ROOT: An Object-Oriented Data Analysis Framework

A report on a data analysis tool currently being developed at CERN.

Fons Rademakers & René Brun

Introduction

ROOT is a system for large scale data analysis and data mining. It is being developed for the analysis of Particle Physics data, but it can be equally well used in other fields where large amounts of data need to be processed.

Having had many years of experience in developing interactive data analysis systems like PAW and PIAF (see Resources), we realized that the growth and maintainability of these products, written in FORTRAN and using 20-year-old libraries, had reached its limits. Although still popular in the physics community, these systems do not scale up to the challenges offered by the next generation particle accelerator, the Large Hadron Collider (LHC), currently under construction at CERN, Geneva, Switzerland. The expected amount of data produced by the LHC will be on the order of several petabytes (1PB = 1,000,000GB) per year. This is two to three orders of magnitude more than what is being produced by the current generation of accelerators.

Therefore, in early 1995, we started the development of a system which should overcome the deficiencies of these previous programs. One of the first decisions we made was to follow the object-oriented analysis and design methodology and to use C++ as our implementation language. Although all of our previous programming experience was in FORTRAN, we soon realized the power of OO and C++, and after some initial "throw-away" prototyping the ROOT system began to take shape.

In November 1995, we gave a first public presentation of ROOT at CERN and, at the same time, version 0.5 was released via the Web. By then Nenad Buncic and Valery Fine had joined our team.

Since the initial release, there has been a constantly increasing number of users. In response to comments and feedback, we've been regularly releasing new versions containing bug fixes and new features. In January 1997, version 1.0 was released and in March 1998 version 2.0. Since the release of version 1.0 more than 16000 copies of the ROOT binaries have beendownloaded from our web site, about 700 people have registered as ROOT users, and the web site gets more than 150,000 hits per month.

ROOT is currently being used in many different fields: physics, astronomy, biology, genetics, finance, insurance, pharmaceutics, etc.

The source and binaries for many different platforms can be downloaded from the ROOT web site (`http://root.cern.ch/`). The current version can be used, modified and distributed freely as long as proper credit is given and copyright notices are maintained. For commercial use, the authors would like to be notified.

Main Features of ROOT

The main components of the ROOT system are:

- A hierarchical object-oriented database (machine independent, highly compressed, supporting schema evolution and object versioning)
- A C++ interpreter
- Advanced statistical analysis tools (classes for multi-dimensional histogramming, fitting and minimization)
- Visualization tools (classes for 2D and 3D graphics including an OpenGL interface)
- A rich set of container classes that are fully I/O aware (list, sorted list, map, btree, hashtable, object array, etc.)
- An extensive set of GUI classes (windows, buttons, combo-box, tabs, menus, item lists, icon box, tool bar, status bar and many more)
- An automatic HTML documentation generation facility
- Run-time object inspection capabilities
- Client/server networking classes
- Shared memory support
- Remote database access either via a special daemon or via the Apache web server
- Ported to all known Unix and Linux systems and also to Windows 95 and NT

The complete system consists of about 450,000 lines of C++ and 80,000 lines of C code. There are about 310 classes grouped in 24 different frameworks (each represented by their own shared library).

The CINT C/C++ Interpreter

One of the key components of the ROOT system is the CINT C/C++ interpreter. CINT, written by Masaharu Goto of Hewlett Packard Japan, covers 95% of ANSI C and about

85% of C++ (template support is being worked on, exceptions are still missing). CINT is complete enough to be able to interpret its own 70,000 lines of C and to let the interpreted interpreter interpret a small program.

The advantage of a C/C++ interpreter is that it allows for fast prototyping since it eliminates the typical time consuming edit/compile/link cycle. Once a script or program is finished, you can compile it with a standard C/C++ compiler (**gcc**) to machine code and enjoy full machine performance. Since CINT is very efficient (for example, for/while loops are byte-code compiled on the fly), it is quite possible to run small programs in the interpreter. In most cases, CINT out performs other interpreters like Perl and Python.

Existing C and C++ libraries can easily be interfaced to the interpreter. This is done by generating a dictionary from the function and class definitions. The dictionary provides CINT with all necessary information to be able to call functions, to create objects and to call member functions. A dictionary is easily generated by the program **rootcint** that uses as input the library header files and produces as output a C++ file containing the dictionary. You compile the dictionary and link it with the library code into a single shared library. At run time you dynamically link the shared library, and then you can call the library code via the interpreter. This can be a very convenient way to quickly test some specific library functions. Instead of having to write a small test program, you just call the functions directly from the interpreter prompt.

