LHCb Note 2006-014
TRACKING/SOFTWARE

Tracking in Python

E. Rodrigues

NIKHEF, Amsterdam, The Netherlands

Abstract

This note describes the packages, modules and tools that have been developed to make it possible to access in Python the tracking classes, tools and association tables. Examples are given of how to use the existing framework, and how to develop more complicated user tools and algorithms.

Contents

1 Introduction

In the past couple of years Python has become ever more popular among LHCb. On the one hand, there has been a constant and steady improvement in the "Python-ized" version of Gaudi – GaudiPython [1]; and sophisticated applications such as Bender [2] have seen the light. On the other hand the number of so-called "Python enthusiasts" has also increased considerably.

As far as the LHCb tracking code is concerned, an effort was put into making all the Track Event Model classes and the most useful tracking tools accessible in Python. This note describes what has been developed aiming at this goal.

For completeness, we briefly start by presenting all the packages presently available to deal with the tracking in Python. The remaining sections will subsequently show, with examples, how to access and use the tracking classes, tools and linkers (association) tables.

At the end some examples are given on how to exploit the existing to fabricate more complicated user tools and algorithms.

All examples in this note require at least the following versions of the LHCb software: Gaudi v18r3, GaudiPython v8r4, LHCb v20r4, Kernel/LHCbKernel v6r5, Tr/TrackSys v2r1.

2 Python Packages

This section presents the three packages used and needed to interface the tracking to Python. They are merely descriptive; the actual usage will be explained in the following sections 3. In this latter section 3 it will also be outlined some of the basic steps required to set up the "Python environment".

2.1 GaudiPython

`GaudiPython` is the package that interfaces the Gaudi framework with Python, i.e. the top-level framework for working with Gaudi applications in Python.

The latest version of the package can be consulted on the Web browser of the CVS repository at the URL [1]. A complete description is not in the scope of the present note. But note that a TWiki page [3] has recently been set up to document `GaudiPython` and to collect a series of "HowTo" and typical examples of applications.

Still, it is worth mentioning the main modules the general user is bound to need for standard jobs: `gaudimodule.py` and `GaudiAlgs.py`. `gaudimodule.py` allows to instantiate in Python the application manager, the heart of any Gaudi application; and it also defines the structure, properties and methods of a standard Python algorithm. With `GaudiAlgs.py` it is possible to write in Python algorithms deriving from GaudiAlgorithm, GaudiHistoAlg and GaudiTupleAlg. An example will be given at the end, in section 4.

In GaudiPython one can also find the units.py module containing the standard High Energy Physics (coherent) system of units as used in Geant4, CLHEP and all of Gaudi.

2.2 Tr/TrackPython

The package Tr/TrackPython has been introduced about a year ago to expose to Python the main tracking tools (via their interfaces). This is a simple package that is likely to evolve; eventually it may well be that its usefulness becomes redundant, as GaudiPython evolves. For the time being, it helps in dealing with the tracking tools.

The latest version of the package can be consulted on the Web browser of the CVS repository at the URL [4]. The package contains some examples of usage. But its central component is the gtracktools.py module, where the actual interfaces are defined. For the moment the following tracking interfaces are exposed to Python:

- ITrackExtrapolator: interface for tracking extrapolators;
- ITrackProjector: interface for tracking (state-on-measurement) projectors;
- ITrackFitter: interface for track fitters;
- ICloneFinder: interface for clone finder tools;
- IMeasurementProvider: interface for measurement provider tools.

