pyLCIO - Python bindings for LCIO

Christian Grefe

CERN

28. May 2013

ROOT Dictionaries for LCIO

- ROOT dictionaries available for LCIO since v2
- Compile LCIO with `BUILD_ROOTDICT=ON`

ROOT Dictionaries for LCIO

- ROOT dictionaries available for LCIO since v2
- Compile LCIO with `BUILD_ROOTDICT=ON`
- Start interactive root session, load the dictionaries and use LCIO classes

```
gSystem->Load("liblcio.so");
gSystem->Load("liblcioDict.so");
IO::LCReader* reader = IOIMPL::LCFactory::getInstance().createLCReader
reader->open("test.slcio");
EVENT::LCEvent* event = reader->readNextEvent();
while (event) {
    std::cout << event->getRunNumber() << std::endl;
    event = reader->readNextEvent();
}
reader->close();
```

ROOT Dictionaries for LCIO

- ROOT dictionaries available for LCIO since v2
- Compile LCIO with `BUILD_ROOTDICT=ON`
- Start interactive root session, load the dictionaries and use LCIO classes

```
gSystem->Load("liblcio.so");
gSystem->Load("liblcioDict.so");
IO::LCReader* reader = IOIMPL::LCFactory::getInstance().createLCReader
reader->open("test.slcio");
EVENT::LCEvent* event = reader->readNextEvent();
while (event) {
    std::cout << event->getRunNumber() << std::endl;
    event = reader->readNextEvent();
}
reader->close();
```

Use a real scripting language instead!

Using LCIO with pyROOT

- If ROOT is installed with python we get the LCIO python bindings for free!
- Compile LCIO with `BUILD_ROOTDICT=ON`

Using LCIO with pyROOT

- If ROOT is installed with python we get the LCIO python bindings for free!
- Compile LCIO with `BUILD_ROOTDICT=ON`
- Start interactive python session, import ROOT, load the dictionaries and use LCIO classes

```
from ROOT import gSystem
gSystem.Load("liblcio.so")
gSystem.Load("liblcioDict.so")
from ROOT import IOIMPL
reader = IOIMPL.LCFactory.getInstance().createLCReader()
reader.open("test.slcio")
event = reader.readNextEvent()
while event:
    print event.getEventNumber()
    event = reader.readNextEvent()
reader.close()
```

Automatic Loading of Dictionaries

- `pyLCIO` package adds automatic loading of dictionaries on import
- Transparent import of the LCIO namespaces (identical to import from ROOT)

Automatic Loading of Dictionaries

- pyLCIO package adds automatic loading of dictionaries on import
- Transparent import of the LCIO namespaces (identical to import from ROOT)

```
from pyLCIO import IOIMPL
reader = IOIMPL.LCFactory.getInstance().createLCReader()
reader.open("test.slcio")
event = reader.readNextEvent()
while event:
    print event.getEventNumber()
    event = reader.readNextEvent()
reader.close()
```

Making LCIO Objects Iterable

- Add proper `__iter__()` method to relevant LCIO classes on import

Making LCIO Objects Iterable

- Add proper `__iter__()` method to relevant LCIO classes on import

```
from pyLCIO import IOIMPL
reader = IOIMPL.LCFactory.getInstance().createLCReader()
reader.open("test.slcio")
for event in reader:
    print event.getEventNumber()
reader.close()
```

Making LCIO Objects Iterable

- Add proper `__iter__()` method to relevant LCIO classes on import

```
from pyLCIO import IOIMPL
reader = IOIMPL.LCFactory.getInstance().createLCReader()
reader.open("test.slcio")
for event in reader:
    print event.getEventNumber()
reader.close()
```

- LCEvent acts like a list of tuples

```
for collectionName, collection in event:
    print collectionName, collection.getNumberOfElements()
```

Making LCIO Objects Iterable

- Add proper `__iter__()` method to relevant LCIO classes on import

```
from pyLCIO import IOIMPL
reader = IOIMPL.LCFactory.getInstance().createLCReader()
reader.open("test.slcio")
for event in reader:
    print event.getEventNumber()
reader.close()
```

- LCEvent acts like a list of tuples

```
for collectionName, collection in event:
    print collectionName, collection.getNumberOfElements()
```

- LCCollection acts like a list

```
for element in collection:
    print element
```

Enhanced Object Interfaces

- Avoid use of c-style arrays, e.g. `double[]`
- Decorate LCIO classes automatically on import depending on existing methods
- Add `getVariableVec()` that returns `TVector3` for all methods that return `double[3]`, e.g. `getPositionVec()` for all classes with `getPosition()`
- Similarly add `setVariableVec(TVector3)` to all IMPL classes
- Add `getLorentzVec()` that returns `TLorentzVector` to all classes that support `getMomentum()` and `getEnergy()`

Reading LCIO and StdHep Files

- IO::LCReader and UTIL::LCStdHepRdr now fulfill iterable interface

Reading LCIO and StdHep Files

- IO::LCReader and UTIL::LCStdHepRdr now fulfill iterable interface
- Offer two wrapper classes that streamline the interface of both readers
- Allow transparent loop over all input files

```
from pyLCIO.io.StdHepReader import StdHepReader
reader = StdHepReader("test.stdhep")
reader.addFile("test2.stdhep")
reader.addFiles(["test3.stdhep", "test4.stdhep"])
reader.addFileList("stdhepFiles.txt")
reader.skip( 10 )
for event in reader:
    print event.getEventNumber()
reader.close()
```