The CINT interpreter is fully embedded into the ROOT system. It allows the ROOT command line, scripting and programming languages to be identical. The embedded interpreter dictionaries provide the necessary information to automatically create GUI elements like context pop-up menus unique for each class and for the generation of fully hyperized HTML class documentation. Further, the dictionary information provides complete run-time type information (RTTI) and run-time object introspection capabilities.

Installation

The binaries and sources of ROOT can be downloaded from `http://root.cern.ch/root/Version200.html`. After downloading, uncompress and unarchive (using **tar**) the file `root_v2.00.Linux.2.0.33.tar.gz` in your home directory (or in a system-wide location like `/opt`). This procedure will produce the directory `./root`. This directory contains the following files and sub directories:

- `AA_README`: read this file before starting
- `bin`: directory containing executables
- `include`: directory containing the ROOT header files
- `lib`: directory containing the ROOT libraries (in shared library format)
- `macros`: directory containing system macros (e.g., GL.C to load OpenGL libs)

- `icons`: directory containing xpm icons
- `test`: some ROOT test programs
- `tutorials`: example macros that can be executed by the bin/root module

Before using the system, you must set the environment variable ROOTSYS to the root directory, e.g., export ROOTSYS=/home/rdm/root, and you must add $ROOTSYS/bin to your path. Once done, you are all set to start rooting.

First Interactive Session

In this first session, start the ROOT interactive program **root**. This program gives access via a command-line prompt to all available ROOT classes. By typing C++ statements at the prompt you can create objects, call functions, execute scripts, etc. Go to the directory `$ROOTSYS/tutorials` and type:

```
bash$ root
root [0] 1+sqrt(9)
(double)4.000000000000e+00
root [1] for (int i = 0; i <\<> 5; i++)
printf("Hello %d\n", i)
Hello 0
Hello 1
Hello 2
Hello 3
Hello 4
root [2] .q
bash $
```

As you can see, if you know C or C++, you can use ROOT. No new command-line or scripting language to learn. To exit use `.q`, which is one of the few "raw" interpreter commands. The dot is the interpreter escape symbol. There are also some dot commands to debug scripts (step, step over, set breakpoint, etc.) or to load and execute scripts.

Let's now try something more interesting. Again, start root:

```
bash$ root
root [0] TF1 f1("func1","sin(x)/x", 0, 10)
root [1] f1.Draw()
root [2] f1.Integral(0,2)
root [3] f1.Dump()
root [4] .q
```

