Diploma Thesis
ruby-root
Extending ROOT's functionality with a Ruby interpreter interface

Elias Athanasopoulos*
elathan@phys.uoa.gr

UA/PHYS/HEP/2-2-2005

*University Of Athens, HEPA Lab

Contents

Abstract

ruby-root aims on providing Ruby bindings for the ROOT Object Oriented Framework. ROOT is a very popular software solution for data analysis in the field of High Energy Physics. Using ruby-root you can have the basic functionality ROOT provides via Ruby; a powerful modern scripting language.

1 Fundamental Introduction

1.1 High Energy Physics

High Energy Physics (HEP) is considered the most active field in Experimental Physics. The aim of HEP is to identify the nature of the fundamental forces and particles of our universe. The basic concept of particle physics experiments is the acceleration of particles (protons, electrons, etc) which then collide with each other. The collision can explode the internal structure of the particles, produce sub particles and give us a better picture of nature's structure at the fundamental level. The energy of the colliding particles is critical, since higher energy means stronger collision and thus a more detailed picture of the internal particles' structure. This is the main reason we use the term 'High Energy'.

The final goal of HEP is to construct a theory for the description of all the elementary particles and elementary forces of our world. This theory is also called 'The Standard Model' (see Figure 1). Currently, the standard model contains 12 elementary particles and we have knowledge about 4 fundamental interactions. The 12 elementary particles are the 6 quarks, the electron and its neutrino, the muon and its neutrino and the tau and its neutrino. The latter, the neutrino of the tau particle, was discovered in the Donut experiment.

Last but not least, there is a crucial effort for the discovery of the Higgs particle, known also as the mass carrier. The discovery of the Higgs particle will give us strong feelings that the Standard Model's theory is correct.

1.2 Neutrino Physics

Neutrinos are quite strange particles in the sense that their difficulty to be observed gives them some interesting properties. Neutrinos can take part

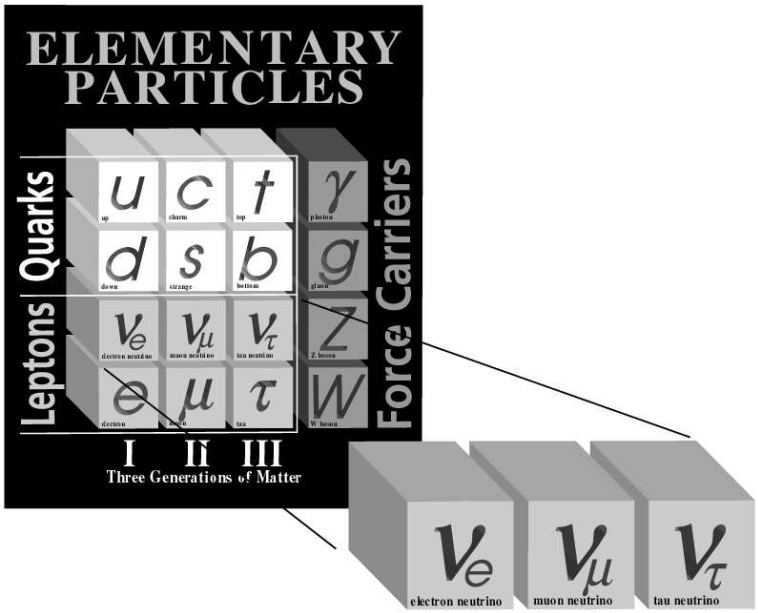

Figure 1: The Standard Model

only in weak interactions, such as the well known β-decay. Actually, β-decay was the first nuclear process that guide the science community to the discovery of the neutrino particle. During a β-decay a proton becomes a neutron (or vice versa) and an electron (or positron) is emitted. In order to have momentum conservation, since the electron's spectrum is continuous, another particle must also be emitted. That is the neutrino particle (or an antineutrino in the case where a bound neutron becomes a proton).

Since β-decay is a process leaded by the weak interaction and since neutrinos can interact only in weak interaction processes, we have a major difficulty to observe and measure neutrino's properties. For example, there is a crucial debate about neutrino's mass. Since, we have no way to measure explicitly, but rather implicitly via the electron's properties emitted in a β-decay, a neutrino's mass, the only thing we can accomplish is to define some mass limits. We know that neutrino's mass is very small, or even zero, but, there is no strong theoretical reason for the neutrino's mass to be zero. On the other hand, there is a strong theoretical reason for the photon's mass to be zero.

R	Experiment
0.60 +- 0.05	Kamiokande (sub-GeV)
0.57 +- 0.07	Kamiokande (multi-GeV)
0.63 +- 0.03	Super-Kamiokande (sub-GeV)
0.65 +- 0.05	Super-Kamiokande (multi-GeV)

Table 1: The ratio R as verified by Kamiokande and Super-Kamiokande[1].

