your output

- 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`, repeating single entries for all particles defined in the PDG sequence.

Running **`sim_tutorial_gennon0_T0.fcl`** currently:

```
%MSG-s ArtException:  SingleGen:generator@Construction 20-Oct-2025 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 select 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

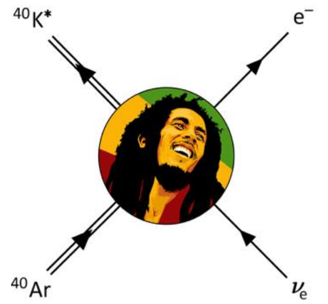
     Simulation of the primary particles being looked at.
  2. Particle propagation

     Handling of how these particles propagate through argon.
  3. Detector simulation

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

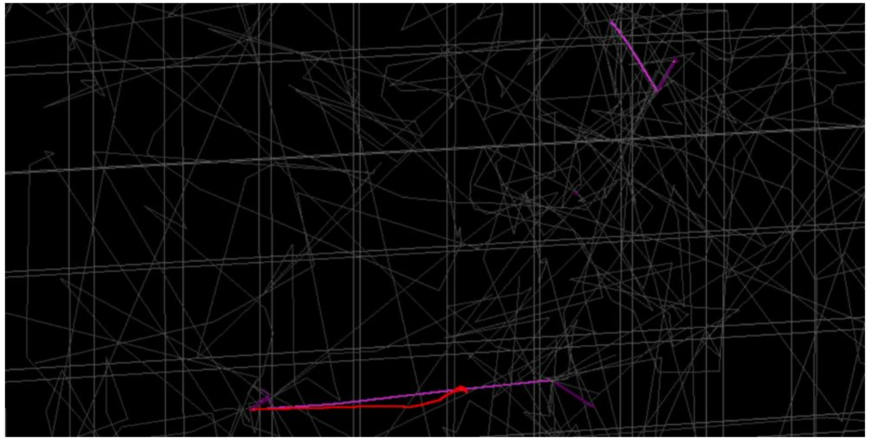
Particle generation

- 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.
  - **CORSIKA:** Cosmic ray simulations.
  - **MARLEY:** Low energy neutrinos, such as supernova and solar neutrinos.
  - **TextFileGen:** Simulate particles based on an input text file (ie. in the hepevt format), typically the output of a BSM generator.
- After this stage, the ART-ROOT file will typically include a number of simb::MCTruth objects.



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 showering that is seen as particles traverse a volume of 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.
- We now have additional recob::wire and raw::OpDetWaveform objects in our ART-ROOT file, the simulated detector response to our input.

Simulation workflow

- These stages each have an associated FHiCLs 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 version of these simulation FHiCLs. In the case of SBND, they are called `standard_g4_sbnd.fcl` and `standard_detsim_sbnd.fcl`

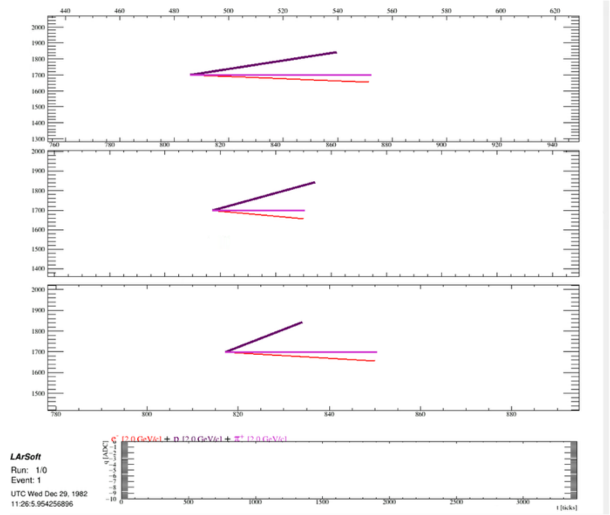
Checking your outputs

- Once you've run through the simulation chain, it can be useful to inspect your generated events by eye:

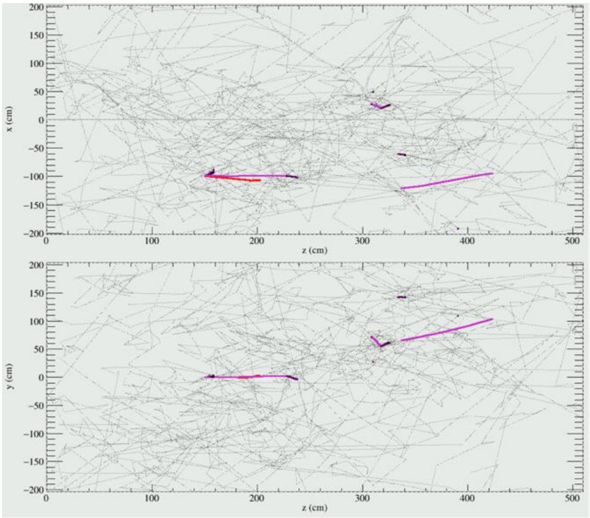
```
lar -c evd_sbnd.fcl -s your_detsim_output_file.root
```

- There are three view options using `evd_sbnd.fcl`:

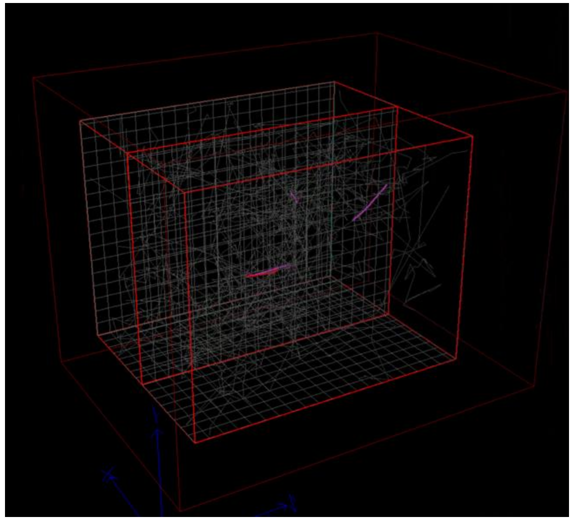
Time vs wire, charge view



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 color.

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, 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!

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
```

