Upgrading to ldmx-sw 2.0

Tom Eichlersmith

University of Minnesota

eichl008@umn.edu

May 5, 2020

Outline

1. Big Changes
2. General Suggestions
3. Basics
4. Analysis
5. Advanced Simulation
6. Homework

Big Changes

- Migrate containers to STL containers (instead of TClonesArray)
- Integrate simulation into `ldmx-app` pipeline
- Harness more power from python-nature of configuration files
- Introduction of unit testing framework
- New Dark Brem simulation method (use LHE vertex in Geant4 process)
- Standardize v12 detector

What this means for you...

- **Can run processors outside of ldmx-sw** (checkout ldmx-analysis repository![1])
- Encouraging people to *only* edit ldmx-sw for developing activities
- You will need to update/rewrite any scripts using `ldmx-sim` and maybe some scripts using `ldmx-app`
- Re-learn how to write python configs ⇒ interaction with integrated simulation prompted us to update how the python configuration operates

[1] `https://github.com/LDMX-Software/ldmx-analysis`

What this means for you...

- **Can run processors outside of ldmx-sw** (checkout ldmx-analysis repository![1])
- Encouraging people to *only* edit ldmx-sw for developing activities
- You will need to update/rewrite any scripts using `ldmx-sim` and maybe some scripts using `ldmx-app`
- Re-learn how to write python configs ⇒ interaction with integrated simulation prompted us to update how the python configuration operates

Keep Refreshing the Wiki

Omar and I have been focusing more on keeping the wiki up-to-date. Please familiarize yourself with it; it should be the first place you go to check for information.

[1] `https://github.com/LDMX-Software/ldmx-analysis`

ldmx-sw 101

(let's pretend you are a new student starting on LDMX)

Assumptions

- Built and installed version of ldmx-sw 2.0
  - Building changes on a computer-by-computer basis, so not going to go into that here
  - Check out "Installing ldmx-sw" wiki for detailed instructions and reach out to me with your computer details if you get stuck
- Somewhat familiar with python
- Understand the goals of LDMX and have a general idea of the experimental setup

Big Idea

Simulate/Reconstruct/Analyze **event-by-event**

- Particles/Hits/Objects are put onto *event bus* (so they are *passengers*)
- *Processors* look at one event bus at a time in a certain *sequence*
- Generally, processors take passengers off the bus to *analyze* and/or *produce* passengers to put onto the bus
- *Process* knows the sequence and feeds the event buses to the processors

Process

- The pass name
- Input and output files
- Histogram file
- Maximum number of events
- Logging frequency

```
from LDMX.Framework import ldmxcfg
# create my process object
p = ldmxcfg.Process( "sim" )
# how many events to process?
p.maxEvents = 10
# we want to see every event
p.logFrequency = 1
# we also only have an output file
p.outputFiles = [
        "myFirstSim_"
        + str(p.maxEvents)
        + "_events.root"
        ]
```

Simulation

Required

- Detector description
- At least one primary generator

Optional

- Run Number
- Run Description
- Biasing operators
- Scoring plane description
- Filtering (or other User Actions)

Minimum Simulation

```
# link necessary libraries
p.libraries.append( "libSimApplication.so" )
mySim = ldmxcfg.Producer( "mySim", #run-unique name
        "ldmx::Simulator")
# get the path to the installed detector description
from LDMX.Detectors.makePath import makeDetectorPath
mySim.parameters["detector"] = makeDetectorPath( "ldmx-det-v12" )
# Set a run number
mySim.parameters["runNumber"] = 9000
# Get a pre-written generator
from LDMX.SimApplication import generators as gen
mySim.parameters["generators"] = [ gen.single_4gev_e_upstream_tagger() ]
# add your configured simulation to the sequence
p.sequence.append( mySim )
# can still edit parameters after adding to sequence
mySim.parameters["verbosity"] = 1
```

Reconstruction

```
# include necessary libraries
p.libraries.extend([
    "libEcal.so", "libEventProc.so"
    ])
# import all the templates
from LDMX.Ecal.ecalDigis import ecalDigis
from LDMX.Ecal.ecalRecon import ecalRecon
from LDMX.EventProc.hcalDigis import hcalDigis
from LDMX.EventProc.trigScintDigis import *
# add them to the sequence
p.sequence.extend([
    ecalDigis, ecalRecon,
    hcalDigis,
    trigScintDigis, trigScintDigisDn, trigScintDigisTag
    ])
```

Let's Run It

Analysis

...Up to You!

