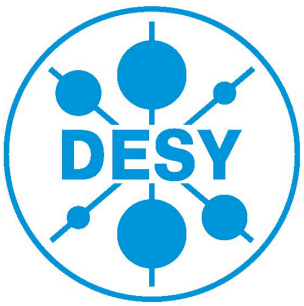


Development of a Python module for interactively controlling the C Event Display (CED)

Björn Klaas, Georg-August-Universität Göttingen, Germany

September 4, 2013

Abstract

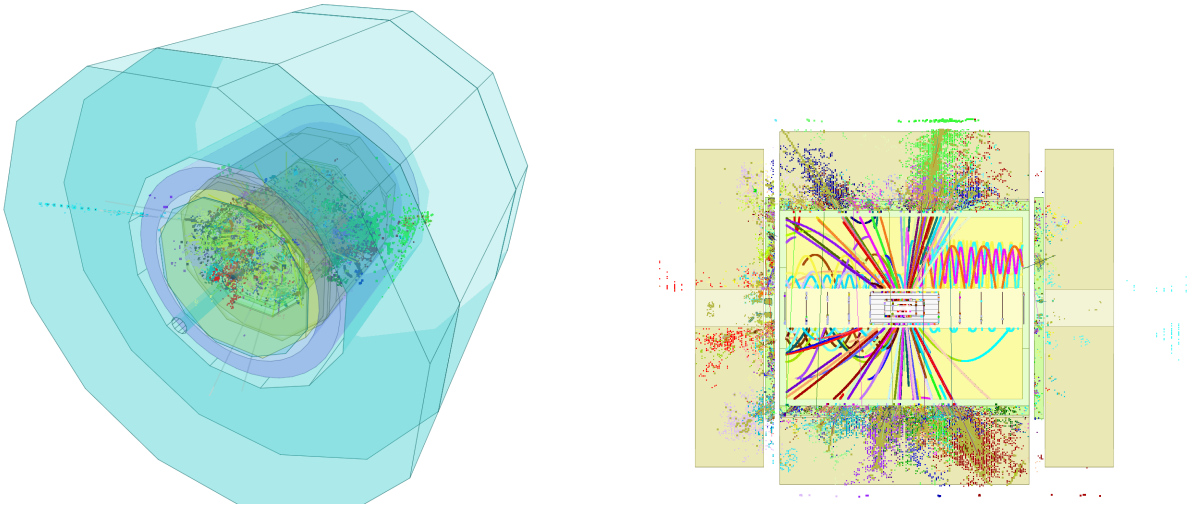
We describe the development of a Python Event Display (pyced) for the International Linear Collider (ILC). Pyced is based on Python bindings to the C++/C libraries LCIO and CED, the event data model and event display currently in use. The use of Python introduces additional flexibility in interactive event analysis. The development of the Python module *pyced* is described and a user guide presented in a *How To* form. Instructions for advanced usage are given. Options for expanding functionality and optimizing performance are discussed.

Contents

1 Introduction

The presented work was done in the *Forschung an Lepton Collidern* (FLC) group, whose focus currently lies on the development of the *International Linear Collider* (ILC). Two detectors are being developed for the ILC, the *International Large Detector* (ILD) and the *Silicon Detector* (SiD). The DESY FLC group contributes to the ILD.

To study the performance of the proposed detector design events are generated using Monte Carlo methods. These simulated events are visualized using the *C Event Display* (CED), written in `C` [1]. To load events from an *LCIO* file [2], process them and send them to CED the `C++` application *CEDViewer* [3] is used. Once compiled and executed the functionality of the application cannot be modified further, therefore presenting static images. These can only be altered using the functions build into CED, like rotating the view and showing/hiding individual data or detector layers (Figure 1).



(a) Complete Event. **(b)** Side-view of Tracker- and Calorimeterhits.

Figure 1: An Event viewed using CED. Shown is a complete Event and a side-view with Tracker and Calorimeter layer, and several Detector layers.

The newly developed `Python` module *pyced* is designed to replace the `C++` application. In its final version it will contain all functionality from the `C++` code and expand it by offering an Interactive Mode for dynamically modifying the contents of the data layers during runtime. This allows the user to e.g. dynamically apply various cuts, like energy or p_t cuts, and tune the display to his/her liking.

