Data Analysis/Visualization for the Object-Oriented LHC era

Y. Adesanya *CERN, Geneva, Switzerland*

Abstract

The LHC++ [1] (Libraries for HEP Computing) working group was established in mid-1995, in response to a stated requirement from the LHC collaborations, to investigate a ”CERNLIB-like” environment, but based on C++. The strategy of the working group, which involves representatives from CERN experiments and outside, has been to adopt standards - either de-jure and de-facto - wherever appropriate and to concentrate HEP effort on HEP-specific problems. The presentation will focus on those areas of the proposed LHC++ environment related to data analysis and visualization, and will include a discussion of HEP-specific histogram classes, why they are needed, and their relationship with other components of the strategy, including Object Databases (ODBMS) and industry-standard scientific visualization frameworks (AVS Express and IRIS Explorer).

Keywords: Object-Oriented; C++; Standards; Histograms; Persistency; Visualization

1 Introduction

The LHC era introduces new requirements which cannot be met by the CERNLIB software foundation. Effective re-evaluation demands a change in our approach. In the past, there was just cause for CERN’s in-house development of mechanisms used to handle fundamental computing tasks. Today’s modular approach, adopted through object-oriented and component-ware environments, enables common elements of performance demanding applications to be identified and exploited. Such elements are far more accessible today than yesteryear. The move towards object-oriented perspectives of problem solving has left FORTRAN in its wake. The latest generation of high energy physicists regard C++ as the current leader in high-level languages. The imminent arrival of the ANSI standard [2] should resolve large compiler differences, easing the pain of porting code. Many features are introduced as part of the language. Not all are ground breaking, but the point is that they form a standard, to be adopted by all who use C++.

The STL [3] (Standard Template Library) need not be used directly by users but we must ensure that class libraries are as compatible as possible while avoiding redundancy. We need to identify which parts of a future framework should be developed specifically for LHC. Object-Oriented components will help us to create a consistent infrastructure, preserving objects as the units of operation throughout all levels of the data analysis process. Nobody believes that todays implementation will address all the needs of 2006. But we can fuse together a data analysis/visualization environment that is far more future-proof than its predecessor, *i.e.* a software foundation that is coherent, maintainable and based on recognized standards. The prototype data analysis/visualization framework will addresse the immediate needs of LHC physicists with an eye to the future.

2 Areas associated with HEP unique requirements

2.1 Histogram classes

The data analysis/visualization process model relies heavily upon suitable data structures. The working group considered existing solutions, both commercial and otherwise. General purpose histogramming packages seemed to fall short of our demands. This prompted the creation of the Histogram sub-group whose role it was to draw up a well defined set of requirements for a design. It appeared deceptively easy. What physicist couldn’t describe the features of a histogram? The requirements are still being revised.

2.2 Object Peristency

Persistency was not treated as an implicit feature of the histogram classes but it was advised that prototype coding should begin as soon as possible, to help validate the analysis and see how much had to be taken into consideration when making a class persistent capable. Object databases fitted into the LHC++ scheme. Nobody wanted to see persistent data as rows and columns in a file. The RD45 [4] group was established in 1995 to research object persistency and to produce recommendations for HEP projects such as GEANT 4. They understood the storage and performance demands of future experiments and had chosen the Objectivity ODBMS [5] as a suitable choice for prototype histogram class library.

2.3 Modular interactive visualization for HEP

Interactive data analysis and visualization is a major concern. Experience comes from several working group members who have spent a considerable amount of time both developing and using the PAW visualization package (amongst others). Over the course of time, lessons were learned relating to maintainability of such systems. The AGOCG (Advisory Group On Computer Graphics) confirmed our beliefs in Modular Visualization Systems. They provide well structured functionality which can be easily extended. The modular approach allows supports encapsulation, vital for the reuse of legacy code. Although such systems tend to have larger learning curves, their benefits provide ample justification.