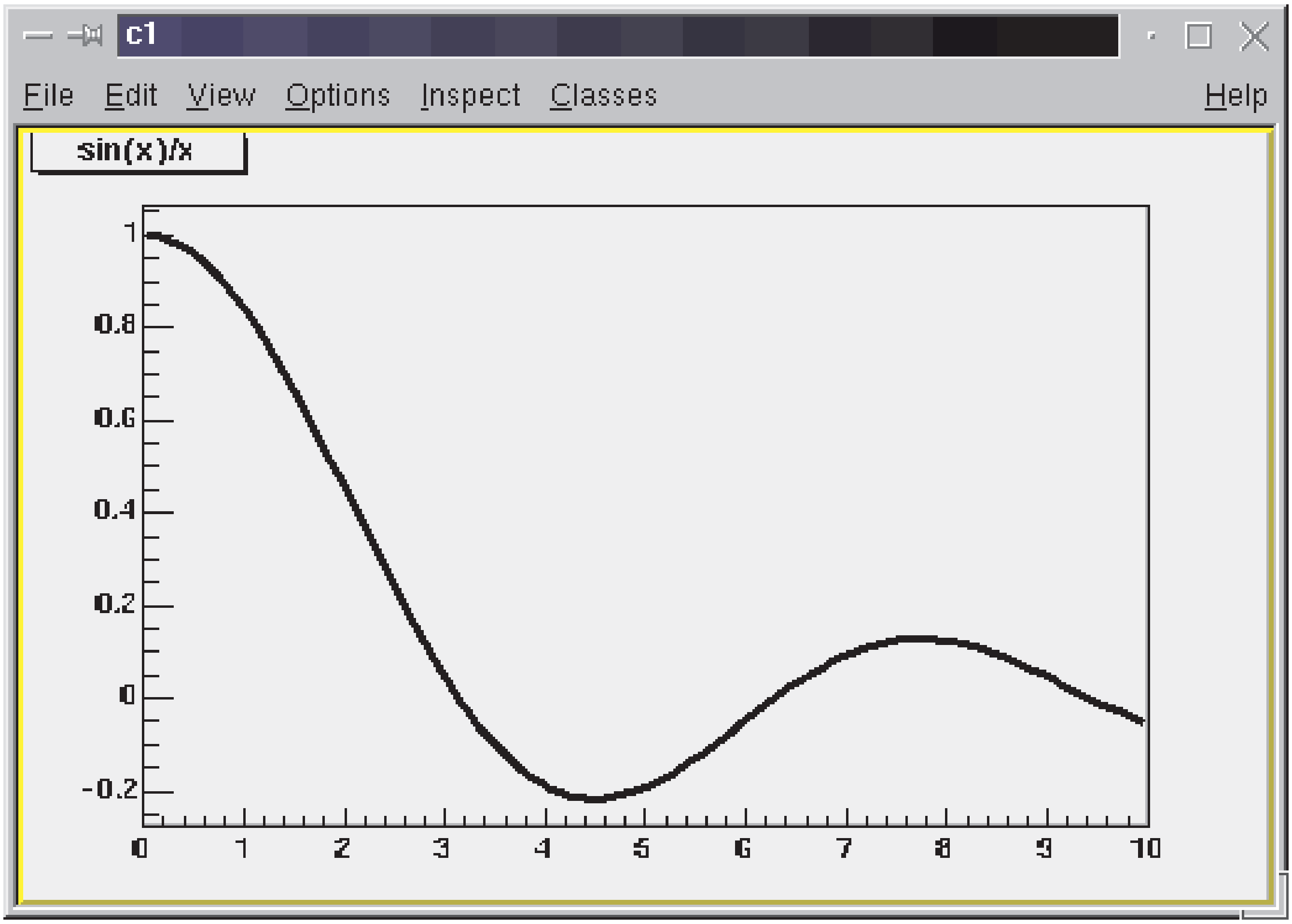

Figure 1. Output of `f1.Draw()`

Here you create an object of class `TF1`, a one-dimensional function. In the constructor you specify a name for the object (which is used if the object is stored in a database), the function and the upper and lower value of x. After having created the function object you can, for example, draw the object by executing the **TF1::Draw()** member function. Figure 1 shows how this function looks. Now, move the mouse over the picture and see how the shape of the cursor changes whenever you cross an object. At any point, you can press the right mouse button to pop-up a context menu showing the available member functions for the current object. For example, move the cursor over the function so that it becomes a pointing finger, and then press the right button. The context menu shows the class and name of the object. Select item `SetRange` and put `10, 10` in the dialog box fields. (This is equivalent to executing the member function `f1.SetRange(10,10)` from the command-line prompt, followed by `f1.Draw()`.) Using the **Dump()** member function (that each ROOT class inherits from the basic ROOT class `TObject`), you can see the complete state of the current object in memory. The **Integral()** function shows the function integral between the specified limits.

Histogramming and Fitting

Let's start root again and run the following two macros:

```
bash$ root
root [0] .x hsimple.C
root [1] .x ntuple1.C
 // interact with the pictures in the canvas
root [2] .q
```

Note: if the above doesn't work, make sure you are in the tutorials directory.