Typing

```
>>> import gtracktools
```

one can easily inspect what is defined in the module:

```
>>> dir(gtracktools)
['IMeasurementProvider', 'ITrackCloneFinder', 'ITrackExtrapolator',
'ITrackFitter', 'ITrackProjector', 'Interface', 'TOOLSVC',
'__builtins__', '__doc__', '__file__', '__name__', 'cloneFinder',
'extrapolator', 'fitter', 'gaudimodule', 'measurementProvider',
'projector', 'setToolSvc']
```

Access to the actual tools can be done via the "helper" functions extrapolator, projector, cloneFinder, etc. All of them accept as a string argument the name of the actual tool. In case of doubt do, for instance:

```
>>> help(gtracktools.extrapolator)
Help on function extrapolator in module gtracktools:

extrapolator(name)
Usage:
e.g. TrackMasterExtrapolator = extrapolator( 'TrackMasterExtrapolator' )
```

2.3 Event/LinkerInstances

The package `Event/LinkerInstances` exists to facilitate the access in Python to the main linker classes `LinkedTo` and `LinkedFrom` defined in `Event/LinkerEvent`. These are the classes that deal with linker tables (produced earlier) from a user point of view.

The latest versions of the packages `Event/LinkerEvent` and `Event/LinkerInstances` can be consulted on the Web browser of the CVS repository at the URLs [5] and [6], respectively . Refer also to the LHCb note [7] to learn about all the functionality imbedded in the linker tables, and how to make use of it.

The interface to Python is done via the `eventassoc.py` module. Again, it is straightforward to inspect what it defines:

```
>>> import eventassoc
>>> dir(eventassoc)
['__builtins__', '__doc__', '__file__', '__name__', 'gaudimodule',
'linkedFrom', 'linkedTo']
```

Practical examples on how to deal with the `linkedTo` and `linkedFrom` helper functions are in section 3.4.

2.4 Mapping between C++ and Python

As a preliminary to what is following, it is good to keep in mind, for reference, how the syntaxes in C++ and Python map. Some of the most common examples are collected in table 1.

3 Tracking in Python

We will assume in all these sub-sections that the user is working in the Brunel environment, for the sake of example. All the tests have indeed been made in Brunel v30r2, and assume at least one event has been run, in order to produce tracks. But, of course, many

C++ to Python Mapping	
:: (the global namespace)	gaudimodule.gbl
Namespace::Class	gaudimodule.gbl.Namespace.Class
object = new Class(...)	object = Class(..)
enum::item	enum.item
null pointer	None

Table 1: Some examples of the mapping between the syntaxes in C++ and Python.

features would be equally applicable to any other Gaudi application; though, of course, the set of classes available would not be the same if running some events in DaVinci instead.

It goes without saying that the corresponding working environment is supposed to have been set previously (e.g. BrunelEnv). In addition, it is necessary to check that all the necessary packages are specified in the requirements file. Most if not all of the tracking packages are now included in a standard Brunel configuration. The exception might be `TrackPython`. Better do a quick check with

```
cmt sh use | grep Python
```

You may need to include in the requirements file (this package will be included by default in future versions of Brunel)

```
use TrackPython v* Tr
```

3.1 Retrieving Tracks in Brunel

For simplicity, we will assume that the user has run at least one event with the default Brunel options (`v200601.opts` options file). This can for instance be done with the following commands:

```
>>> import gaudimodule
>>> appMgr = gaudimodule.AppMgr( outputlevel=3,
                                 joboptions='../options/v200601.opts' )
>>> appMgr.run( 4 )
```

At this point several tracks containers have been produced; therefore a partial output of the Transient Event Store looks like

```
>>> EVT = appMgr.evtSvc()
>>> EVT.dump()
/Event
/Event/Gen
/Event/MC
/Event/MC/Header
# some lines skipped ...
/Event/Link/Rec
/Event/Link/Rec/Track
/Event/Link/Rec/Track/Forward
/Event/Link/Rec/Track/RZVelo
/Event/Link/Rec/Track/Velo
# some lines skipped ...
/Event/Rec
/Event/Rec/Header
/Event/Rec/Status
/Event/Rec/Track
/Event/Rec/Track/Forward
/Event/Rec/Track/RZVelo
/Event/Rec/Track/Velo
# some lines skipped ...
```

Note that as far as the containers on the DST goes, only those that have been requested are loaded, and will be seen calling this `dump()` method (the top level DST containers are always loaded by default). In the printout above one can easily spot the containers produced by the RZ Velo, 3D Velo and forward pattern recognition algorithms, as well as the linker tables associating those tracks to the MCParticles.