2 Development

In a first step the `C++` code from *CEDViewer* was ported to `Python`. The resulting code was split into multiple functions and expanded by adding an easy to customize set of maps containing the drawing parameters, and a set of functions making the Interactive

Mode more user-friendly. Lastly the resulting code was optimized using Python's build-in profiling module profile.

2.1 Porting CEDViewer to Python

2.1.1 CEDViewer

The original C++ code contains only one large method, doing all necessary calculations and CED function calls. Using if statements the type of the currently processed collection is determined and the proper code executed. Configuration of the layer, marker, and size settings for individual collections is possible via a separate config file written in XML.

2.1.2 Python Bindings

The LCIO framework is written in C++ and CED in C. Typically this would require porting the entire framework to Python to use the classes and functions defined in LCIO and CED. Fortunately Root provides a set of Python bindings, and the Python module ctypes a set of methods for converting Python types to C types. This way the LCIO objects containing the event data can be loaded directly into Python and used in a pythonic way. The C functions can be called using wrappers, converting the arguments passed to the functions via the ctypes module.

2.1.3 Python Implementation

The Python code was split into multiple functions to increase customizability, modularity and re-usability.

Class PYCED() The definition of the class contains a set of enum like variables, which are to be seen as constants and left untouched. They encode the names of the various color schemes and variants of drawing the helix for tracks. The definition also contains a function to load the color codes of the chosen scheme.

An instance of the class, called g for *global*, is created in the global scope of the pyced module and dynamically expanded with all required and optional settings used for displaying the events. The assignments of these settings are grouped in a separate configuration file.

Configuration File All user definable settings are combined in a separate config file (by default *pyced.cfg.py*). To offer the greatest flexibility and customizability the config file is written in Python and loaded into the module *as is*. This means the Python syntax has to be obeyed when editing the config file. It also means that the config file can contain function definitions and calls. This can be useful for adding custom functions without editing the main module.

The collection specific parts of the configuration are held in three maps, the *default map* (dm), *type map* (tm) and *name map* (nm). The dm contains all basic settings necessary

for drawing a collection. When a collection is processed the settings from the `dm` are loaded and then extended and possibly overwritten by the settings from the `tm` entry matching the collection type. These are in turn extended and possibly overwritten by the `nm` entry matching the collection name. This means changing the entries of a map only changes the collection settings if they are not defined in a higher level map as well. The maps are individually wrapped in functions, to allow reseting them to the hard-coded values when edited during runtime.

Event Reading and Processing The main loop of the module is contained in a dedicated function. It initiates reading of an event from a chosen LCIO file, resets the CED, initiates loading the detector design from a GEAR file [4], calls the function processing the loaded event, and sends the data to the CED. After each iteration of the loop it waits for user input and either draws the next event or terminates. It also terminates when the LCIO file contains no more events.

All events are processed by the same function, looping over all collections in the event. It initiates the dynamic expansion of the processed collection with the drawing parameters defined in the configuration file, checks if the collection is selected for drawing and, if true, calls the drawing function associated with the collection. Once all enabled collections are drawn it passes the collected layer descriptions to the CED.

Collection Processing Depending on the type of the collection one of five functions is called, either processing collections containing only hits, or also tracks, clusters, Monte Carlo particles or reconstructed particles. The functions process the data stored in the collection and call the wrapper functions matching the data types. These convert the arguments to the corresponding `C` types and pass them to the CED function, either drawing hits, lines, or helices.

Support Functions To perform smaller tasks a set of support functions was developed. These set-up initial values, process the coloring of *Tracker* and *Calorimeter* collections, process the layer descriptions, collect hits from subtracks of tracks, add the drawing parameters from the configuration file to the collections, load `C` and `C++` libraries and wrap `C` functions. They also provide various cuts, implemented using closures, to limit e.g. the energy or p_t range of drawn collections.

Interactive Mode Functions This set of small functions is intended to ease the usage of the Interactive Mode by offering shortcuts, e.g. for modifying drawing parameters, redrawing events, loading the next/previous event or reseting the parameters.