Lots of examples in `DQM` module and `ldmx-analysis` repo.

Features to Look At:

- `HistogramPool`: Manage histograms by name instead of managing histogram pointers yourself
  - Helpful for creating *lots* of histograms
  - `EcalPN` analyzer in `DQM` module
- `NtupleManager`: Create tree of *once per event* values in histogram file
  - Helpful for higher-level physics variables when you don't know which you want to compare yet
  - `DummyAnalyzer` in `Framework` module
- `setStorageHint`: Skim events based on criteria in your processor
  - Helpful for picking out specific (maybe frustrating?) events to look at in more detail
  - `TriggerProcessor` in `EventProc` module
- `Process::keep`: drop or keep event objects to save space in your output files
  - Can be done only in python configuration

Starting an Analysis

General Outline

- Clone the `ldmx-analysis` repo and make your own branch (e.g. *myusername-dev*)
- Use the `ldmx-sw-scripts/utils/NewProcessory.py` script to make a template analyzer.

```
./NewProcessor.py --incDir include --srcDir src MyAnalysis
```

- `MyAnalysis.h` is in `include` and `MyAnalysis.cxx` is in `src`.
- Start make histograms/ntuplizing/skimming!

To Build (in README of `ldmx-analysis`)

```
mkdir build; cd build; #make a build directory
cmake -DCMAKE_INSTALL_PREFIX=../install \
    -DLDMX_INSTALL_PREFIX=<path-to-ldmx-sw-install> ..
make install
```

Analyzing

```
from LDMX.Framework import ldmxcfg
# create my process object
p = ldmxcfg.Process( "ana" )
# we want to see every event
p.logFrequency = 1
# include necessary libraries
p.libraries = [ "<path-to-ldmx-analysis-install>/lib/libAnalysis.so" ]
# add them to the sequence
myAna = ldmxcfg.Analyzer( "myAna" , "ldmx::MyAnalysis" )
# maybe configure analyzer?
# add analyzer to sequence
p.sequence = [ myAna ]
#input files to analyze
p.inputFiles = [ sys.argv[1] ]
```

Common Pitfalls in Config Files

- Not providing the required libraries
  - Get a "Unable to create" error
  - Link ldmx-app with the libraries your processors are in with `p.libraries`
  - Will need full path if library is not in ldmx-sw
- Misspelling parameter names
  - ldmx-sw gives each parameter the compiler minimum as a default, so the process will run if a parameter is not given
  - Misspelling a parameter means that your parameter value is just silently ignored
- Obviously lots more...
  - Many quirks of ldmx-sw that we all need to learn
  - Please put questions in slack and/or reach out to me
  - If you can imagine a C++ solution, put your ideas in the GitHub issue tracker

Advanced Simulation

(as advanced as you can get without touching C++)

Documentation on Configuring the Simulation:

`https://github.com/LDMX-Software/ldmx-sw/wiki/Configuring-the-Simulation`[2]

[2] An attempt at gathering all available simulation parameters into one location. A good stopping point to check before being forced to look at the source code.

Generator

- Implemented as `PrimaryGenerators`
- Can have several running at once
- Multi-Particle Gun Example:

```
from LDMX.SimApplication import generators as gen
myMPG = gen.multi("myMPG")
myMPG.parameters["nParticles"] = 1
myMPG.parameters["enablePoisson"] = True
myMPG.parameters["pdgID"] = 11
myMPG.parameters["vertex"] = [ 0., 0., 0. ] #mm
myMPG.parameters["momentum"] = [ 0., 0., 4000. ] #MeV
# add to simulator defined earlier
mySim.parameters["generators"] = [ myMPG ]
#smear primary vertex from simulator
mySim.parameters["beamSpotSmear"] = [ 80., 20., 0. ] #mm
```