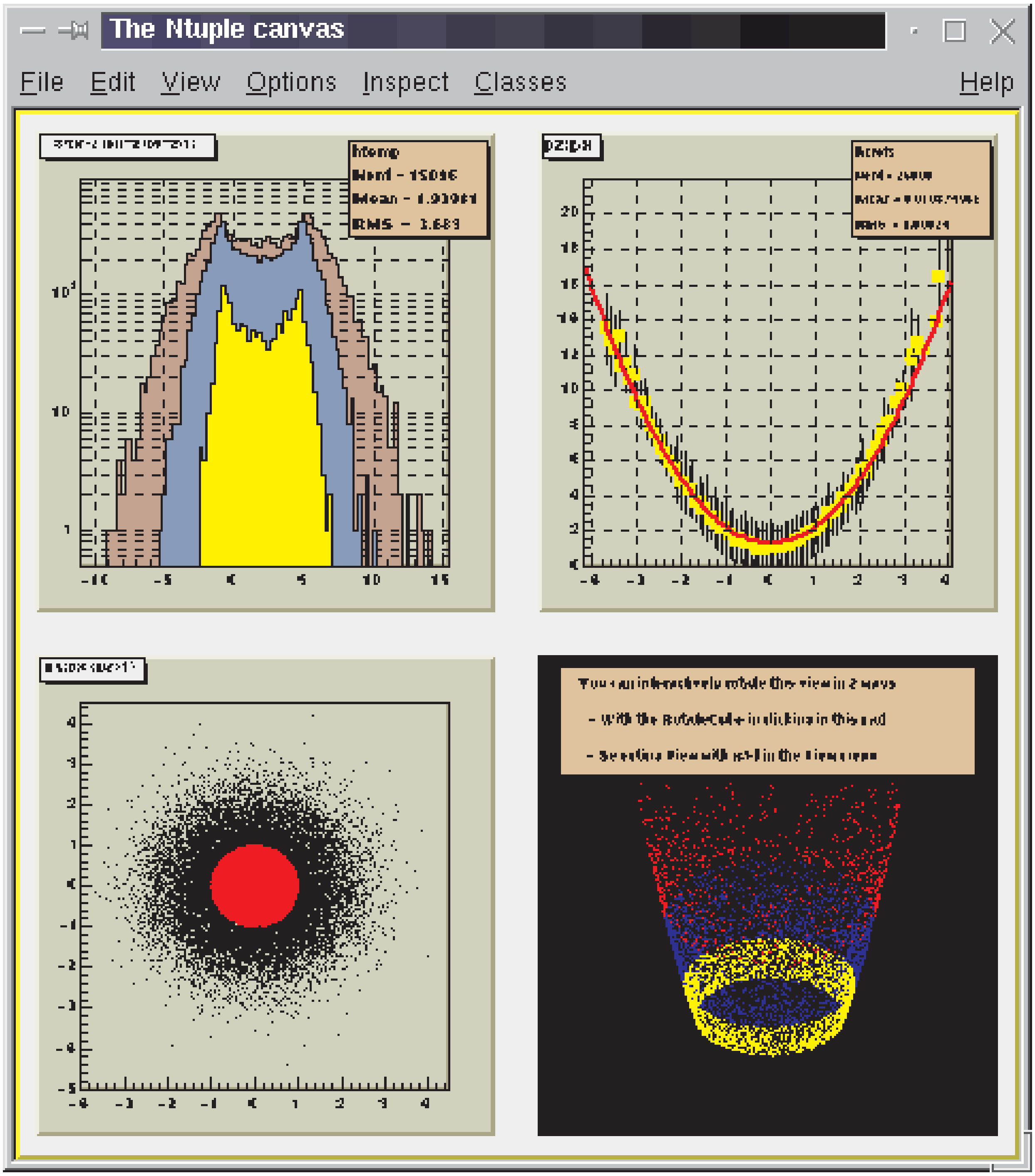

Figure 2. Output of `ntuple1.C`

Macro `hsimple.C` (see `$ROOTSYS/tutorials/hsimple.C`) creates some 1D and 2D histograms and an Ntuple object. (An Ntuple is a collection of tuples; a tuple is a set of numbers.) The histograms and Ntuple are filled with random numbers by executing a loop 25,000 times. During the filling the 1D histogram is drawn in a canvas and updated each 1,000 fills. At the end of the macro the histogram and Ntuple objects are stored in a ROOT database.

The `ntuple1.C` macro uses the database created in the previous macro.It creates a canvas object and four graphics pads. In each of the four pads a distribution of different Ntuple quantities is drawn. Typically, data analysis is done by drawing in a histogram with one of the tuple quantities when some of the other quantities pass a certain condition. For example, our Ntuple contains the quantities `px`, `py`, `pz`, random and `i`. The command:

```
ntuple->Draw("px", "pz< 1")
```

will fill a histogram containing the distribution of the `px` values for all tuples for which `pz < 1`. Substitute for the abstract quantities used in this example quantities like, name, sex, age, length, etc., and you can easily understand that Ntuples can be used in many different ways. An Ntuple of 25,000 tuples is quite small. In typical physics analysis situations Ntuples can contain many millions of tuples, each consisting of several hundred numbers. Besides the simple Ntuple, the ROOT system also provides a Tree. A Tree is an Ntuple generalized to complete objects. That is, instead of sets of tuples, a Tree can store sets of objects. Objects may be complex hierarchical or graph structures. A Tree is organized in branches. Branches have leaves. The object attributes, the leaves, can be analyzed like the tuple quantities. Only branches referenced in a query are read into memory. Trees are designed to support large files (GBs). Trees can be logically grouped into Chains to create multi terabyte databases. For more information on Trees see the ROOT HOWTOs at `http://root.cern.ch/root/Howto.html`.

