# Simulation Tutorial

# Goals of the lecture

- **Understand how `FHiCL` files work and how to put one together**
- **Get to grips with lar commands**
  - `lar -c please_for_the_love_of_god_work_v8.fcl`
- **Generating your first events**
- **Running the event display**
- **A really simple analysis of your first events**

# What is a FHiCL file?

- **FHiCL or fcl (pronounced fickle, not faecal) files are Fermilab Hierarchical Configuration Language files**
- **What the hell does this mean?**
  - FHiCL files are the configuration files for different stages of larsoft
  - They let us choose what we want to run and how we want to run it
- **What does hierarchical mean?**
  - FHiCLs can inherit from FHiCLs which can inherit from FHiCLs which can inherit…
  - LArSoft is highly object oriented and parameters can be inherited from parent files
- **Is FHiCL its own language?**
  - You'll see FHiCL files are very JSON-ish
  - That's enough to call it it's own language… almost

# Why use FHiCL files?

- **It avoids having to hard code values into your larsoft modules**
- **More importantly, you can change these values on the fly without having to recompile anything!**
- **Also, you just have to**

# Using the FHiCL language

- **How do we define variables in a FHiCL file?**
  - Everything is based on name-value pairs
  - For example:

```
pi: 3.14
this_number: 17
mass_ordering: “normal”
```

  - Commenting can be done in Python or C++ style

```
comment_style: “Python” #  wow, look, a comment
comment_style: “C++”    // damn, another comment
```

# FHiCL sequences

- **All sequences are defined by square bracketed lists [] with comma delimiters**

```
list1: [1, “two”, 3]               # this is fine

list2: [6, [7, “Eight”], 9, 10] # this is cool too
```

- **You can also overwrite any of the entries after the fact**

```
list2[3]: 4 # 10 changed to 4
```

# FHiCL tables

- **Tables are basically dictionaries in python, they're enclosed in curly braces**

```
tab1:
{
  a: 123
  b: "I hope my code runs"
  list: ["you", "suck", "at", "coding"]
}
```

- **And overwriting works similar to before**

```
tab1.a: 456 # change the value of a from 123 to 456
```

- **Entire tables can be referenced using `@local::var`, like this**

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

# Table splicing

- **You can splice two tables together using a reference `@table::tab`**

```
tab3: {
  @table::tab1
  new_value: true
}
```

- **`Which is the equivalent to`**

```
tab3: {
  a: 123
  b: “I hope my code runs”
  list: [“you”, “suck”, “at”, “coding”]
  new_value: true
}
```

# Prologs

- **Prologs contain configurations that can be accessed in other files**
- **Writing a prolog lets us define alternative values to feed into our simulations**
- **They look like this**

```
BEGIN_PROLOG
numi: 120 # 120 GeV beam energy
END_PROLOG

BeamEnergy: @local::numi
```

# Includes

- **Instead of writing long, bulky files we can write our configurations in one file and include it in another**
- **We could write a file, MyBeamConfiguration.fcl, which contains the prolog from the previous slide**
- **We'll touch more on this later, but it's good to mention now**

# Structure of a complete fhicl

- **The FHiCL files you actually run have a very important structure and some fields that a) have to be there and b) need to be filled out properly**

- **The overall structure is**

```
#include

process_name:

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

- **Let's go through these one by one**

# Includes

- **Different experiments have their own files and configurations that go into each simulation**

- **In general you'll see:**
  - experiment specific configurations

```
#include "services_dune.fcl"
```

  - Configuration files containing prologs

```
#include "singles_dune.fcl"
```

- It can be super annoying trying to find these FHiCLs to see what's in there. You can use findfcl.sh to find them

```
./findfcl.sh singles_dune.fcl
```

*hint hint* keep this file It'll always be useful. I Literally can't stress that enough

# process_name

- **Smart people who write smart code have given smart names to the different modules they've made**

- **For example, the module that generates single particles is called `SingleGen` 🤯**

- **If you want to write a FHiCL for generating your own single particles you would add**

  **process_name: SingleGen**

- **These modules exist for generation, propagation, reconstruction, etc**

# Services

- **Services is where you put all of the simulation specific services for what you're trying to run**
  - This can mean things like:
    - Detector geometry
    - Physical properties
    - File management

```
services: {
  @table::dunefd_1x2x6_simulation_services
  TFileService: { fileName: "my_dank_file_name.root" }
  RandomNumberGenerator: {}
             }
```

# services

- **Services is where you put all of the simulation specific services for what you're trying to run**
  - This can mean things like:
    - Detector geometry
    - Physical properties
    - File management

SBND specific services
Loaded from
simulationservices_sbnd.fcl

```
services: {
  @table::dunefd_1x2x6_simulation_services
  TFileService: { fileName: "my_dank_file_name.root" }
  RandomNumberGenerator: {}
              }
```

# FHiCL structure: source

- **Services is where you put all of the simulation specific services for what you're trying to run**
  - This can mean things like:
    - Detector geometry
    - Physical properties
    - File management