2.1.4 Expanded Functionality

The cut functionality and Interactive Mode functions mentioned above were added to the functionality found in the original CEDViewer. By using `Python`, an interpreted language, the possibility to escape the modules main loop to the interpreter (Interactive Mode) and modify and call functions during runtime was added.

2.2 Optimization

Among the modules provided by the *Python Standard Library* are the `time` and the `profile` modules, allowing precise timing of the execution time of the entire `pyced` module and individual parts thereof. The `time` module was used by storing timing information at various points of the program and comparing these to each other. The `profile` module was invoked by executing

```
python -m profile pyced.py [LCIO file]
```

It returns a list containing the number of times each function of the module was called and various execution times; combined time and time per call, including and excluding function calls from inside the function.

3 Usage

There are various levels of user involvement and customization with which the module can be used. *Basic Usage* describes how to run the program and view events, *Intermediate Usage* explains how to enter the Interactive Mode and presents an overview of the functions available in the current `pyced` version. *Advanced Usage* gives some examples on how to modify not just the drawing parameters, but the module itself.

3.1 Basic Usage

The most basic usage is simply calling the module and viewing the events. This is achieved by entering

```
python pyced.py [LCIO file]
```

in a console window. The events will be displayed one by one, drawing of the next event is initiated by hitting `<Enter>`, execution is stopped by entering `q`.

For performance reasons it is advised to do simple viewing using the `C++` version, since compiled `C++` code is almost always faster then interpreted `Python` code. In this case by several seconds per event full event.

3.2 Intermediate Usage

To use the Interactive Mode of `pyced` it has to be launched with the `-i` option,

```
python -i pyced.py [LCIO file]
```

This allows users to access the `Python` interpreter when entering `q`, instead of exiting `Python`. The interpreter is shut down by entering `<Ctrl>-D`.

When in Interactive Mode drawing parameters can be changed dynamically by using the predefined functions listed in Table 1. Table 2 shows the available drawing options. Use common sense when applying these, since not all options fit all collections.

Function	Description
redraw()	Redraws the displayed event using the current settings.
next()	Draws the next event using the current settings.
prev()	Loads the previous event. (Not yet functional)
set(col, attr, val)	Sets the chosen attribute of the collection to the entered value. Collection and attribute have to be in quotation marks (string literals). Possible values for `col` are a collection name, collection type, "default", and "all".
redrawWith (col, attr, val)	Calls `set()` and redraws the event.
reset(*maps)	Resets the entered maps to the values defined in the config file. Options are "all" and combinations of "nm", "tm" and "dm".
enable(col)	Enables the collection for drawing, options are the same as for `set()`.
disable(col)	Excludes the collection from drawing to increase performance, options are the same as for `set()`.
picking()	Re-enters the command loop to allow picking.

Table 1: Summary of the Interactive Mode Functions in `pyced` version 1.0.

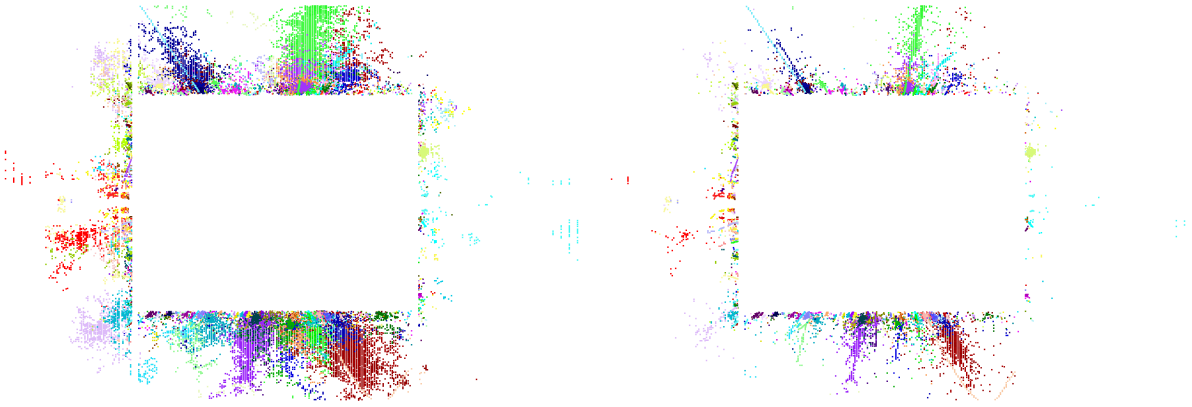
Parameter	Value	Description
marker	Int	Sets the type of marker.
layer	Int	Sets the layer.
size	Int	Sets the size of the collection's main data type.
hitsize	Int	Sets the size of the collection's hits.
clustersize	Int	Sets the size of the collection's clusters.
draw	Bool	En-/Disables the collection, for better performance.
cut	Callable	Sets the type and value of a cut. Predefined are `eCut(val, type)`, `ptCut(val, type)` and `cosThetaCut(val, type)`. The optional `type` argument can be "greater" (than cut), "smaller" (than cut) or a number (range: cut < energy < cutType), default is "greater".
color	Callable	`fixedColor("colorcode")`, `SimTrackerHitColor()`, or `SimCalorimeterHitColor()`. Only applies to *Hit* types
callDraw	Callable	Sets the function for drawing the collection, options are `drawHits`, `drawTracks`, `drawClusters`, `drawMCParticles`, and `drawReconstructedParticle`.