During data analysis you often need to test the data with a hypothesis. A hypothesis is a theoretical/empirical function that describes a model. To see if the data matches the model, you use minimization techniques to tune the model parameters so that the function best matches the data, this is called fitting. ROOT allows you to fit standard functions, like polynomials, Gaussians, exponentials or custom defined functions to your data. In the top right pad in Figure 2, the data has been fit with a polynomial of degree two (red curve). This was done by calling the **`Fit()`** member function of the histogram object:

```
hprofs->Fit("pol2")
```

Moving the cursor over the canvas allows you to interact with the different objects. For example, the 3D plot in the lower-right corner can be rotated by clicking the left-mouse button and moving the cursor.

The GUI Classes and Object Browser

Embedded in the ROOT system is an extensive set of GUI classes. The GUI classes provide a full OO-GUI framework as opposed to a simple wrapper around a GUI such as Motif. All GUI elements do their drawing via the `TGXW` low-level graphics abstract base class. Depending on the platform on which you run ROOT, the concrete graphics class (inheriting from `TGXW`) is either `TGX11` or `TGWin32`. All GUI widgets are created from "first principles", i.e., they only use routines like **DrawLine**, **FillRectangle**,

CopyPixmap, etc., and therefore, the `TGX11` implementation only needs the X11 and Xpm libraries. The advantage of the abstract base class approach is that porting the GUI classes to a new, non X11/Win32, platform requires only the implementation of an appropriate version of `TGXW` (and of `TSystem` for the OS interface).

All GUI classes are fully scriptable and accessible via the interpreter. This allows for fast prototyping of widget layouts.

The GUI classes are based on the XClass'95 library written by David Barth and Hector Peraza. The widgets have the well known Windows 95 look and feel. For more information on XClass'95, see `ftp://mitac11.uia.ac.be/html-test/xclass.html`.