1.3 Neutrino Oscillations

One of the great mysteries in neutrino physics was the anomaly of the atmospheric neutrino's flux. Large experiments, such as Kamiokande and Super-Kamiokande, were developed in order to measure the ratio of μ-like to e-like events from cosmic rays. In order to verify our theory about the flux of neutrinos from cosmic rays, theoretical Monte-Carlo calculations was performed. So, the goal of the experiments was to verify the quantity given by: $R = (\mu/e)_{data}/(\mu/e)_{MC}$. That is the ratio of the observable events over the theoretical calculations.

As it was manifestly shown by Kamiokande and Super-Kamiokande (see Table 1) the ratio is smaller than 1. That is there is an anomaly in the atmospheric neutrino's flux and a new concept must be introduced: neutrino oscillations. That is, neutrinos have the property to interchange themselves. Thus, neutrinos of one of the three available flavors (namely ν_e, ν_μ and ν_τ) can oscillate (can change) to another flavor.

Now, if we adapt the neutrino oscillations concept, there is a strong argument that neutrinos can't be massless, because massless particles cannot oscillate. Put another way, observation of oscillation implies that the masses of the neutrinos involved cannot be equal to one another. Since they cannot be equal to one another, they cannot both be zero. In fact it is quite likely that if any neutrinos have non-zero mass, all of them do.

2 Data Analysis in HEP

2.1 Introduction to ROOT

ROOT[1] is a collection of libraries, implemented in C++, which aims to provide a complete solution for various scientific tasks such as data analysis for large experiments in the field of High Energy Physics.

Among other experiments, ROOT is heavily used in MINOS[2] experiment, in various projects at FNAL[3] and at SLAC[4].

ROOT delivers more than 700 different C++ Classes, ideal for Linear Algebra, Function Plotting and Fitting, Histogram Presentation and many more. In addition ROOT provides Classes for system-oriented tasks, such as GUI widgets, Database Connectivity, Networking facilities and others.

All the above transform ROOT to a complete framework for application building in C++. However, programming in C++ is not always trivial. Thus, ROOT comes with a tightly integrated C/C++ Interpreter, CINT[5]. CINT can interprete, or compile if speed is an issue, scripts written in C/C++. CINT can evaluate C/C++ code at run-time and resolve Class information using a Class dictionary. The latter stands for a special file, which describes a Class' behavior.

ROOT comes with special tools for generating at will Class dictionaries. Thus, every Class which has the appropriate generated dictionary can be accessible by any C/C++ script, executed via CINT. In addition, the dictionary technology enhances the ROOT framework with full RTTI (Run-Time Type Information) support. There are ROOT Classes (TClass, TMethod to name a few) which provide the user information about a Class' behavior, inheritance tree, member functions available, etc. This feature is vital in the development of ruby-root.

[1] `http://root.cern.ch`

[2] `http://www-numi.fnal.gov/offline_software/srt_public_context/WebDocs/Companion/index.html`

[3] `http://www-cpd.fnal.gov/CPD/root/`

[4] `http://www.slac.stanford.edu/BFROOT/www/doc/workbook/workbook.html`

[5] `http://root.cern.ch/root/Cint.html`

3 The MINOS Experiment

The MINOS (Main Injector Neutrino Oscillation Search) is a first generation, long baseline neutrino oscillation experiment. MINOS is designed to make a precise study of the "atmospheric" neutrino oscillations observed recently by underground experiments. That is, the main purpose of the MINOS experiment is to identify neutrino oscillations and if so to measure with great precision the oscillations parameters.

3.1 MINOS Architecture

MINOS consists of two detectors, namely the Near one and the Far one detector and it uses the NuMI neutrino beam. The two detectors are located at distance of 1 km and 735 km from the neutrino source, respectively. The Near detector weights 980 tons, while the Far one weights 5400 tons[2].

The MINOS experiment will use neutrinos produced in the NuMI beam line by 120 GeV protons. The beam is generated by the Main Injector at Fermilab in a fast extraction mode (10 μs). The proton beam is aimed at the Soudan mine in northern Minnesota, where the Far detector is located. Because of the earth's curvature the parent hadron beam has to be pointed at an angle of 57 mrad.

The resulting hadron beam is focused by specially designed focusing elements. It travels via a two-magnetic horn system followed by a 700 m long decay pipe and muon absorber to produce finally the ν_μ beam. In more detail, the hadron beam is focused and transported through an evacuated decay pipe, 1 m in radius and 675 m long, before striking a secondary hadron absorber downstream. The total decay length is 725 m. The dolomite between the hadron absorber and the Near detector provides sufficient shielding to range out all the muons produced by π and K in the beam pipe.