3 Designing histograms for HEP

Before the decision was taken to develop HEP histograms, some individuals had already contemplated creating their own ad-hoc classes. It was time for a common solution. An evaluation of Rogue Waves Math.h++ histogram class showed several limitations. Work began on formalizing histogram package requirements. The criteria include:

- support for weighted entries;
- multi-dimensional histograms;
- recording underflow and overflow;
- variety of binning systems which can be extended;
- persistence capability;
- general transformations;
- statistics.

Once a draft requirements document had been produced, an initial class diagram was derived. It's most striking feature is the distinct separation between histogram bins and the problem space which defines their boundaries. **Histogram** objects contain a collection of bins which store weights and errors. **Partition** instances are responsible for generating a bin identifier from a data point.
The classes are connected via a one-to-many relationship. This acknowledges the fact that users often wish to create create several histograms in a program which share the same characteristics. Bin contents can be set through the **Histogram** member functions:

- set_bin_value(X:Data_point,value:Real)
- set_bin_error(X:Data_point,error:Real)

Both calls cause the recipient **Histogram** to retrieve the bin qualifier ID from its respective partition. The appropriate bin can then be referenced and the value or error is returned. Histograms can also undergo operations such as addition, multiplication and subtraction. The OMT diagram (Figure 1.) lists operations which accept **Histogram** objects as arguments. This will be extended to scalar types in the future. Two subclasses directly inherit from the **Histogram** class: **DataHistogram** is for accumulating weights. The other subclass is **ProfileHistogram**.