The Python dictionary for the Track Event Model classes – in `Event/TrackEvent` – is loaded typing

```
>>> appMgr.loaddict( 'TrackEventDict' )
```

This will probably become unnecessary with the future versions of `GaudiPython` (maybe already in v8r5), where the dictionaries will be loaded upon need, the search in LD_LIBRARY_PATH being done behind the scenes.

After running one or more events, get for the sake of argument the "Long" Tracks output by the long tracking pattern recognition algorithm `Pat/PatForward`:

```
>>> tracks = EVT[ 'Rec/Track/Forward' ]
>>> print tracks.size()
11
```

3.2 Using the Tracking Classes

3.2.1 How to Instantiate an Event Class

As a side comment, it may be handy at this point to stress that one can also instantiate the LHCb event classes (calling the constructor, really). All our LHCb event classes are defined in the "LHCb" namespace. As it turns out, the equivalent for GaudiPython of the C++ global namespace ("::") is `gaudimodule.gbl` (refer to table 1 for some examples of syntax mapping). Our event classes can then be represented in `gaudimodule.gbl.LHCb`. Here are some possible manipulations:

```
>>> Track = gaudimodule.gbl.LHCb.Track
>>> Track
<class '__main__.LHCb::Track'>
>>> defaultTrack = gaudimodule.gbl.LHCb.Track()
>>> defaultTrack
{ chi2PerDoF :  0
nDoF :  0 flags : 0
lhcbIDs :
states :
measurements :
nodes :
 }
```

You will have noticed that `Track` and `defaultTrack` are not quite the same. `defaultTrack` is an instance of `Track` – the `Track` constructor was called; whereas `gaudimodule.gbl.LHCb.Track` merely gets hold of the class definition.

Declaring via `gaudimodule.gbl.LHCb.Track` will have a direct usage when retrieving linker tables (section 3.4).

The purely geometrical tracking `Trajectory`-related classes are defined in `Kernel/LHCbKernel`. The "state trajectory" `StateTraj` is defined in `Tr/TrackFitEvent`. Dictionaries are now also built for all of them, as can be checked straightforwardly:

```
>>> appMgr.loaddict( 'LHCbKernelDict' )
>>> appMgr.loaddict( 'TrackFitEventDict' )
aLineTraj = gaudimodule.gbl.LHCb.LineTraj()
aStateTraj = gaudimodule.gbl.LHCb.StateTraj()
```

3.2.2 Side Comment on Dealing with LHCb Event Classes Enumerations

We've just seen how to instantiate a class, and how to get hold of the namespace of an LHCb event class. This provides in fact a simple and self-documenting way of getting hold of the enumerations defined in the event classes. Take the example of States and Measurements:

```
>>> State = gaudimodule.gbl.LHCb.State
>>> State.LocationUnknown
0
>>> State.AtT
5
>>> State.BegRich2
8
```

```
>>> Measurement = gaudimodule.gbl.LHCb.Measurement
>>> Measurement.Unknown
0
>>> Measurement.VeloR
1
>>> Measurement.TT
3
```

I would definitely recommend this syntax to make the usage of enumerations in Python as clear and close as possible to the C++ syntax: for instance the C++ statement `State::AtT` then "translates" in Python to `State.AtT`. We will use this throughout the rest of our examples. (This is clearer than `myState.AtT` (where `myState` would be some `State` instance variable), for example, though this also works, of course.)

3.2.3 Playing with the Tracking Classes

In section 3.1 we got hold of the "Long" tracks container produced by `Pat/PatForward`. We are now set to start "playing" with the tracks at hand ... and with all that can be retrieved from a Track, such as a State, a Measurement, etc.