```
services: {
  @table::dunefd_1x2x6_simulation_services
   TFileService: { fileName: “my_dank_file_name.root” }
   RandomNumberGenerator: {}
                  }
```

Naming the output root file

# FHiCL structure: source

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

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

# FHiCL structure: source

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

This means we're starting with an empty event. We can also specify that we're reading from ROOT file with ROOTInput

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

# FHiCL structure: source

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

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

Default number of events to generate and default run and event number

# Why use FHiCL files?

- **This is where we configure the modules that actually do something on the event**

```
physics: {

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

# FHiCL structure: physics

- **This is where we configure the modules that actually do something on the event**

Add information to the ROOT file

```
physics: {

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

# FHiCL structure: physics

- **This is where we configure the modules that actually do something on the event**

```
physics: {

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

← Perform analysis on the ROOT file. Notice the "z" because, you know, Americans

# FHiCL structure: physics

- **This is where we configure the modules that actually do something on the event**

```
physics: {

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

Remove events we don't want (← filters)

# FHiCL structure: physics

- **This is where we configure the modules that actually do something on the event**

```
physics: {

  producers: {
    rns: {module type: “RandomNumberSaver”}
    generator: @local::dunefd_singlep
  }
  analyzers: {}
  filters: {}
  simulate: [rns, generator]
  stream1: [out1]
  trigger_paths: [simulate]
  end_paths: [stream1]
}
```

Define the order you want to run things

# FHiCL structure: physics

- **This is where we configure the modules that actually do something on the event**

```
physics: {

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

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

# FHiCL structure: physics

- **This is where we configure the modules that actually do something on the event**

```
physics: {

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

Everything that modifies the event

# FHiCL structure: physics

- **This is where we configure the modules that actually do something on the event**

```
physics: {

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

Everything that doesn't modify the event, such as analysers and output streams

# FHiCL structure: outputs

- **Finally, we define where the output goes**

```
outputs: {
  out1: {
    module_type: RootOutput
    fileName: "%ifb_ana.root"
  }
}
```

Take the file name you started with "my_file.root" and return a file called "my_file_ana.root".

Another good option is to use "my_file _%p-%tc.root". Try it and see what it does

# Configuring fhicls

- **Most of the time you'll want to make small changes without having to re-write all of the configurations**

- **You can override a parameter after you define them**

```
physics: {
  producers: {
    rns: {module type: "RandomNumberSaver"}
    generator: @local::dunefd_singlep
  }
}

# Set some parameters for the generator
physics.producers.generator.PDG: [2112] # generate a neutron
physics.producers.generator.P0: [0.5]   # give it 500 MeV
```

# How to find the configurable parameters?

- **You start with a FHiCL file like this**

```
#include “singles_dune.fcl”

physics: {
 producers : {
     generator: @local::dunefd_singlep
   }
}
```

- **The generator is being sourced from the included file… so look in there**

- **Remember that findfcl.sh script!**

# Why use FHiCL files?

- **Look in the first file**

```
./findfcl.sh singles_dune.fcl

Found fhicl file(s):
/long/path/to/singles_dune.fcl
```

- **See what we find**

# Why use FHiCL files?

```
File Edit Options Buffers Tools Help
#include "singles.fcl"

BEGIN_PROLOG


################
###### FD ######
################

dunefd_singlep: @local::standard_singlep
dunefd_singlep.Theta0YZ:                [ 0.0 ]     # beam is along the z axis
dunefd_singlep.Theta0XZ:                [ 0.0 ]     # beam is along the z axis
dunefd_singlep.P0:                      [ 6. ]

# Start it in the first TPC, first cryostat
dunefd_singlep.X0:                      [ -1474. ]
dunefd_singlep.Y0:                      [ -351. ]
dunefd_singlep.Z0:                      [ 0. ]



#################
###### 35t ######
#################

dune35t_singlep: @local::standard_singlep
-UU-:----F1  singles_dune.fcl    Top L1     (Fundamental) ----------------------------------------------------------------
For information about GNU Emacs and the GNU system, type C-h C-a.
```

- **This isn't exactly what we're looking for, but there is another file included at the top**

# Why use FHiCL files?

```
File Edit Options Buffers Tools Help
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
-UU-:%%--F1  singles.fcl     Top L1     (Fundamental) ----------------------------------------------------------------
Note: file is write protected
```

- **Now we’ve found all of the different configurable parameters**
- **We got there by looking through all of the files included (which is something you’re going to do a lot of)**

# Event generators

- **There are a few generators used in larsoft simulations, all for different purposes**

- **The simplest one is the single particle gun, literally fires off one particle at a time**

- **Some more fancy ones are:**
  - GENIE: for generating neutrinos
  - CORSIKA: for cosmic rays
  - MARLEY: for supernova and solar neutrinos
  - People doing BSM usually write their own generators or modify GENIE