As already has been stated, the MINOS experiment utilizes two detectors with the basic structure of a segmented iron-scintillator calorimeter and magnetized muon spectrometer. The use of two detectors is dictated by the need to measure neutrino disappearance and to control systematics, while the primary function of the near detector is to serve as a reference for the main MINOS detector; the far one.

The fact that both the near and far detector have been constructed as similar as possible is vital. That is, MINOS tries to measure the neutrinos' behavior before (Near) and after (Far) they have the chance to oscillate. The

Parameter	Value
Near detector mass	0.98 (metric) kt total, 0.1 kt fiducial
Far detector mass (2 supermodules)	5.4 (metric) kt total, 3.3 kt fiducial
Steel planes (far detector)	8-m wide, 2.54-cm thick octagons
Magnetic field (far detector)	Toroidal, 1.5 T at 2 m radius
Active detector planes	Extruded polystyrene scintillator strips
Active detector strips	4.1-cm wide, 1-cm thick, $\sim$ 8-m long
Near detector distance from decay pipe	290 m
Far detector distance from decay pipe	730 km
Cosmic ray rates	270 Hz in near det., 1 Hz in far det.
Neutrino energy range (3 configurations)	1 to 25 GeV
Detector energy scale calibration	5% absolute, 2% near-far
Detector EM energy scale calibration	$23\%/\sqrt{(E)}$ ($< 5\%$ constant term)
Detector hadron energy resolution	$60\%/\sqrt{(E)}$ ($< 7\%$ constant term)
Detector muon energy resolution	$< 12\%$ (from curvature or range)
NC-CC event separation	Efficiency $> 90\%$, correctable to 99.5%
Electron/π separation	Hadron rejection $\sim 10^3$ for $\epsilon_e \sim 20\%$
Far det. ν event rate (high-energy beam)	3000 ν_μ CC events/kt/yr (no oscillations)
Near det. ν event rate (high-energy beam)	20 events/spill in target region
Near-far relative rate uncertainty	20%

Table 2: MINOS experimental parameters with the wide-band (PH2) beam[1].

results of the two detectors then can be compared to see if oscillations have been occurred. Having as a reference a second, almost similar technically, detector (Near) the results of the main Far detector may not be compared with Monte Carlo predictions but with real data, which is one of the great advantages of MINOS.

Table 1 summarizes some technical properties of the MINOS architecture.

3.2 Aims and Goals

Briefly, the physics goals of MINOS[1] are:

a) If Nature has chosen not to have neutrino oscillations in the parameter space accessible to MINOS, we want to be able to demonstrate this fact

convincingly over as large an area in oscillation parameter space as possible.

b) If oscillations do exist in the space accessible to MINOS, we want to convincingly demonstrate their existence, measure the oscillation parameters with high precision, and determine the oscillation modes. Specifically, we want to ensure that we can cover the full region of parameter space suggested by Super-Kamiokande experiment.

3.3 MINOS Software

MINOS is a demanding modern experiment in High Energy Physics and a rich framework for data analysis must be used in the construction of the MINOS applications. Thus, the MINOS software group decided to build the whole software for MINOS data analysis using the ROOT framework. Although the great majority of the MINOS source code which has already been implemented and will probably be extended in the near future is written in C++, ROOT scripts in a pure scripting language, such as Ruby or Python, might be very handy, especially in every-day jobs. Tasks as it is the collection of data from remote hosts or data analysis test-cases can easily developed using Ruby or Python. Since, ROOT's support for Ruby and Python is native, scripts written in Ruby or Python can interact with major C++ applications and exchange data.

Also, the ability of writing pluggins in Ruby or Python for the central MINOS Software Unit if a specific plugin-technology is developed must be examined.

4 Extending ROOT functionality

4.1 Introduction to Ruby

Ruby[6] is a modern scripting language with over 10 years of development. Ruby combines features from Python[7], Perl[8] and Smalltalk[9], as well as modern ideas from the field of Programming Languages. Ruby delivers a clean syntax and a full dynamic and Object Oriented nature. Using Ruby one can

[6] `http://www.ruby-lang.org`
[7] `http://www.python.org`
[8] `http://www.perl.org`
[9] `http://www.smalltalk.org`

write easily few-lines scripts that can cope with complex system tasks, such as opening files and manipulating their contents, matching patterns with regular expressions, connecting to services using sockets and many more.

Ruby is full dynamic typed, so the user is not dealing with complex definitions and prototyping. Its syntax is quite human-oriented, so as users with minimal Computer Science experience can learn its syntax and grammar very easily. In addition, Ruby embeds some powerful pre-built structures, like Arrays, Hashes and Strings, making programming quite productive.

The idea of extending Ruby and exporting ROOT's basic functionality to Ruby looked ideal and thus the idea of ruby-root was born.