A Track defines and contains a considerable amount of properties, enumerations and methods; a partial print out is:

```
>>> track = tracks[0]
>>> dir(track)
['AlreadyUsed', 'Backward', 'CnvForward', 'CnvKsTrack', 'CnvMatch',
'CnvSeed', 'CnvVelo', 'CnvVeloBack', 'CnvVeloTT', 'Downstream',
'FitFailed', 'FitUnknown', 'Fitted', 'HistoryUnknown', 'IPSelected',
'Invalid', 'IsA', 'Kalman', 'Long', 'PIDSelected', 'PatForward',
'PatKShort', 'PatRecIDs', 'PatRecMeas', 'PatVelo', 'PatVeloTT',
'ShowMembers', 'StatusUnknown', 'Streamer', 'StreamerNVirtual',
'TrackIdealPR', 'TrackKShort', 'TrackMatching', 'TrackSeeding',
'TrackVeloTT', 'TrgForward', 'TsaTrack', 'Ttrack', 'TypeUnknown',
'Unique', 'Upstream', 'Velo', 'VeloR', '__class__', '__delattr__',
# a few lines skipped ...
'__weakref__', 'addToAncestors', 'addToLhcbIDs',
'addToMeasurements', 'addToNodes', 'addToStates', 'ancestors',
'charge', 'checkFlag', 'checkHistory', 'checkHistoryFit',
'checkStatus', 'checkType', 'chi2', 'chi2PerDoF', 'clID', 'classID',
'clearAncestors', 'clearStates', 'clone', 'cloneWithKey',
'closestState', 'copy', 'fillStream', 'firstState', 'flag', 'flags',
'hasKey', 'hasStateAt', 'history', 'historyFit', 'index',
'isMeasurementOnTrack', 'isOnTrack', 'key', 'lhcbIDs',
'measurement', 'measurements', 'momentum', 'nDoF', 'nLHCbIDs',
'nMeasurements', 'nMeasurementsRemoved', 'nStates', 'nodes', 'p',
'parent', 'posMomCovariance', 'position', 'positionAndMomentum',
'pt', 'removeFromAncestors', 'removeFromLhcbIDs',
'removeFromMeasurements', 'removeFromNodes', 'removeFromStates',
'reset', 'serialize', 'setChi2PerDoF', 'setFlag', 'setFlags',
'setHistory', 'setHistoryFit', 'setLhcbIDs', 'setNDoF', 'setParent',
'setSpecific', 'setStatus', 'setType', 'slopes', 'specific',
'stateAt', 'states', 'status', 'type']
```

We will now concentrate on typical manipulations, leaving it up to the reader the possibility to further explore the classes. Operations involving the enumerations and related functions:

```
>>> track.checkType( Track.Long ), track.type() == Track.Long
(1, True)
>>> track.checkHistory( Track.PatForward )
1
>>> track.checkStatus( Track.Fitted )
0
```

How to look at the basic properties of the track:

```
>>> track.charge()
1
>>> track.nDoF()
12
```

How to look at the contents of the track:

```
>>> track.nStates()
2L
>>> state = track.states()[0]
>>> state.x(), state.y(), state.z()
(16.987653188753605, 0.0, 397.76935188731028)
```

The track can also be cloned, modified, etc.:

```
>>> newtrack = track.clone()
>>> newtrack.setFlag( Track.Clone, True )
>>> newtrack.checkFlag( Track.Clone )
1
```

The `setFlag(...)` method needs to be called to set the `Track.Clone` flag to True, as this is not done in the `clone()` method.

All the examples above were concerned with Tracks and States. But one can of course get hold of Measurements, Nodes, and LHCbIDs. Some examples are given below:

```
>>> print track.nLHCbIDs()
32
>>> print track.nMeasurements()
0
>>> ids = track.lhcbIDs()
>>> ids
<ROOT.vector<LHCb::LHCbID> object at 0xf4eab24>
```

3.3 Using the Tracking Tools

As presented in section 2.2, several tracking tools have been made accessible in Python. We here exemplify some possible, simple and handy ways of dealing with such tools. We assume, again, that some Tracks container was retrieved (see section 3.1) and stored in the variable `tracks`.