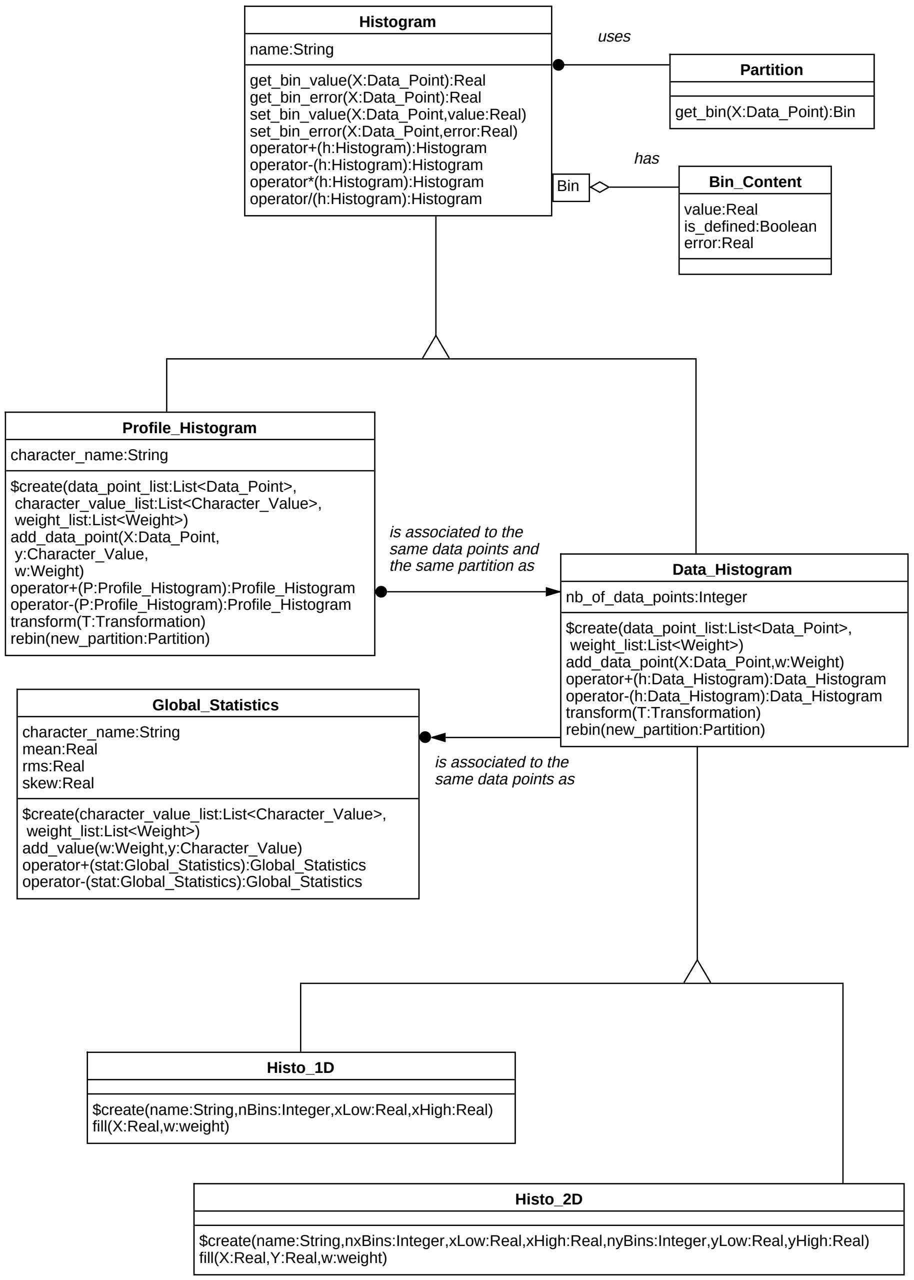
Histogram
name:String
get_bin_value(X:Data_Point):Real
get_bin_error(X:Data_Point):Real
set_bin_value(X:Data_Point,value:Real)
set_bin_error(X:Data_Point,error:Real)
operator+(h:Histogram):Histogram
operator-(h:Histogram):Histogram
operator*(h:Histogram):Histogram
operator/(h:Histogram):Histogram
uses
Partition
get_bin(X:Data_Point):Bin
has
Bin
Bin_Content
value:Real
is_defined:Boolean
error:Real
Profile_Histogram
character_name:String
$create(data_point_list:List<Data_Point>, character_value_list:List<Character_Value>, weight_list:List<Weight>)
add_data_point(X:Data_Point, y:Character_Value, w:Weight)
operator+(P:Profile_Histogram):Profile_Histogram
operator-(P:Profile_Histogram):Profile_Histogram
transform(T:Transformation)
rebin(new_partition:Partition)
is associated to the same data points and the same partition as
Data_Histogram
nb_of_data_points:Integer
$create(data_point_list:List<Data_Point>, weight_list:List<Weight>)
add_data_point(X:Data_Point,w:Weight)
operator+(h:Data_Histogram):Data_Histogram
operator-(h:Data_Histogram):Data_Histogram
transform(T:Transformation)
rebin(new_partition:Partition)
Global_Statistics
character_name:String
mean:Real
rms:Real
skew:Real
$create(character_value_list:List<Character_Value>, weight_list:List<Weight>)
add_value(w:Weight,y:Character_Value)
operator+(stat:Global_Statistics):Global_Statistics
operator-(stat:Global_Statistics):Global_Statistics
is associated to the same data points as
Histo_1D
$create(name:String,nBins:Integer,xLow:Real,xHigh:Real)
fill(X:Real,w:weight)
Histo_2D
$create(name:String,nxBins:Integer,xLow:Real,xHigh:Real,nyBins:Integer,yLow:Real,yHigh:Real)
fill(X:Real,Y:Real,w:weight)

Figure 1: Top-level OMT diagram of Histogram class structure

The **Partition** class holds a list of references to elementary partitions. An **ElementaryPartition** object is abstract, corresponding to a single dimension within a partition. Through sub-classing, different kinds of dimension are implemented such as fixed length, variable length, sets, etc. **ElementaryPartition** objects can also be shared amongst different **Partition** objects. The design is fairly open in terms of extensibility. The initial class structure had been defined but concrete test cases would aid validation and refinement. Prototypes would act as a catalyst by promoting user feedback. Other issues included the interface design. Persistency also had to be dealt with. Just how much work is required in making C++ classes persistent?