4.2 Introduction to ruby-root

ruby-root[10] is a compact solution for bridging Ruby and ROOT's basic functionality. The use of ruby-root offers the user with the ability to write native Ruby scripts that can take advantages of the components that ROOT's libraries provide.

Not all of the ROOT's features are available to Ruby via ruby-root, but the most vital ones that a scientist depends on. Thus, most of the Computer-related parts of ROOT are not touched, since Ruby provides a rich set of similar functions; available in a native Ruby fashion.

On the other hand, as it will be shown in the next sections, fundamental ROOT constructs such as Classes dealing with Collections of Objects and Strings, to name a few, are converted internally in Ruby constructs, which can then manipulated by a user quite easier.

4.3 Installing ruby-root

In order to use ruby-root you must have installed Ruby and ROOT in your system. Assuming you have downloaded ruby-root and you have uncompressed it in a place of your choice, the installation process is the following:

```
% ruby ./extconf.rb
% make
# make install
```

[10] `http://null.edunet.uoa.gr/~elathan/rr/`

Keep in mind that the last step needs root privileges.

Since ruby-root comes with already wrapped ROOT classes of a specific ROOT version, you may have conflicts and failures in the compilation process. This means that you must rebuild ruby-root. The rebuilding process is the following:

```
% ./rebuild
% make
# make install
```

Rebuilding requires indent(1) installed in your system; a GNU tool to format the autogenerated C++ code.

4.4 Using ruby-root

Since ruby-root is correctly installed in your system, you can start developing scripts in Ruby which utilize ROOT's functionality. The following is a ruby-root classic 'Hello World' program:

```
require 'root'

tapp = TApplication.new "rr: Hello ROOT!"

tc = TCanvas.new "tc", "Hello", 168, 8, 699, 499
pt = TPaveText.new 0.1, 0.1, 0.5, 0.5, "blNDC"
pt.AddText 0, 0, "Hello ROOT from Ruby! :-)"
pt.Draw

tc.Modified
tc.cd

tapp.Run
```

Assuming the source file is named 'hello.rb', you can run it using the command:

```
% ruby hello.rb
```

5 Understanding ruby-root internals

5.1 Extending Ruby

Ruby is written in C. A rich C API (Application Programming Interface) exists, in order to extend Ruby with support of external functionality. By using the Ruby C API the creation of a library with C code known as 'Ruby extension' is straight forward. Whenever the user wants to use the new extension, he/she must use the 'require' command to load the library in a Ruby script. After the 'require' command the script is full aware of the new functionality. Actually this the reason that our ruby-root script, in the previous section had as the first line:

```
require 'root'
```

This line instructs Ruby to load the ruby-root extension, so as to use ROOT's functionality via Ruby.

Now, in order to extend Ruby with ROOT functionality, each ROOT class must be wrapped in Ruby using the Ruby C API. That is, for every C++ member function of a ROOT class, a C function must be created in order to call this member function in a fashion Ruby supports. This C function uses constructs that Ruby supports. Its main purpose, as we will see later in more details, is to bridge a Ruby method with a C++ ROOT method.

Also, some extra C code must be implemented for the creation of Ruby classes which encapsulate the ROOT classes behavior.

The following example shows the wrapped TCanvas::cd() member function. The arguments are translated to the equivalent C ones, and the TCanvas::cd() is called.

```
static VALUE cTCanvas_cd(int argc, VALUE argv[], VALUE self)
{
/* void TCanvas::cd(Int_t subpadnumber=0) */

    VALUE arg1;
    rb_scan_args(argc, argv, "01", &arg1);

    RRCALL(self, TCanvas)->cd(NIL_P(arg1) ?
        (Int_t) 0 : NUM2INT(arg1));
```

```
    return self;
}
```

Let's explain the above snippet in more detail. The 'VALUE' construct belongs to the Ruby C API. Almost everything in Ruby is an object and in order to manipulate it in C you have to assign a VALUE to it. That is, the only thing Ruby can understand from a C perspective is VALUEs (which in reality are pointers to more complicated structures).

Now, in order to exchange data between the Ruby side and the ROOT side, all ROOT's data must be converted to VALUEs and vice versa, depending on the call phase. In our example, the above snippet will be executed whenever Ruby tries to execute the code below (assuming that 'c' is a TCanvas instance):

```
c.cd(2)
```

That is, in the Ruby side the '2' parameter is a VALUE, which means that it must be converted to what the actual ROOT method expects; in our case to a C integer. So, using rb_scan_args(), which is another Ruby C API function, we can map the input arguments to VALUEs (in our case there is only one argument) and then use some handy Ruby macros to convert the VALUEs to the right C counterparts. In the above snippet, NUM2INT() is used in order to convert the VALUE '2' to a C integer '2'.