Biasing

- Implemented as Geant4 `XsecBiasingOperators`
- Only one operator at a time
- PN Example:

```
# Enable and configure the biasing
mySim.parameters['biasing.enabled'] = True
mySim.parameters['biasing.particle'] = 'gamma'
mySim.parameters['biasing.process'] = 'photonNuclear'
mySim.parameters['biasing.volume'] = 'target'
mySim.parameters['biasing.threshold'] = 2500.
mySim.parameters['biasing.factor'] = 100000
```

Filters

- Implemented as `UserActions`
- Used mainly for event filtering and track filtering
  - Event Filtering: Abort the sim event early to save time if criteria aren't met
  - Track Filtering: Make sure sim saves a specific particle information to output file
- Check out python parts of `Biasing` module for examples
- Filtering Example:

```
# Most filters defined in Biasing module
p.libraries.append( "libBiasing.so" )
from LDMX.Biasing import filters
# add to simulator defined earlier
mySim.parameters["actions"] = [
        filters.tagger_veto_filter(),
        # Only consider events where a hard brem occurs
        filters.target_brem_filter(),
        # Tag all photo-nuclear tracks to persist them to the event.
        filters.pn_track_filter()
        ]
```

Helpful Tips

- Never type out file paths or file names. Input files can be fed on the command line and output files can be formatted based off of input names and parameters.
- `os.path.realpath` to get full path without symlinks to files
- `argparse` to configure your run by changing processors or passing input files or changing parameters
- `print` ldmx classes to see what parameters are given to it after your python nonsense *but before they have been taken for processing*
- Define python functions (maybe even write your own python module) containing settings/processors/sequences/processes that you use a lot

Helpful Tips

- Never type out file paths or file names. Input files can be fed on the command line and output files can be formatted based off of input names and parameters.
- `os.path.realpath` to get full path without symlinks to files
- `argparse` to configure your run by changing processors or passing input files or changing parameters
- `print` ldmx classes to see what parameters are given to it after your python nonsense *but before they have been taken for processing*
- Define python functions (maybe even write your own python module) containing settings/processors/sequences/processes that you use a lot

Overall

Remember that the configuration script is run as a python script *before* the parameters are taken for processing, so you can automate a lot of stuff.

All Together Now

(How complicated can you get? Answer: pretty complicated)

Let's look at my workflow for the ECal as Target (EaT) project.

Homework

Feel free to ask your friendly-neighborhood tutor Tom for help!

Exercise 1 (Simulation)

Generate 1000 events where the number of primary electrons is poisson distributed around an average of 2. These events should have the reconstruction done as well. *Bonus Points*: Can you drop the "Sim" collections but keep the "SimParticles"?

Exercise 2 (Analysis)

Using your events from Exercise 1, make a histogram comparing the total reconstructed energy in the ECal to the number of primary electrons that the event started with. *Hint*: You will need to use both the "EcalRecHits" collection and the "SimParticles" map. *Bonus Points*: Can you skim the events for the ones with more than 2 primary electrons?

Backup

HistogramPool

- Exemplified by `EcalPN` analyzer in `DQM` module
- `get`: Get a histogram by name
- `create`: Make a new histogram by name and type
- You can tell the HistogramPool what kinds of 1D histograms to create in your python configuration by using `build1DHistogram` member function of a processor
  - Then you don't need to `create` those histograms in the Analyzer code

NtupleManager

- Only needs to be done in Analyzer code
- `create`: Make a new tree to put variables in
- `addVar`: Put a new variable name into a tree (template)
- `setVar`: Set the variable value during processing

Skimming Events

After using `setStorageHint` in your processor, you also need to tell the process that it should listen to your processor.
In your python configuration file:

```
myAna = ldmxcfg.Analyzer( "myAna" , "ldmx::MyAnalzer" )
# p is Process object created earlier
p.skimDefaultIsDrop() # drop events unless told otherwise
p.skimConsider( "myAna" ) # tell Process to listen to myAna
```

Drop/Keep Event Objects

Maybe you know *for a fact* that you will *never* use the simulated hits later in your analysis chain and you want to save some space in your output file.
In your python configuration file:

```
# p is Process object created earlier
p.keep = [
    # don't put any collection with 'EcalSim' in the name into output file
    "drop EcalSim.*",
    ]
```