With this line one gets access to those tracking tools made accessible with `Tr/TrackPython`:

```
>>> appMgr.loaddict( 'TrackPythonDict' )
```

3.3.1 Extrapolators

Suppose you will want to play with the linear extrapolator. Then do:

```
>>> from gtracktools import setToolSvc, extrapolator
>>> setToolSvc( appMgr )
>>> linprop = extrapolator( 'TrackLinearExtrapolator' )
>>> dir(linprop)
# some lines skipped ...
'__setattr__', '__str__', '__weakref__', 'addRef', 'finalize',
'initialize', 'interfaceID', 'momentum', 'name', 'p', 'parent',
'position', 'positionAndMomentum', 'propagate', 'pt',
'queryInterface', 'release', 'slopes', 'transportMatrix', 'type']
```

Note that `setToolSvc(...)` only needs to be called once per job, not for each of the tools one wants to instantiate. This call may not be necessary anymore in the future.

The extrapolation of a state proceeds via a call to the propagate(...) method (below we decided to first clone the state, in order to keep unchanged the original state):

```
# "state" variable contains some random State
>>> state.x(), state.y(), state.z()
(491.95847296136429, -39.394353509484759, 9520.0)
>>> newstate = state.clone()
>>> newstate.x(), newstate.y(), newstate.z()
(491.95847296136429, -39.394353509484759, 9520.0)
>>> linprop.propagate( newstate, 5000. )
SUCCESS
>>> newstate.x(), newstate.y(), newstate.z()
(72.562676254077573, -21.114433639356392, 5000.0)
```

It is in fact preferable and safer, for general users, to extrapolate the track rather than a state from the track: internally, the extrapolator examines the states on the track, picks up the one closest to the z-position the track is to be extrapolated to, and calls the method above to extrapolate the state. To check the possible ways of extrapolation (i.e. the different signatures the `propagate(...)` method has), do, for instance

```
>>> help(linprop.propagate)
# one gets several lines of documentation
```

To propagate a track one proceeds along these lines:

```
# get hold of the Track to be extrapolated
>>> track = tracks[3]
# instantiate a State, to retrieve the result of the extrapolation
>>> state = gaudimodule.gbl.LHCb.State()
>>> state.x(), state.y(), state.z()
(0.0, 0.0, 0.0)
# instantiate the parabolic extrapolator
>>> parprop = extrapolator( 'TrackParabolicExtrapolator' )
# define the z-position to extrapolate to
>>> znew = 100.
>>> parprop.propagate( track, znew, state )
SUCCESS
>>> state.x(), state.y(), state.z()
(-5.859069220236659, -1.5738179596044015, 100.0)
```

3.3.2 Loading the Measurements from the List of LHCbIDs

After the pattern recognition, a standard Track contains by definition at least one State – the seed useful for the fitting – and a list of LHCbIDs. Internally, the tracks event fitter (the `TrackEventFitter` algorithm) – that fits "in one go" all the tracks in an input container –, calls at the very beginning the `MeasurementProvider` tool to "load" the Measurements on the Track from the list of LHCbIDs (unless the loading has already been done in a previous algorithm). These Measurements are substantially used by the fitting code. If wanted, one can of course do the loading of the Measurements at any point; the procedure is as below (it is not needed to call again `setToolSvc(...)`)

```
>>> from gtracktools import measurementProvider
>>> MeasurementProvider = measurementProvider( 'MeasurementProvider' )
>>> track.checkStatus( Track.PatRecIDs ) == True
True
>>> MeasurementProvider.load()
>>> MeasurementProvider.load( track )
SUCCESS
>>> track.setStatus( Track.PatRecMeas )
```

It should not be forgotten to set the relevant Track status flag – `Track::PatRecMeas` – accordingly.

Trying to redo `MeasurementProvider.load( track )` will have no effect but to trigger a series of WARNING messages, as the tool tries to re-load the Measurements, and this has already been made.