4 The impact of the Objectivity on class design

A subset of the histogram class structure was chosen for immediate coding, enabling the creation of data histograms of n-dimension (fixed/variable width bins). The first stage was simple enough. Transient C++ classes were written using normal pointers to handle associations. The base types of the data members belonging to persistent capable classes were switched to ODMG [6] (Object Database Management Group) compliant types. These types have a fixed implementation to overcome compiler and machine architecture differences. C++ memory pointers cannot be stored inside persistent classes. Objectivity provides a set of generic “smart” pointers used to reference persistent objects. The de-reference operators are overloaded so object data members and functions can be accessed just like any normal object pointer. These smart pointers are also used to implement associations between persistent objects. Objectivity also provides persistent string classes and STL like containers including vectors and ordered lists which can hold persistent objects or their references.

4.1 The role of HepRef

The provisions are there but what if you wish to build a class that will exist in transient and persistent forms without having to make extensive code changes? The RD45 group were already tackling the problem. Using smart pointers and macros a generic interface to references and containers was developed. In its persistent state, Objectivity references and containers were used. In transient form, containers were implemented using Rogue Wave’s Tools.h++ [7] class library.The generic reference is known as **HepRef**.

4.2 Creating and accessing persistent histograms

In Objectivity the key unit of database operations is the **Transaction** class. There can be only one active transaction open at any time (per-process). While a transaction is running, objects can be created, deleted and amended but changes are only made to the database once the transaction is closed. There is also the option of aborting a transaction. Everything must be done through persistent references. This is an important point to remember because when you de-reference in order to access an objects member functions, objectivity hands the controls over to the C++ compiler.
Some test code was written to perform the simple task of creating a Histogram, binning some values and storing it in a database. Objectivity allows for different object clustering schemes within databases based around “containers” (not to be confused with STL container classes). Hashing algorithms can be used for optimized access. At this stage, we simply used the default database containers. Here are the stages involved for creating a 2-D histogram:

- Start a transaction
- Obtain the desired database clustering handle
- Create an **ElementaryPartition** object for 1st Dimension
- Create an **ElementaryPartition** object for 2nd Dimension
- Create a **Partition** object
- Register **ElementaryPartition** objects with the **Partition** object
- Create a **DataHistogram** using the **Partition** object
- Fill the **DataHistogram**
- Close transaction

There are several steps involved, but one must be aware that the above example represents the most detailed case since all the associated objects are created from scratch. **Partition** and **Elementary-Partition** can be referenced by many client objects so if several histograms are created using similar dimensions and base types, steps can be omitted. It’s straightforward but it was felt more could be done to simplify matters. During many meetings, the subject of HBOOK was raised because its represents the bare minimum which the histogram package has to provide. This covered the API too.

Two new **DataHistogram** subclasses were created: **Histo_1D** and **Histo_2D**. They contain no new data members, just more convenient functions. Their constructors create 1D and 2D fixed width bin histograms respectively. They take care of **Partition** creation and provide a HBOOK style interface. A Session class provides a high level interface encapsulating the behavior of **Transaction**, simplifying the task of database access. The C++ Objectivity interface did not produce any unwelcome surprises. The API doesn't pose any problems for competent C++ programmers. Clearly, things were taken into consideration when writing persistent capable classes but the effect on the design was not overwhelming. Currently, the class library has been compiled for Silicon Graphics and HP-UX platforms.

5 Histogram Interfaces for Visualization

In reality, no vendor supplies a shrink-wrapped HEP visualization system ready to ship. An MVS (Modular Visualization System) is designed to allow a high degree of extensibility through code integration. A HEP tailored visualization package could include function fitting and minimalization through a MINUIT interface as well as tools to manipulate the persistent histograms. The code prototyping continued in conjunction with the evaluation of two leading MVS packages: AVS Express [8] and IRIS Explorer [9].