Table 2: Summary of the Drawing Parameters in `pyced` version 1.0.

Figure 2 demonstrates an energy cut at 0.0001 GeV of the *SimCalorimeterHits* of an event, achieved by entering

```
set("SimCalorimeterHit", "cut", eCut(0.0001))
redraw()
```

when in Interactive Mode.



(a) Before energy cut. **(b)** After energy cut.

Figure 2: Energy cut of the SimCalorimeterHits of an Event. The Event is shown in side-view without perspective and all detector layers disabled.

3.3 Advanced Usage

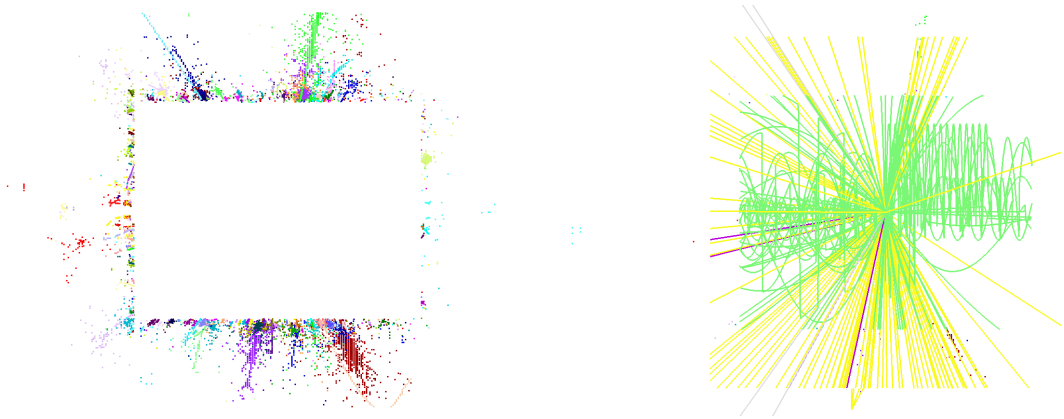
Users familiar with `Python` can edit the module as well as the parameters. This can be done either temporarily and on-the-fly during runtime, or permanently in the config file. Additions which prove useful for a wide range of users may be directly implemented in future versions of `pyced`.

To demonstrate how to add a function on the fly the example from *Intermediate Usage* is extended by adding a shortcut for doing a p_t cut and applying this to the *MCParticle* collection. The code

```
enable("MCParticle")
redrawWith("SimCalorimeterHit", "cut", eCut(0.01))
```

enables the previously disabled MCParticle collection and draws it while simultaneously increasing the energy threshold for the SimCalorimeterHits to 0.01 GeV (Figure 3).
The code

