pyROOT for LCIO

Christian Grefe

CERN PH-LCD

5. December 2012

Outline

Motivation

- ROOT dictionaries recently introduced to LCIO
- python very convenient for high level coding: plotting, analysis, etc.
- pyROOT gives access to all ROOT classes including user classes with ROOT dictionaries
- Provide user friendly python package that allows convenient analysis of LCIO data
- Add functionality to avoid clunky code, e.g. use of c-style arrays

Requirements

- ROOT compiled with python, e.g. CERN AFS installation
  `/afs/cern.ch/sw/lcg/app/releases/ROOT/...`
- LCIO v02 compiled with `BUILD_ROOTDICT=ON`, e.g. CERN LCD software
  `/afs/cern.ch/eng/clic/software/lcio/v02-01-02/`
- pyLCIO package from CERN LCD SVN (world readable with cern account)
  `svn+ssh://svn.cern.ch/reps/clicdet/trunk/PyRootLcio`
- Environment variables
  - Add path of pyLCIO to `$PYTHONPATH`
  - Set `$LCIO` to LCIO installation directory to allow automatic loading of ROOT dictionaries
- Full installation on CERN AFS
  `source /afs/cern.ch/eng/clic/software/setupSLC6.sh`

Missing ROOT Dictionaries

- Some of the LCIO classes that are used are currently not part of the default ROOT dictionaries
- Need to add some includes before building LCIO
  `${LCIO}/src/cpp/include/rootDict/rootio_templates.h`

```
#include "EVENT/LCIO.h"
#include "UTIL/LCStdHepRdr.h"
#include "UTIL/LCTOOLS.h"
```

Decoration of LCIO Classes

- Avoid use of c-style arrays, e.g. `double[]`
- Use python introspection to modify classes at runtime
- Add `getVariableVec()` that returns `TVector3` for all methods that return `double[3]`, e.g. `getPositionVec()` for all classes with `getPosition()`
- Add `getLorentzVec()` that returns `TLorentzVector` to all classes that support `getMomentum()` and `getEnergy()`

Ensure python Style Code

- `IOIMPL::LCCollectionIOVec` does not allow python loops:
  "maximum recursion depth exceeded while calling a Python object"
- These are all collections read from an existing LCIO files, i.e. `event.getCollection( collectionName )`
- Provide a wrapper class that inherits from python list and fulfills the `EVENT::LCCollection` interface

```
from pyLCIO.base.LCCollection import LCCollection

collection = event.getCollection( collectionName )
fixedCollection = LCCollection( collection )

for item in fixedCollection:
    print item
```

- `EVENT::LCEvent.getCollectionVec( collectionName )` returns the wrapper class instead

Managed Event Loop and Drivers/Processors

- Provide a managed event loop similar to Marlin/`org.lcsim`
- Plug in user classes that are executet for each event
- Support (at the moment) LCIO and StdHep input
- File type handled by event loop: `eventLoop.setFile( fileName )` independent of file type (determined by file extension)
- User class must inherit from `Driver` or implement: `startOfData()`, `process( event )`, `endOfData()`

Example Driver

```
from pyLCIO.drivers.Driver import Driver
from ROOT import TH1D, TCanvas

class McParticlePlotDriver( Driver ):
    def __init__( self ):
        Driver.__init__( self )
        self.histograms = {}

    def startOfData( self ):
        # Create two histograms for energy and pT
        self.histograms['Energy'] = TH1D( 'Energy', 'Energy;Energy [GeV];Entries', 100, 0., 1000. )
        self.histograms['Pt'] = TH1D( 'Pt', 'pT;p_T [GeV];Entries', 100, 0., 1000. )

    def process( self, event ):
        # Get the MCParticle collection from the event
        mcParticles = event.getMcParticles()
        # Loop over all MCParticles
        for mcParticle in mcParticles:
           # Get the four vector of the MCParticle
            v = mcParticle.getLorentzVec()
            # Fill the histograms
            self.histograms['Energy'].Fill( v.Energy() )
            self.histograms['Pt'].Fill( v.Pt() )

    def endOfData( self ):
        # Create a canvas for each histogram and draw them
        plots = []
        for histogramName in self.histograms:
            plot = TCanvas( 'c%s' % ( histogramName ), histogramName )
            self.histograms[histogramName].Draw()
            plots.append( plot )
        raw_input( 'Press any key to continue ...' )
```

Example Executable

```
import sys, os
from pyLCIO.base.EventLoop import EventLoop
from pyLCIO.drivers.EventMarkerDriver import EventMarkerDriver
from pyLCIO.drivers.McParticlePlotDriver import McParticlePlotDriver
from pyLCIO.drivers.WriteLcioDriver import WriteLcioDriver

def testReader( fileName ):
    # Create a new event loop
    eventLoop = EventLoop()
    # Set the input file
    eventLoop.setFile( fileName )
    # Create a new driver, configure it and add it to the loop
    marker = EventMarkerDriver()
    marker.setInterval( 10 )
    eventLoop.add( marker )
    # Add another driver to the loop
    mcParticle = McParticlePlotDriver()
    eventLoop.add( mcParticle )
    # Add another driver to the loop
    writer = WriteLcioDriver()
    writer.setOutputFileName( 'test.slcio' )
    eventLoop.add( writer )
    # Skip some events if desired
    eventLoop.skipEvents( 200 )
    # Execute the event loop
    eventLoop.loop( 100 )

if __name__ == "__main__":
    if len( sys.argv ) < 2:
        sys.exit( 2 )
    testReader( sys.argv[1] )
```

Conclusion

- pyROOT works nicely out of the box with ROOT LCIO dictionaries
- pyLCIO package
  - Automatic loading of dictionaries
  - Decorate LCIO classes to return `TVector3` and `TLorentzVector` where appropriate
  - Wrap LCCollection to allow *pythonic* loops
  - Streamlined user interface for LCIO and StdHep readers
  - Managed event loop with driver/processor style plug-in of user code
- Aim is to allow easy plotting, high level analysis code and creation of tuples/trees
- Can not replace reconstruction algorithms in Marlin/`org.lcsim`
- No geometry information (except raw cell IDs stored with hits)

Let me know if you need help or have suggestions for improvements!