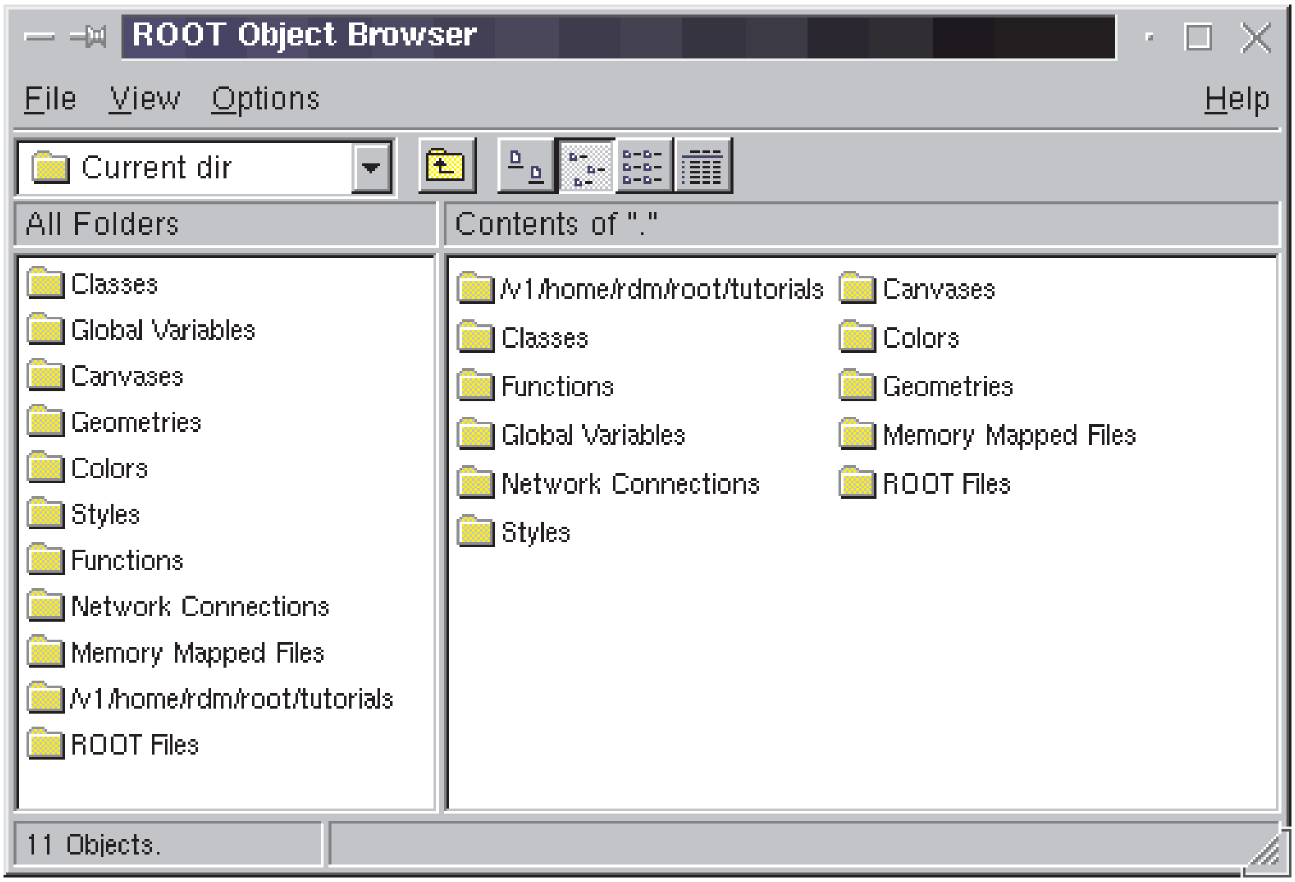

Figure 3. ROOT Object Browser

Using the ROOT Object Browser all objects in the ROOT system can be browsed and inspected. To create a browser object, type:

```
root [0] new TBrowser
```

The browser, as shown in Figure 3, displays in the left pane the browsable ROOT collections and in the right pane the objects in the selected collection. Double clicking on an object will execute a default action associated with the class of the object. Double clicking on a histogram object will draw the histogram. Double clicking on an Ntuple quantity will produce a histogram showing the distribution of the quantity by looping over all tuples in the Ntuple. Right clicking on an object will bring up a context menu (just as in a canvas).

Integrating Your Own Classes into ROOT

```
#ifndef __PERSON_H
#define __PERSON_H

#include <TObject.h>

class Person : public TObject { // need to inherit from TObject

private:
   int age;      // age of person
   float height; // height of person

public:
   Person(int a = 0, float h = 0) : age(a), height(h) { }
   int get_age(void) const { return age; }
   float get_height(void) const { return height; }

   void set_age(int a) { age = a; }
   void set_height(float h) { height = h; }

   ClassDef(Person,1) // Person class
};

#endif
```

Listing 1. Class describing Person attributes.

In this section we'll give a step-by-step method for integrating your own classes into ROOT. Once integrated you can save instances of your class in a ROOT database, inspect objects at run-time, create and manipulate objects via the interpreter, generate HTML documentation, etc. A very simple class describing some person attributes is shown in Listing 1. The Person implementation file Person.cxx is shown in Listing 2.

```
#include "Person.h"
// ClassImp provides the implementation of some
// functions defined in the ClassDef macro
ClassImp(Person)
```

Listing 2. Implementation File Person.cxx

The macros **ClassDef** and **ClassImp** provide some member functions that allow a class to access its interpreter dictionary information. Inheritance from the ROOT basic object, `TObject`, provides the interface to the database and introspection services.

Now run the **rootcint** program to create a dictionary, including the special I/O streamer and introspection methods for class `Person`:

```
bash$ rootcint -f dict.cxx -c Person.h
```

Next compile and link the source of the class and the dictionary into a single shared library:

```
bash$ g++ -fPIC -I$ROOTSYS/include -c dict.cxx
bash$ g++ -fPIC -I$ROOTSYS/include -c Person.cxx
bash$ g++ -shared -o Person.so Person.o dict.o
```

Now start the ROOT interactive program and see how we can create and manipulate objects of class Person using the CINT C++ interpreter:

```
bash$ root
root [0] gSystem->Load("Person")
root [1] Person rdm(37, 181.0)
root [2] rdm.get_age()
(int)37
root [3] rdm.get_height()
(float)1.810000000000e+02
root [4] TFile db("test.root","new")
root [5] rdm.Write("rdm") // Write is inherited from theTObject class
root [6] db.ls()
TFile** test.root
 TFile* test.root
 KEY: Person rdm;1
root [7] .q
```

Here the key statement was the command to dynamically load the shared library containing the code of your class and the class dictionary.