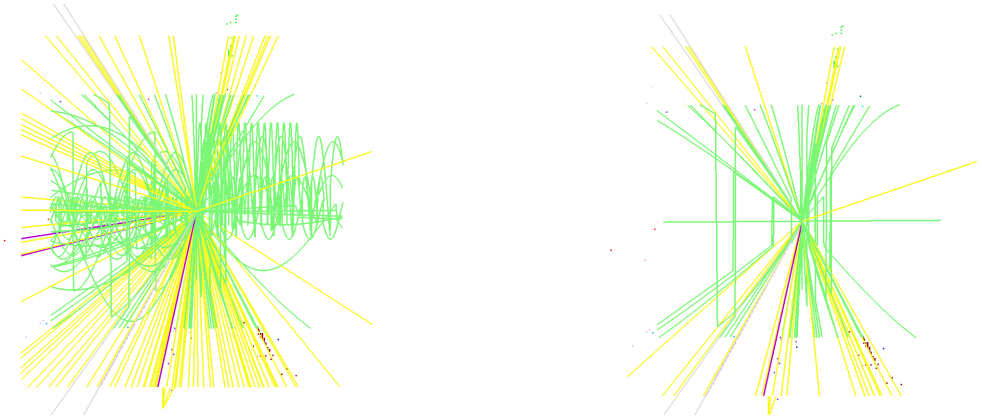
```
def makePtCut(collection, value) :
    set(collection, "cut", ptCut(value))
    redraw()
makePtCut("MCParticle", 1)
```



(a) Before cut and adding MCParticles. **(b)** After cut and adding MCParticles.

Figure 3: Increased energy threshold for the SimCalorimeterHits and added MCParticles.

adds a function for easily adding p_t cuts and is called to apply one to the MCParticle collection. This is shown in Figure 4.



(a) Before cut. **(b)** After cut.

Figure 4: Applying a p_t cut to the MCParticle collection.

For advanced usage the `pyced` functions can be loaded into the interpreter or other modules using

```
from pyced import *
```

This makes all functions from the module directly available and creates the instance `g` of the `PYCED` class and adds all values from the config file to it. The attribute `g.fileName` has to be manually set to the LCIO file to read. Then `init()` can be called to load the necessary libraries, open the specified file and connect to CED.

4 Outlook

Even though almost all functionality from the CEDViewer was ported to the `Python` module there is still some work to be done. The functionality of the module could be extended and the user-friendliness improved. The code needs further optimization as well, to increase its usefulness by shortening the load time of events. Possibilities to achieve this are pointed out below.

4.1 Further Development

A `Python` vector class should be developed to add full support of the *TrackerHitPlane* collection type to `pyced`. This is the last remaining step to having ported all functionality from CEDViewer.

The existing Interactive Mode functions can be extended by adding a `setType` function, which changes the parameters for all collections of a certain type. So far only the type map is edited when applying changes to a collection, which has no effect if the same setting exists in the name map. A similar option already exists for setting the attributes of all collections.

An easy way to apply multiple cuts to the same collection should be added. This could be achieved by adding another closure providing the option to set the "cut" attribute to something like `multiCut(eCut(0.0001), ptCut(1))`, taking an undefined number of parameters.

Proper reading of GEAR files could be implemented, to allow e.g. spatially differing B-Fields.

The amount and simplicity of the Interactive Mode functions can always be increased.

4.2 Optimization

Various general and `Python`-specific optimizations are still to be done.

Generally the amount of string comparisons should be minimized and as many `if`-statements removed from loops as possible.

More specifically for `Python` it is much faster to use local variables than global ones. Dots (for referencing attributes) should be avoided, which is done by e.g. defining `marker = col.marker`. This is already done for all drawing functions. There are also multiple looping techniques and data structures which should be examined.

Updating the code to `Python 3` would further increase performance, but require all machines running `pyced` to be updated as well.

The loading of the LCIO objects into `Python` and the look-up of their attributes seem to require a substantial amount of the execution time and leave room for improvement.

To measure the performance of the code the modules `time` and `profile` have been proven to be very handy.

References

[1] http://ilcsoft.desy.de/portal/software_packages/ced/

[2] http://ilcsoft.desy.de/portal/software_packages/lcio/

[3] http://ilcsoft.desy.de/portal/software_packages/cedviewer/

[4] http://ilcsoft.desy.de/portal/software_packages/gear/