RRCALL() is a ruby-root macro defined in rrcommon.h:

```
#define RRCALL(obj, type)   \
type *v;                    \
Data_Get_Struct(rb_iv_get (obj, "__rr__"), type, v); ((type *)(v))
```

This macro, along with some other similar ones (RRCALL2(), RRMODCALL(), RRMODCALL2(), etc.) are used to make the C code nicer, since making calls between ROOT and Ruby, requires some heavy C casting.

Ending this technical discussion, we note that NIL_P() is another Ruby macro that will check if the input VALUE is nil.

There is no doubt that wrapping each ROOT class member function using the Ruby C API is a very difficult and demanding process. That is the main reason that we tried to describe in detail the wrapping of one ROOT method, which actually belongs to the family of ROOT methods that is very easy to

wrap. Other methods, especially the ones that can be overloaded in the ROOT side are extremely difficult to be wrapped.

Also, a small change in the ROOT interfaces requires changes in the wrapped interfaces. The whole process is difficult to be maintained by a human, so there is a need for developing a machine interface for the automatic generation of the wrapper code.

ruby-root uses the ROOT dictionary technology to cope with the above task.

5.2 Understanding ROOT dictionaries

ROOT uses CINT as a user-friendly interface to develop C/C++ scripts with ROOT functionality. CINT stands for a C/C++ interpreter. A C/C++ interpreter may be slower than a C/C++ compiler, but it is easier to use, since the execution phase of the user's code is more interactive.

CINT needs to have all the type information of the C/C++ source at run-time. That is, when a user executes a member function via CINT, information such as the class that the member function belongs to, the arguments that it takes, its class scope (private, public, protected) and other vital information must be known at the stage of execution; at run-time.

In order CINT to cope with the above, for each C++ class the user wants available via CINT, it generates a file called dictionary. This file describes the class' behavior. CINT contains a full featured API in order to export this information to third party programmers. Thus, anyone can have access to the dictionaries via CINT or even via ROOT (a higher level API) at run-time. ruby-root uses ROOT's dictionaries in order to create the wrapper code.

5.3 mini ruby-root

A utility that uses ROOT's dictionaries must be constructed in order to produce automatically the wrapper code. Since Ruby is easier to use than C/C++ the idea of exporting the ROOT dictionary API to Ruby by hand and then developing a Ruby script that compiles ROOT's prototypes to the Ruby C API is very challenging.

mini ruby-root stands for a small ruby-root distribution that embeds all the necessary wrapper code for the ROOT's classes that export the dictio-

naries' information. That is, using mini ruby-root, a Ruby script that has access to the ROOT dictionaries can be developed.

5.4 ruby-root compiler

rrc stands for the ruby-root compiler. rrc is a Ruby program that aims to compile ROOT classes to the Ruby C API. As it can be easily understood, rrc is the most vital part of the ruby-root distribution. It tries to cope with all the C++ complexity and transform in a universal and generic way a number of C++ classes to the Ruby C API.

rrc can cope with Multiple Inheritance, Overloaded member functions, Static member functions and a number of C++ to Ruby type conversions.

The rrc program is invoked during the building of ruby-root and mini ruby-root must have been built before. The safest way to invoke rrc is via the 'rebuild' script which is part of the ruby-root distribution.

5.5 Complex issues

Although a full compact solution for the automatic wrapper code generation has been developed, there are still parts of ruby-root that require hardcoding. We can refer to these cases us 'complex issues' since most of them deal with tasks that can not be identified by a machine program and a human's interpretation is required.

For a short example, consider the following case:

```
Double_t        *GetX() const {return fX;}
```

The above is the prototype of the GetX() method, which belongs to the TGraph class. This prototype cannot be wrapped by any machine program, since GetX() returns an array of doubles, but nobody, except the user, knows its size in advance. That is, the prototype does not describe exactly the GetX() behavior but rather a summary of how someone expects that GetX() works. In order to resolve the exact usage of GetX() the whole TGraph class must be examined.

A careful TGraph examination, unveils:

```
Int_t            GetN() const {return fNpoints;}
```

Apparently, GetN() will return the size of the array that GetX() returns, but this can only be resolved by a human that understands how TGraph works and not by a machine program that processes the prototypes. There is no way, for a program to conclude that the size of the array GetX() returns can be computed by calling GetN().

Methods like the GetX() one must be hardcoded:

```
static VALUE cTGraph_GetX (VALUE self)
{
    VALUE arr = rb_ary_new ();

    double *x;
    RRCALL2(self, TGraph, x)->GetX ();
    for (int i = 0; i < v->GetN(); i++)
        rb_ary_push (arr, rb_float_new (x[i]));
    return arr;
}
```

All the hardcoded methods are located in the tools/rrhardcode.rb file of the ruby-root distribution.