3.3.3 Fitting a Track "from A to Z"

At this point we have in hand, with the above-described, all that we need to fit, in Python, a track output by a pattern recognition algorithm, that is, a track with a list of LHCbIDs and a seed state. Two procedures are described in here. First one needs to instantiate the master fitter:

```
>>> from gtracktools import fitter
>>> masterfitter = fitter( 'TrackMasterFitter' )
# one gets some printout at this level ...
```

Fitting tracks with the default options cannot get simpler. The "complete job", i.e. fitting and setting of the appropriate flags, only takes a few lines:

```
>>> track = tracks[0]
>>> sc = masterfitter.fit( track )
>>> if sc == gaudimodule.SUCCESS:
...    track.setStatus( Track.Fitted )
... else:
...    track.setStatus( Track.FitFailed )
...    track.setFlag( Track.Invalid, True )
```

3.3.4 Checking on Clone Tracks

The `TrackCloneFinder` tool checks whether two input tracks are clones of each other based on their (possible) overlap of hits (LHCbIDs). Again the tool is retrieved with

```
>>> from gtracktools import cloneFinder
>>> TrackCloneFinder = cloneFinder( 'TrackCloneFinder' )
```

Some self-explanatory examples of usage are:

```
>>> track0 = tracks[0]
>>> track1 = tracks[1]
>>> TrackCloneFinder.areClones( track0, track1 ) == True
False
>>> track1 = track0.clone()
>>> TrackCloneFinder.areClones( track0, track1 ) == True
True
```

3.4 Using Linker Tables Associated to Tracks

The LHCb linker tables can be accessed in Python via the `eventassoc.py` module in `Event/LinkerInstances`.

Check first that `Event/LinkerInstances` is set in the requirements; otherwise just add to the requirements file the line

```
use LinkerInstances v* Event
```

Then the usage is standard. For a change, let's look at the 3D Velo Tracks in the `'Rec/Track/Velo'` container. Retrieving the linker table of these Tracks to the MC-Particles needs some declarations to define the type of the objects related:

```
>>> location = 'Rec/Track/Velo'
>>> tracks = EVT[ location ]
>>> from eventassoc import linkedTo
>>> Track = gaudimodule.gbl.LHCb.Track
>>> MCParticle = gaudimodule.gbl.LHCb.MCParticle
>>> LT = linkedTo( MCParticle, Track, location )
>>> LT
<ROOT.LinkedTo<LHCb::MCParticle,LHCb::Track> object at 0xf485460>
>>> LT.notFound() == False
True
```

The penultimate line confirms the table was found. A simple manipulation is then :

```
>>> track = tracks[7]
>>> track.key()
7
>>> mcp = LT.first(track)
>>> mcp.key()
849
>>> mcp
{ momentum :    (-673.62,-416.03,11761.6,11789)
particleID :    { pid : 211
 }
 }
>>> mcp = LT.next()
>>> mcp == None
True
```

Calling `next()` gets the next MCParticle associated to the Track, if any. The penultimate line indicates that the Track #7 actually had only 1 associated MCParticle, since `mcp == None` means that `mcp` is a `NULL` pointer (c.f. table 1).

It is also easy to "go back and forth" in the association Track-MCParticle combining the `LinkedTo` and `LinkedFrom` classes:

```
>>> from eventassoc import linkedFrom
>>> LF = linkedFrom(Track,MCParticle,location)
>>> LF
<ROOT.LinkedFrom<LHCb::Track,LHCb::MCParticle> object at 0xf4d6210>
>>> LF.notFound() == False
True
>>> track.key()
7
>>> mcp = LT.first(track)
>>> mcp.key()
849
>>> trk = LF.first(mcp)
>>> trk.key()
7
```

The linker tables can be manipulated, in addition, with the looks of the relations (association) tables:

```
>>> range = LT.range(track)
>>> range
<ROOT.vector<LHCb::MCParticle*> object at 0xfe27e90>
>>> range.size()
1L
>>> mcp = range[0]
>>> mcp.key()
849
```

4 Developing New Tools and Algorithms

The fact of being in the (interactive or not) Python environment can be exploited much further. First of all, the existing and rather extensive Python modules can be used at will - see e.g. [8] for the list of the standard modules, the standard library. This makes it very easy to write new tools and functions, and to use them in the same environment. Not to say the least, a rather complete part of ROOT is available to Python via PyRoot. One can also write dedicated GaudiPython algorithms.