In the next session we access the **rdm** object we just stored on the database `test.root`:

```
bash$ root
root [0] gSystem->Load("Person")
root [1] TFile db("test.root")
root [2] rdm->get_age()
(int)37
root [3] rdm->Dump() // Dump is inherited from the TObjectclass
age            37          age of person
height         181         height of person
fUniqueID      0           object unique identifier
fBits          50331648    bit field status word
root [4] .class Person
[follows listing of full dictionary of class Person]
root [5] .q
```

```
//----------- begin fill.C
{
   gSystem->Load("Person");
   TFile *f = new TFile("test.root","recreate");
   Person *t;
   for (int i = 0; i < 1000; i++) {
      char s[10];
      t = new Person(i, i+10000);
      sprintf(s, "t%d", i);
      t->Write(s);
      delete t;
   }
   f->Close();
}
```

Listing 3. C++ Macro for Storing Database Information

A C++ macro that creates and stores 1000 persons in a database is shown in Listing 3. To execute this macro, do the following:

```
bash$ root
root [0] .x fill.C
root [1] .q
```

This method of storing objects would be used only for several thousands of objects. The special Tree object containers should be used to store many millions of objects of the same class.

```
//----------- begin find.C
void find(int begin_age = 0, int end_age = 10)
{
   gSystem->Load("Person");
   TFile *f = new TFile("test.root");
   TIter next(f->GetListOfKeys());
   TKey *key;
   while (key = (TKey*)next()) {
      Person *t = (Person *) key->Read();
      if (t->get_age() >= begin_age && t->get_age() <= end_age) {
         printf("age = %d, height = %f\n", t->get_age(), t->get_height());
      }
      delete t;
   }
}
```

Listing 4. Database Query Macro

Listing 4 is a C++ macro that queries the database and prints all persons in a certain age bracket. To execute this macro do the following:

```
bash$ root
root [0] .x find.C(77,80)
age = 77, height = 10077.000000
age = 78, height = 10078.000000
age = 79, height = 10079.000000
age = 80, height = 10080.000000
NULL
root [1] find(888,895)
age = 888, height = 10888.000000
age = 889, height = 10889.000000
age = 890, height = 10890.000000
age = 891, height = 10891.000000
age = 892, height = 10892.000000
age = 893, height = 10893.000000
age = 894, height = 10894.000000
age = 895, height = 10895.000000
root [2] .q
```

With Person objects stored in a Tree, this kind of analysis can be done in a single command.

```
//------------ begin method.C
{
   gSystem->Load("Person");
   TClass *c = gROOT->GetClass("Person");
   TList *lm = c->GetListOfMethods();
   TIter next(lm);
   TMethod *m;
   while (m = (TMethod *)next()) {
     printf("%s %s%s\n", m->GetReturnTypeName(), m->GetName(),
             m->GetSignature());
   }
}
```

Listing 5. Print Macro

Finally, a small C++ macro that prints all methods defined in class `Person` using the information stored in the dictionary is shown in Listing 5. To execute this macro, type:

```
bash$ root
root [0] .x method.C
class Person Person(int a = 0, float h = 0)
int get_age()
float get_height()
void set_age(int a)
void set_height(float h)
const char* DeclFileName()
int DeclFileLine()
const char* ImplFileName()
int ImplFileLine()
Version_t Class_Version()
class TClass* Class()
void Dictionary()
class TClass* IsA()
void ShowMembers(class TMemberInspector& insp, char* parent)
void Streamer(class TBuffer& b)
class Person Person(class Person&)
void ~Person()
root [1] .q
```

The above examples prove the functionality that can be obtained when you integrate, with a few simple steps, your classes into the ROOT framework.

Linux an Increasing Force in Scientific Computing

Analysis of the FTP logs of the more than 16000 downloads of the ROOT binaries reveals the popularity of the different computing platforms in the mainly scientific community. Figure 4 shows the number of ROOT binaries downloaded per platform.