6 Dynamic ruby-root

6.1 Introduction to dynamic ruby-root

Although ruby-root embeds a complete solution to generate Ruby C API compliant code for the ROOT classes, there is still the problem that someone must collect all the ROOT classes to be wrapped and use the rrc to generate all the needed code.

The idea of wrapping all the ROOT classes sounds challenging, but the rrc is not an elegant solution. The task of compiling all the ROOT classes using rrc is very complex and the produced output is huge in size. On the other hand, it is almost impossible someone to utilize all the ROOT classes in a script or an application. That is, an elegant solution that will give the user access to every ROOT class via Ruby must be developed.

The approach to solve the problem is similar to the one PyROOT[11]. follows. There is no wrapper code generation, but a minimal interface that tries

[11] `http://wlav.home.cern.ch/wlav/scripting/`

to resolve each ROOT method at the run-time of a Ruby script. This means that all the ROOT/C++ to Ruby and vice versa conversion will happen at run-time and there will be no wrapper code generation at compile-time.

This approach has as an advantage that every Ruby script can be aware on all ROOT classes on demand. That is, whenever a user tries to construct a new instance of a ROOT class and call a method, dynamic ruby-root tries to resolve the requested method and call it using the CINT API.

Due to the fact that everything is happening at run-time, dynamic ruby-root is slower than ruby-root. Also, complex issues of ruby-root can not be handled easily in the dynamic version.

6.2 Dynamic ruby-root internals

In order to have the ability of bridging Ruby and ROOT at run-time two major issues must be solved. The first one is that Ruby must know in advance that a ROOT class is instantiated or a ROOT method is called. That is, in the Ruby side of the extension, all ROOT calls must be added to Ruby at run-time. For example when the user creates a new TCanvas object, Ruby must create the appropriate Ruby TCanvas class, which encapsulates the actual ROOT TCanvas class. As we said, there is no wrapper code generation at compile-time, so Ruby is completely unaware of ROOT. All the magic happens at run-time.

The second issue regards to the ROOT side. Whenever the user tries to create a TCanvas object, Ruby must create the Ruby TCanvas class and call the ROOT TCanvas' constructor. Again, at run-time, we have to find a way to resolve the actual ROOT TCanvas constructor, call it, and give the result back to Ruby, as a Ruby object.

The first issue is solved using Ruby's Object#const_missing[12] and Object#method_missing methods. Denote that in Ruby 'Object' is the fundamental Object class (i.e. everything inherits from Object).

In order to solve the problem, a DRRAbstractClass is defined which inherits from Object and DRRAbstractClass#const_missing and DRRAbstraClass#method_missing is used to resolve any ROOT call at run-time. So, whenever the user tries to create a new TCanvas, the const_missing method of DRRAbstractClass is called, and we are in the phase of trying to resolve

[12]It is common in the Ruby world to refer to methods using the '#' character, exactly as we do using '::' in C++. That is, Foo#bar can be seen as the C++ idiom Foo::bar().

if TCanvas is an actual ROOT class. Finding if TCanvas is an actual ROOT class is part of the second major issue. We need the ability to resolve ROOT code at run-time. In order to accomplish this task we use CINT's API. That is, inside the const_missing method we 'ask' CINT if a TCanvas dictionary is available. If so, we call the TCanvas constructor -again using the CINT API- and return the new pointer to the Ruby side, as a Ruby object.

In a similar fashion, the method_missing method is called whenever the user tries to call a method of a ROOT class. That is, after the user has created his/her Ruby TCanvas and tried to call, for example, TCanvas#SetTitle, method_missing will grant control. It will 'ask' ROOT via the CINT API, if the actual ROOT TCanvas class has a member function called SetTitle() and if so it will try to execute it and give back the results to Ruby.

In order to speed up things, since for every call a lot of work must be done in const_missing and method_missing of the DRRAbstractClass, dynamic ruby-root maintains an internal cash of ROOT calls. That is, if the user asks for a ROOT method, the resolving will be done once, at least in the Ruby side. Subsequence calls will be from the internal cash. An exception to this is in overloaded methods. Overloading in dynamic ruby-root is done by inspecting the input Ruby arguments and by constructing an equivalent C prototype. If calls in a specific method are done with different prototypes, that means that the cash mechanism will not be used. That is overloaded methods are treated just like different methods.

It is important to note that Ruby's ability to create Classes and methods at run-time is vital to the implementation of dynamic ruby-root.

6.3 The ROOT Ruby module

Dynamic ruby-root has been adapted officially by the ROOT team, as the official Ruby interface of ROOT. This means that the latest versions of ROOT include dynamic ruby-root by default in the distribution. Inside ROOT, dynamic ruby-root, is called Ruby module. So, whenever we write 'ROOT Ruby module' we actually refer to dynamic ruby-root.