  - Try using `evd_sbnd.fcl` and have a look at the events you've generated.

Main Task: Simulating 1μ1p

- The goal of this tutorial is to simulate 10 events, each with 1 muon (13) and 1 proton (2212). To do this, you'll need to modify `sim_tutorial_gennon0_T0.fcl` to meet 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

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 -o output_gen.root -n 10
```

- You can quickly check the output file 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
```

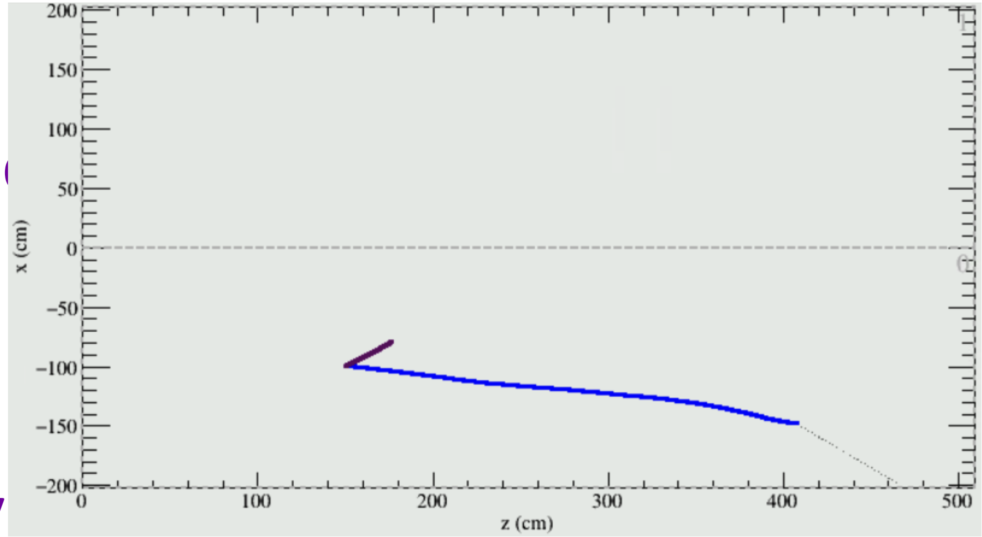
- Once again, you can use `eventdump.fcl` to check that you have the products you expect in `output_detsim.root`.

Main Task: Visualising events

- You can also check your sample using:

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

- If everything has gone correctly, you should see something like this:



- Once you're happy with your output, simulate another 10 events, adding a gaussian variation to theta_xz and theta_y

The University of Manchester
IMPERIAL

Bonus Tasks

- If you manage to create a 10 event 1μ1p 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 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 further 10 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?

* If you want to use this sample with later tutorials, try using `prodcorsika_proton_intime_sbnd.fcl` – This filters out cosmics that are not in time with a beam spill, so you'll need to start with a larger number of events (ie n=50, 100)

Questions?

- Feel free to ask Joe or Anyssa questions in-person or via Slack.
  - @Joe Bateman
  - @A Navrer-Agasson
- There is also a dedicated TPC-Simulation Slack channel:

  `#tpc-simulation`

Back-up samples

- In case you didn't manage to make a 1μ1p sample, there are premade samples here:

```
/scratch/LAR25/simulation/
```

References and Useful Links

- Previous tutorials:
    - 9th Software Workshop
    - 8th Software Workshop
    - 7th Software Workshop
    - 6th Software Workshop
- Other links:
    - SBN Software Wiki
    - SBND Newbie Material (may be slightly out of date)