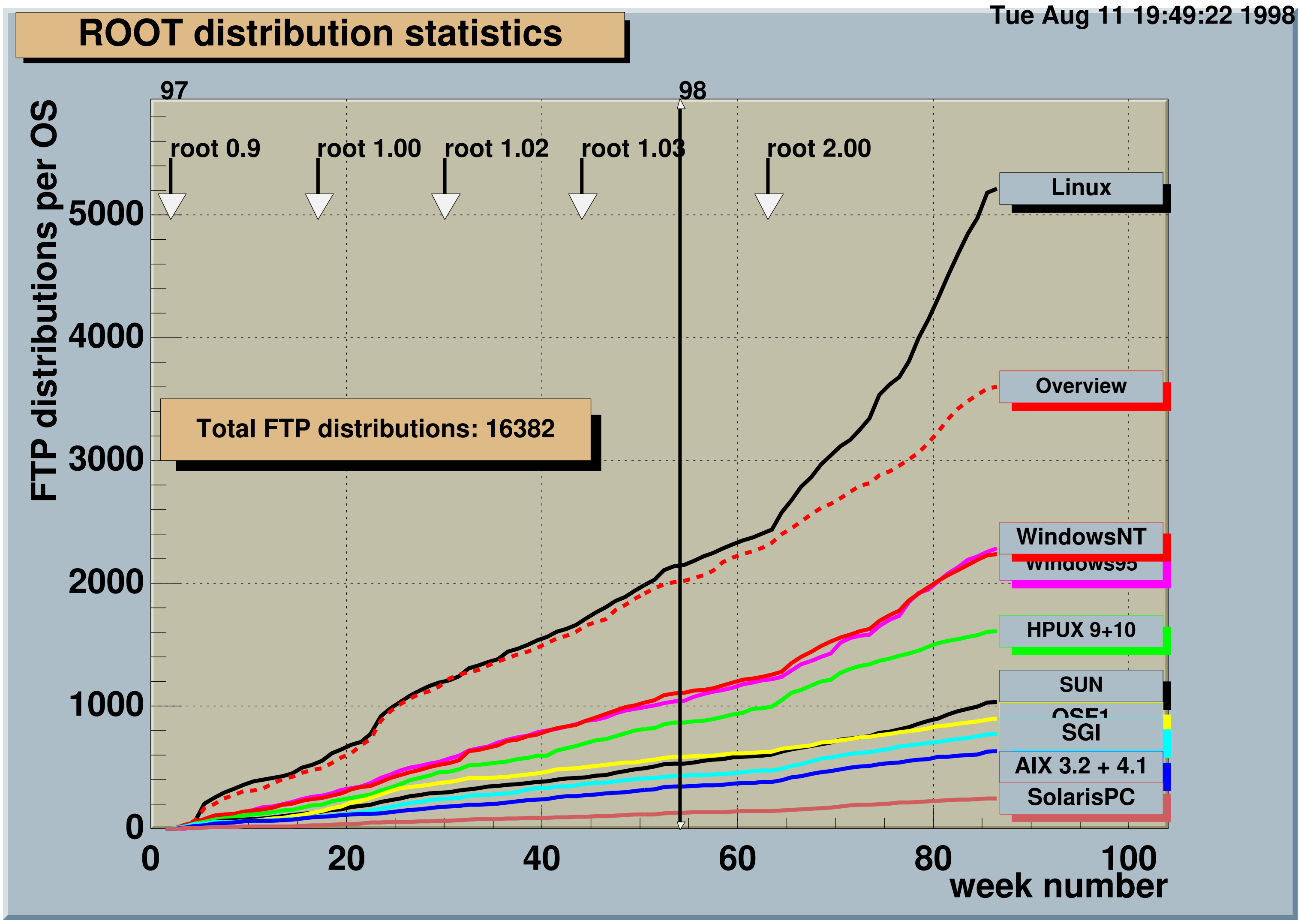

Figure 4. ROOT Download Statistics

Linux is the clear leader. Followed by the Microsoft platforms (Windows 95 and NT together are still less than Linux). The results for the other Unix machines should probably be corrected a bit since many machines are multi-user machines where a single download by a system manager will cover more than one user. Linux and Windows are typical single-user environments.

Summary

In this article we've given an overview of some of the main features of the ROOT data-handling system. However, many aspects and features of the system remained uncovered, such as the client/server classes (the `TSocket`, `TServerSocket`, `TMonitor` and `TMessage` classes), how to automatically generate HTML documentation (using the `THtml` class), remote database access (via the `rootd` daemon), advanced 3D graphics, etc. More on all these topics can be found on the ROOT web site.

Resources

PAW: Physics Analysis Workstation, CERN Program Library.

PIAF: Parallel Interactive Analysis Facility, CERN Program Library.

ROOT: `http://root.cern.ch/`

XClass'95: `ftp://mitac11.uia.ac.be/html-test/xclass.html`

To follow what goes on in the ROOT community, there is an active mailing list on which all aspects of the system are discussed. The mailing list can be joined by sending a mail containing "subscribe roottalk" to `majordomo@root.cern.ch`.

Acknowledgements

The ROOT project would not be possible without the enthusiastic users who steer the direction of the development with their suggestions and comments.