In addition to dynamic ruby-root, the Ruby module contains a TRuby class that gives access to Ruby code via ROOT's command line interface. That is, when a user uses the ROOT command line interface, he/she can execute C++ code or Ruby code via the TRuby interface.

6.4 Configuration

Although, ROOT has adapted dynamic ruby-root in the official distribution, the Ruby module is not activated by default when you build ROOT from source. Below, we will describe the activation process[13].

6.4.1 Building and installing the Ruby module

The Ruby extension module is not built by default when building ROOT from sources. The user should follow the standard installation instructions and enable the build of the Ruby module. Ruby version ¿= 1.8 is required.

```
./configure <arch> --enable-ruby \
        [--with-ruby-incdir=<dir>] \
        [--with-ruby-libdir=<dir>]
gmake
```

If you do not specify the inc and lib directories configure will use Ruby to grab the directories where Ruby's headers and library are located.

A library called libRuby.so [libRuby.dll] will be created in the $ROOTSYS/lib [$ROOTSYS/bin].

6.4.2 Setting up the environment

To work with the Ruby module, the LD_LIBRARY_PATH [PATH] and RUBYLIB need to be set in addition to the standard ROOTSYS.

For Unix platforms:

```
export LD_LIBRARY_PATH=$ROOTSYS/lib:$LD_LIBRARY_PATH
export RUBYLIB=$ROOTSYS/lib:$RUBYLIB
```

for Windows:

```
set PATH=%ROOTSYS%/bin;%PATH%
set RUBYLIB=%ROOTSYS%/bin;%RUBYLIB%
```

[13] You can always find these instructions on-line at the official ROOT site: http://root.cern.ch/root/HowtoRuby.html

6.4.3 Running ROOT scripts from Ruby

The user should make sure that the ruby command is the one of the installation that has been used to build the Ruby extension module. If the RUBYLIB environment variable is set correctly, the user can execute a Ruby script with ROOT functionality in the following way:

```
ruby -rlibRuby foo.rb
```

Another way is to start the Ruby script with the Ruby require command:

```
require 'libRuby'
```

An example is as follows:

```
require 'libRuby'

gROOT.Reset
c1 = TCanvas.new('c1', 'Example with Formula', 200, 10, 700, 500)
#
# Create a one dimensional function and draw it
#
fun1 = TF1.new('fun1', 'abs(sin(x)/x)', 0, 10)
c1.SetGridx
c1.SetGridy
fun1.Draw
c1.Update
```

The user can find a number of examples in the $ROOTSYS/tutorials. To run them you need to execute the command:

```
cd $ROOTSYS/tutorials
ruby demo.rb
```

6.4.4 Invoking the Ruby module from ROOT/CINT interpreter

A ROOT user can run any Ruby command and eventually to run IRB, the Interactive Ruby Shell. The commands to execute are:

```
root [0] gSystem>Load("libRuby");
root [1] TRuby::Exec("require '/usr/local/lib/root/libRuby'");
root [2] TRuby::Exec("c1 = TBrowser.new");
root [3] TRuby::Eval("c1.GetName");
root [4] TRuby::Eval("puts c1.GetName");
Browser
root [5] TCanvas *c2 = new TCanvas("ruby test", "test", 10, 10, 100, 100);
root [6] TRuby::Bind(c2, "$c");
root [7] TRuby::Eval("puts $c.GetTitle");
test
root [8] TRuby::Prompt();
root [9] TRuby::Prompt();
irb(main):001:0> print 1
1=> nil
irb(main):002:0>
```

Notice that whenever you bind a ROOT Object in the Ruby side, you need to use a global Ruby variable, that is a variable with a leading "$".

6.5 Current status

Currently, the Ruby module has been tested on the Linux platform using GCC. The whole development of ruby-root, dynamic ruby-root and the Ruby module has been done using Linux/GCC and further development will be on this platform. Thanks to Axel Naumann[14], the Ruby module can be built under cygwin in the Microsoft Windows platform.

7 Speed Comparison

7.1 Ordinary ruby-root

In this section, we will discuss the speed execution of a Ruby script versus CINT (C interpreted code) and C compiled code. The Ruby script is executed

[14] axel@fnal.gov

using static ruby-root, that means that the wrapper code has been created at compile-time and not at run-time. The benchmark script can be found in Appendix C (stress16.rb). The results are shown in Table 3.

Real Time (secs)	CPU Time (secs)	Language
48.57	47.10	Ruby
27.21	26.28	C Interpreted
0.19	0.19	C Compiled

Table 3: Speed comparison for static ruby-root.