Event Loop and Analysis

- Provide a managed event loop similar to Marlin/`org.lcsim`
- Plug in user classes that are executed for each event
- Support LCIO and StdHep input using the new reader interface
- 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()`, `processEvent( LCEvent )`, `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 ):
        self.histograms['Energy'] = TH1D( 'Energy', 'Energy;Energy [GeV];Entries', 50, 0., 260. )
        self.histograms['Pt'] = TH1D( 'Pt', 'pT;p_T [GeV];Entries', 50, 0., 100. )
        self.histograms['PDGID'] = TH1D( 'PDGID', 'PDG ID;PDG ID;Entries', 1200, -600, 600. )
        self.histograms['GeneratorStatus'] = TH1D( 'GeneratorStatus', 'Generator Status;Generator Status;Entries

    def processEvent( self, event ):
        mcParticles = event.getMcParticles()
        for mcParticle in mcParticles:
            v = mcParticle.getLorentzVec()
            self.histograms['Energy'].Fill( v.Energy() )
            self.histograms['Pt'].Fill( v.Pt() )
            self.histograms['PDGID'].Fill( mcParticle.getPDG() )
            self.histograms['GeneratorStatus'].Fill( mcParticle.getGeneratorStatus() )

    def endOfData( self ):
        plots = []
        for histogramName in self.histograms:
            plot = TCanvas( 'c%s' % ( histogramName ), histogramName )
            self.histograms[histogramName].Draw()
            plots.append( plot )
        userInput = raw_input( 'Press any key to continue' )
```

$LCIO/examples/python/exampleDrivers/McParticlePlotDriver.py

Example Executable

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

def McParticlePlots( fileName ):
    eventLoop = EventLoop()
    # Set the input file. The actual reader is determined from the file ending (stdhep or slcio)
    eventLoop.setFile( fileName )
    # Add a driver to print the progress
    markerDriver = EventMarkerDriver()
    markerDriver.setInterval( 1 )
    markerDriver.setShowRunNumber( False )
    eventLoop.add( markerDriver )
    # Add the driver that draws the MCParticle plots
    mcParticlePlotDriver = McParticlePlotDriver()
    eventLoop.add( mcParticlePlotDriver )
    # Skip some events if desired
    eventLoop.skipEvents( 0 )
    # Execute the event loop
    eventLoop.loop( -1 )

def usage():
    print 'Usage:\n  python %s <fileName>' % ( os.path.split( sys.argv[0] )[1] )

if __name__ == "__main__":
    if len( sys.argv ) < 2:
        usage()
        sys.exit( 0 )
    # Read the file name from the command line input
    fileName = sys.argv[1]
    McParticlePlots( fileName )
```

$LCIO/examples/python/McParticlePlots.py

XML Steering of Drivers (Experimental)

- Provide executable that parses an XML file and sets up the event loop
- At the moment very limited features - can be expanded if there is demand
- Run it: `python $LCIO/src/python/pylcio steering.xml`

```
<pylcio>
    <inputFiles>
        <file> test.slcio </file>
    </inputFiles>

    <control>
        <skipEvents>0</skipEvents>
        <numberOfEvents>-1</numberOfEvents>
        <printDrivers>True</printDrivers>
        <printStatistics>true</printStatistics>
    </control>

    <execute>
        <driver name="markerDriver"/>
        <driver name="mcParticlePlotDriver"/>
    </execute>

    <drivers>
        <driver name="markerDriver" type="pyLCIO.drivers.EventMarkerDriver.EventMarkerDriver">
            <interval> 1 </interval>
            <showRunNumber> False </showRunNumber>
        </driver>
        <driver name="mcParticlePlotDriver" type="exampleDrivers.McParticlePlotDriver.McParticlePlotDriver"/>
    </drivers>
</pylcio>
```

`$LCIO/examples/python/exampleSteering/McParticlePlots.xml`

Summary

- Python bindings work out of the box with ROOT LCIO dictionaries through pyROOT
- pyLCIO package for additional features
  - Automatic loading of ROOT and LCIO dictionaries on import
  - Added iterator methods to container and reader classes to allow *pythonic* loops
  - Additional accessor methods for LCIO classes to directly get `TVector3` and `TLorentzVector` where appropriate
  - Wrapper classes for LCReader and LCStdHepRdr to streamline interface
  - Managed event loop with driver/processor style plug-in of user code
  - XML steering of drivers (experimental)
- Use this for high level tasks like analysis, plotting and creation of tuples/trees
- Can not replace complex reconstruction algorithms in Marlin/`org.lcsim`
- (Currently) no geometry information except raw cell IDs stored with hits

Requirements

- Install ROOT with python bindings
- Get LCIO version v02-04 or newer (included in ILCSoft v01-17-01 - released last week)
- Compile with `BUILD_ROOTDICT=ON`
- Add ROOT and pyLCIO to python environment
  `export PYTHONPATH=$LCIO/src/python:$PYTHONPATH`
- Or simply source `$LCIO/setup.sh`

pyROOT Tips

- Access to template objects (need to be explicitly declared in the ROOT dictionary)

```
# create an empty std::vector<std::string>
v = ROOT.vector( "string" )()

# create an ID encoder for SimTrackerHits
idEncoder = UTIL.CellIDEncoder( IMPL.SimTrackerHitImpl )( encodingString, trackerHits )
```

- Passing basic types by reference

```
x = ROOT.Double( 0.0 )
y = ROOT.Double( 0.0 )
graph = ROOT.TGraph( 1 )
graph.setPoint( 0, 3, 2 )
graph.getPoint( 0, x, y )
print x, y
```