Furthermore, and rather importantly, it is worth emphasizing that the Python environment can be exploited at will to prototype new ideas, testing them "as one goes along". This can be very efficient; in fact several things have been prototyped is such a way.

It is also possible to run alongside and rather transparently, in Python, Gaudi C++ and Python algorithms. We outline in section 5 how to set up a "mixed job".

4.1 User-defined Functions

Anyone working in an interactive Python session is bound to write at some point dedicated functions, be it for debugging, executing a series of calculations, developing new ideas, etc.

"Doing something" on each track in a set/container might be as simple as

```
>>> for t in tracks:
...     do_something( t )
```

And how about getting into a list variable the momenta for some selected tracks? The solution may be trivial depending on the actual problem, e.g.:

```
>>> list_p = [ t.p() for t in tracks if t.chi2PerDoF() < 1. ]
```

4.2 A User-defined Algorithm

A template for a Python instantiation of the standard GaudiAlgorithm looks like this:

```
import gaudimodule
from GaudiAlgs import GaudiAlgo

########################################################################
class MyAlgo( GaudiAlgo ):
   """ My nice and handy user algorithm """
########################################################################

# ======================================================================
# Initialization
# ======================================================================
   def __init__ ( self , name = ’MyAlg’ ) :
      """ Constructor """
      GaudiAlgo.__init__( self , name )
      print self.name(), ’==> Initialize’
      return gaudimodule.SUCCESS

# ======================================================================
# Main execution
# ======================================================================
   def execute( self ):
      """ The ’execute’ method, invoked for each event """
      print self.name(), ’==> Execute’
      # do something here ...
      return gaudimodule.SUCCESS

# ======================================================================
# Finalization
# ======================================================================
   def finalize( self ):
      """ The ’finalize’ method, called at the end of the job """
      print self.name(), ’==> Finalize’
      return GaudiAlgo.finalize()
```

5 A Complete Example of a Tracking Job in Python

We now have all that is necessary to run a complete (Brunel) job in Python. In fact, the framework even allows to easily configure a "mixed job" containing not only the usual sequencers of C++ algorithms, but also some Python Gaudi algorithms, if desired.

5.1 My Example Python Algorithm

As a final example, let us assume that we want to set up a (very simple) Python job to monitor and plot the momentum of Long Tracks produced by the `Pat/PatForward` pattern recognition algorithm. We could follow the traditional way, writing and adding a C++ Gaudi algorithm to the tracking sequencer of Brunel. We want to demonstrate an alternative, writing a Python algorithm, and adding it to the application manager.

For such a simple task, we instantiate in Python a GaudiAlgorithm - one could instead decide to instantiate a GaudiHistoAlg for more complex and detailed histogram manipulations.

For every event this simple algorithm retrieves the "Long" tracks and histograms the magnitude of their momenta. In Python the code can be written in a very compact, yet clear way:

```
>>> tracks = self.get( 'Rec/Track/Forward' )
>>> for t in tracks:
...    histo.Fill( t.p() / GeV )
```

The full piece of code concerning the algorithm is given below:

```
import gaudimodule
from GaudiAlgs import GaudiAlgo

from units import GeV

# Define the histograms
# =======================================================================
from ROOT import TH1F, TFile

histo = TH1F( 'long_p', 'P (GeV) for  Long tracks', 100, 0., 100. )

#######################################################################
class LongTrackPAlgo( GaudiAlgo ):
    """Algorithm to plot the momentum of Long tracks from PatForward"""

    def __init__ ( self , name = 'LongTrackP' ) :
        GaudiAlgo.__init__( self , name )
        print self.name(), '==> Initialize'
# =======================================================================
    def execute( self ):
        print self.name(), '==> Execute'
        tracks = self.get( 'Rec/Track/Forward' )
        for t in tracks:
            histo.Fill( t.p() / GeV )
        return gaudimodule.SUCCESS
# =======================================================================
    def finalize( self ):
        print self.name(), '==> Execute'
        #print some statistics on tracks properties
        print '%34s : mean %10.6f rms %10.6f '
             % ( histo.GetTitle(), histo.GetMean(), histo.GetRMS() )
        # write out the histogram
        f = TFile( 'LongTrackPAlgo.root', 'recreate' )
        histo.Write()
        f.Close()
        return GaudiAlgo.finalize()
```