# Single Particle Gun

- **We're going to solely focus on the single particle gun**
- **This generates a particle (an sims::MCParticle if you wanna be fancy) with some initial parameters:**
  - Start position (x, y, z)
  - Start momentum (px, py, pz)
  - PDG code
  - Energy range
  - Etc

# Geant4

- **GEANT4 is responsible for propagating particles around a geometry (and is also the second laziest acronym to come from CERN)**

- **GEANT4 simulates all the physical processes that go on in the detector**
    - Interaction with argon
    - Ionisation
    - Showers
    - Decays

# Detector simulation

- **Finally there's detector simulation which handles what the APA planes will see when charge passes by the wires and light hits the photon detectors**

- **There's also reconstruction, but we'll worry about that later**

- **DETSIM produces raw::RawDigit objects which tell you**
    - Which APA you're on
    - The channel number and ADC waveform of every wire in the detector over a given time window

# The lar command

- **To run a FHiCL file you need to get comfortable with lar commands**

- **There are a lot of flags you can pass into a lar command but the important ones are:**
  - -c, —config, the fhicl file you're running
  - -s, —source, the source file (a ROOT file made be some previous stage)
  - -n, —evts, the number of events to run
  - -o, —output, overriding the name of the outputted file
  - -k, —nskip, the number of events to skip

- **A typical lar command would look like this**

```
lar -c run_geant4.fcl -s some_particles.root -n -1
```

This means run over every event possible

# Running the event display

- **LArSoft has an event display that you can use to view your events and make sure things are going how you expect**
- **There are lots of features, however it can be quite slow. If you have a VNC working it speeds things up a lot**
- **To run it use**

```
lar -c evd_dunefd.fcl your_detsim_file.root
```

# Time to generate your first events!!

(Let’s ignore the ones from the previous tutorial)

# Main task

- **You have a file "prod_particle_template.fcl"**
- **Fill out the required fields with information from the slides and made sure you give your output file name something interesting**
- **Generate 10 events with 1 muon and 1 proton with the following requirements:**
  - Start position of both particles (-100, 0, 150)
  - Muon:
    - momentum: 700 MeV
    - theta_xz: -10 degrees
    - theta_yz: 0 degrees
  - Proton:
    - momentum 800 MeV
    - theta_xz: 35 degrees
    - theta_yz: 10 degrees
- **Run GEANT4 over the produced particle file**
- **Run DETSIM over the GEANT4 file** `(standard_g4_dune10kt_1x2x6.fcl)`
- **Run the event display over your DETSIM file and see what you've got** `(standard_detsim_dune10kt_1x2x6.fcl)`
- **Repeat everything above, but add some gaussian variation to the angles**

# Bonus task

- **Generate 10 muon proton events like before, but add 5 additional muons distributed randomly throughout the detector to mimic cosmic rays**
- **Check it out in the event display and see what a neutrino event might look like**

# Writing your own fcl file

- **The generation fcl is practically empty**

- **Make sure you have all the necessary includes at the top of your file. If you have something like**

```
services: {
 @table::dunefd_1x2x6_simulation_services
}
```

**You need the right fcl at the top of your file, otherwise larsoft won't find it!**

# Writing your own fcl file

- **If you're running a module such as SingleGen, you'll need to specify all the required fcl parameters needed. Not just what you want**

- **For example, SingleGen required SigmaP (the breadth of the energy range) to be set. If you don't need it set it to a default value**

```
physics.producers.generator.SigmaP: [0.0]
```

- **To find out what parameters are required you can:**
  - Look through other fcl files that use the module
  - Read the documentation
  - Use the ART missing parameter error message

# A word on text editors

- **Using emacs:**
    - Open a file by doing emacs -nw my_file.fcl
    - Once you're done save using crtl+x ctrl+s
    - Exit using ctrl+x ctrl+c
    - This doesn't seem to be available when connecting through ssh but does work in the web client

- **Using vim:**
    - Open a file using vi my_file.fcl
    - Attempt to type by first pressing I to go into insert mode
    - Try saving and quitting by pressing escape, then entering :wq
    - If you have problems ask Dom or anyone else crazy enough to use vim, then listen to the lecture trying to rationalise their use of vim

- **Using nano, pico or any other terminal editor**
    - Why? Just use emacs

# Plotting the angular distribution

- **A directory called PlottingScripts is available to you**
- **Go into PlottingScripts/build and run the following**
  - cmake ../
  - make
- **In PlottingScripts/Analyzer/PlottingScript.cxx, fill out the blank parts to make a histogram of the angle between the muon and the proton**
- **Remember to compile after you've made any changes by going into PlottingScrips/build and running the make command**
- **To run the plotting script go into the build directory and run the following**

```
./Analyzer/PlottingScript -i /path/to/your/file_ana.root -t tree/name -o output_name -n <number of events>
```

**the output name does not need a file extension, a pdf will be produced**

- **If you don't like using cmake feel free to write your own macro to do this**