5.1 Modular as opposed to monolithic

MVS environments are a far cry from the unmaintainable "black box" designs of the past. The process model is now seen as being composed of discrete, re-useable functional units connected together to form a user application. Groups of modules can be encapsulated in order to derive new features. Typically, an off-the-shelf MVS environment will consist of general purpose modules. Categories include:

- File readers/writers
- Filters for extracting data subsets
- Transformers which apply functions to datasets
- 2-D and 3-D graphics toolkits

New modules can be built with existing code using an API provided by the MVS. Data is shared between modules through a variety of types designed to handle different forms of visualization information, from array-like datasets to graphics geometry. There is also the possibility to create user-defined types: The protoyping objectives were:

- Read in histograms
- Provide an interface to operations
- Render the histogram data

5.2 Share the histograms or database references?

A decision had to be made as to how histograms were to be shared between the prototype modules. The choice was either transient C++ classes or persistent database references. MVS packages make use of shared memory mechanisms to reduce the need for data copying. Support of shared C++ objects is of great importance but this would imply that the persisent object would have to be initially duplicated. The alternative was to share database references. Objectivity has a caching system for efficient object access but at the moment, the scope of each cache is limited per-process. The ideal scenario would be either a global, machine scope database cache or an MVS which allows many modules to be executed as a single process.
GUI building facilties allow widgets to be associated to module parameters. **HistogramReader** was the first module to be written, allowing persistent histograms to be read into the MVS using a browser front-end. Databases can be scanned for classes and their titles displayed in a scrolled list. Objectivity

class iterators are used to perform the scanning with the use of run-time type identification. Operator functions had been partially implemented and they were complemented with an MVS interface which allows binary operations to be performed interactively. In order to display the histogram data it was necessary to transform it into a suitable MVS datatype which could in turn be converted into geometry through graphics toolkit modules.
The result was the ability to visualize histograms in two or three dimensions in a variety of ways. These included traditional graph plots, contours and 3D rendered scenes, annotated with axis and legends. OpenGL [10] and Open Inventor [11] are widely regarded as the leading graphics libraries. They also provide standard means of accessing 3D scenes via the World-Wide-Web using the VRML [12] (Virtual Reality Modelling Language).

6 Standards and vendor collaboration

So if industry can give solutions, they are worth our consideration. CERN's program development should center around HEP unique functionality. A great deal of manpower can be saved by sharing the burden of software support but critics of the LHC++ approach raise two common questions:

- How long will vendors and their products last?
- Can you assure product interoperability?

Many developments will take place during the next ten years. Our aim should be to try and make future transitions as smooth as possible. LHC++ strategy involves building layered foundations, based on leading standards. But that isn't enough to guarantee compatibility between products originating from different vendors. The intention is not only to form strong links between CERN and vendors but to get vendors collaborating between themselves.

References

1 The LHC++ group, "Libraries for HEP Computing - LHC++", http://wwwcn1.cern.ch/asd/lhc++.html.
2 P. J. Plauger, *The Draft Standard C++ Library*, Prentice Hall, (1994).
3 M. Lee, A. Stepanov, *The Standard Template Library*, Hewlett-Packard, California, (1995).
4 The RD45 group, *RD45 - A Persistent Object Manager for HEP*, CERN/LHCC, Geneva, (1996).
5 Objectivity inc., *Choosing an ODBMS*, Objectivity, California, (1996).
6 The ODMG, *The Object Database Standard - ODMG-93*, ed. R. G. G. Cattell, Morgan Kaufmann.
7 Rogue Wave, "The Standard C++ Library and Tools.h++", http://www.roguewave.com/products/stdlib/stdlibtools.html.
8 Advanced Visual Systems, "AVS Express information and resources", http://www.avs.com.
9 The Numerical Algorithms Group, "IRIS Explorer Center", http://www.nag.co.uk:70/Welcome_IEC.html.
10 T. Davis, Jackie, Neider, M. Woo, *OpenGL Programming Guide: The Official Guide to Learning OpenGL*, Addison-Wesley, Massachusetts, (1993).
11 J. Wernecke, *The Inventor Mentor*, Addison-Wesley,Massachusetts, (1994).
12 Silicon Graphics, "The Virtual Reality Modeling Language Specification", http://vrml.sgi.com/moving-worlds/index.html.