5.2 Running my Brunel Job in Python

There are very many ways of setting things up to run a job comprising some standard C++ algorithms together with Python algorithms. What follows is just one possible solution, taken here for its simplicity.

Let's assume that the algorithm's code in the section above (in 5.1) has been saved in the file `myJob.py`. We now append to the file this piece of code, where the running part of our job is set up:

```
# ======================================================================
# Job execution
# ======================================================================
if '__main__' == __name__ :

    appMgr = gaudimodule.AppMgr( outputlevel=3,
                                 joboptions='../options/v200601.opts' )

    appMgr.loaddict( 'TrackEventDict' )
    appMgr.loaddict( 'TrackPythonDict' )

    EVT  = appMgr.evtSvc()

    appMgr.addAlgorithm( LongTrackPAlgo( 'LongTrackP' ) )

    appMgr.run( 250 )

    appMgr.finalize()
```

To run the job and save the output to a "log file" simply type at the command prompt something along

```
python myJob.py > myJob.log
```

One could also prefer to enter the interactive mode before the end of the job. One way would be to omit from the above the "appMgr.finalize()" line. and to run the job as

```
python -i myJob.py
```

After the 250 events have been run one enters the Python prompt. It is then possible to plot directly the histogram, run some more events, plot again the (updated) histogram, etc.:

```
histo.Draw()
appMgr.run(100)
histo.Draw()
# continue "playing" interactively ...
```

Once again let me stress that this solution is rather simple, and merely given to "get going". There are alternatives and/or more sophisticated ways of running GaudiPython jobs.

6 Final Remarks

The note has presented for the first time several of the tools of the Python framework that are available. We outlined some of the technical aspects and concentrated on detailing those of most direct interest to the user, taking in particular the viewpoint of a tracking user.

Other topics are rather interesting and would deserve a presentation and discussion. It is left for the future. Among these one should certainly mention that:

- we did not describe at all in this note the Python environment already available in Panoramix [9], our event display. Here also there have been many developments;
- the Bender application provides a lot more functionality, which is needed for physics analysis;
- the LHCb "Reconstruction" webpages collect very useful HowTo's, examples, Python scripts, etc. that deserve a look [10];
- the tools and algorithms are configurable. Changing the options/properties can also be done in Python.

Now it is up to you to make the most out of all that is available ... enjoy it!

References

[1] *http://isscvs.cern.ch/cgi-bin/cvsweb.cgi/GaudiPython/?cvsroot=Gaudi*

[2] *http://lhcb-comp.web.cern.ch/lhcb-comp/Analysis/Bender/*

[3] *https://twiki.cern.ch/twiki/bin/view/LHCb/GaudiPython*

[4] *http://isscvs.cern.ch/cgi-bin/cvsweb.cgi/Tr/TrackPython/?cvsroot=lhcb*

[5] *http://isscvs.cern.ch/cgi-bin/cvsweb.cgi/Event/LinkerEvent/?cvsroot=lhcb*

[6] *http://isscvs.cern.ch/cgi-bin/cvsweb.cgi/Event/LinkerInstances/?cvsroot=lhcb*

[7] O. Callot, *Updated Usage of the Linker Classes*, LHCb Note LHCb 2006-008, March 2006

[8] *http://docs.python.org/modindex.html*

[9] *http://lhcb-comp.web.cern.ch/lhcb-comp/Frameworks/Visualization/*

[10] *http://lhcb-reconstruction.web.cern.ch/lhcb-reconstruction/*