As it was expected Ruby is slower than CINT and of course it can't even been compared with C Compiled code. This is a normal result, since Ruby is an easy to use scripting language and in order to save time for the developer, it must spend additional time on doing the actual work. On the other hand, the script used for the benchmark it is unlikely that it is a realistic one, since what it does is looping in heavy calculations. Although, scientific jobs require often complex calculations, it is unlikely that a scientist will perform such a task in daily work.

Last but not least, execution time is not the only thing we must measure. Deploy time is also an important factor, especially in our modern ages. Sometimes, a machine costs less than a developer's time.

7.2 Ruby module vs PyROOT

In this section we will show the performance of the Ruby module compared to the Python module (PyROOT).

Real Time (secs)	CPU Time (secs)	Language
3.03	2.65	Ruby
2.18	1.85	Python

Table 4: Speed comparison between the ROOT Ruby module and PyROOT.

Obviously, PyROOT is faster than the Ruby module. The real reason for this fact can't be identified very easily. First of all, there is no official information of speed comparison between Python and Ruby, themselves. Even,

if Python is ad hoc faster from Ruby, or vice versa, there is no official information about their C API's performance. For example, Ruby might be faster than Python, but its C API might not be as rich and optimized as the Python one.

On the other hand, PyROOT may be a better implementation of a ROOT interpreter interface than the Ruby module. Although, a technique of caching calls inside the Ruby module has been developed, a lot of additional research must be done, in order to identify spots that lack speed performance and cost of speed bottlenecks.

However, the fact that there is not a huge speed difference (such as between ruby-root and CINT, for example) is very promising.

8 Case Study

In this section, we will develop a small Ruby script, in order to see the practical use of the Ruby interpreter in ROOT. The script can be run using one of the latest ROOT distributions, which contains the Ruby module. The equivalent C++ script will not be shown, however we encourage anyone to try to create a C++ version of the script we will present. We belive it's quite hard to accomplish the whole task, especially by writing a few lines of code.

8.1 Case Study Description

Consider you are part of a worldwide collaboration[15]. Now, a group in the collaboration has been assigned the task of collecting data from a specific source and then share the results with the other groups of the collaboration. This can be done, by collecting the data and then send to the other groups an e-mail with the results. A better way for the group is to place the collected data in a public Web server, so the others can visit a Web page everyday, download the data and make analysis on them. The whole process can be optimized by using cron jobs, in order the Web page to be everyday updated by the 'collectors' group or the data to be downloaded in local places in the workstations used by the collaboration. Although it's easy to create a cron job in a system, in order to have a Web page updated every so on, it is quite verbose to force the whole collaboration to create their own cron jobs for a

[15]This is a very real life situation nowadays with the explosion of Internet and modern communications

single download. So, in our case study we will present a way of optimizing the communication of our theoretical collaboration.

We will substitute the 'collectors' group, with a script that produces 100 random numbers every time someone executes it by visiting a URL. Our task is to develop a Ruby script that will transparently fetch the data (the random numbers), fill a histogram with them and perform a Gaussian fit. The URL is not hypothetical, it is a real one, as it will be shown in the implementation. It is a PHP script located in a public Web server on the net, which produces 100 random numbers between -1 and 1[16]. A sample of the data (100 random numbers between -1 and 1) is the one below:

```
0.846 -0.645 -0.585 0.033 -0.337 0.474 0.631 -0.184 0.203 -0.281 <br/>
-0.004 0.94 -0.437 0.084 0.104 -0.285 0.963 -0.903 0.51 -0.711 <br/>
0.26 -0.583 -0.151 -0.057 -0.968 -0.492 0.562 0.434 0.232 -0.456 <br/>
-0.36 0.078 -0.1 0.055 -0.89 0.564 -0.472 0.742 -0.62 0.732 <br/>
-0.539 0.376 0.672 0.024 -0.54 -0.225 0.74 -0.578 -0.128 0.249 <br/>
-0.288 -0.868 0.667 0.562 0.076 0.699 -0.931 -0.363 0.132 0.301 <br/>
0.181 0.773 -0.621 -0.919 -0.173 -0.511 0.645 0.356 -0.769 -0.976 <br/>
0.087 -0.308 0.401 -0.242 0.716 0.861 0.534 0.455 -0.717 -0.594 <br/>
-0.296 -0.004 -0.462 -0.63 -0.443 0.614 -0.931 -0.374 -0.749 0.202 <br/>
0.928 0.432 -0.026 -0.694 0.514 0.802 -0.204 0.158 0.157 0.027 <br/>
```

Notice the "br" tag, which is an HTML tag, used to format the data. We advise you to visit our case study's URL to have a better feeling on what we are going to do.

8.2 Case Study Implementation

We will try to develop the Ruby script for the task we described, step by step. First thing is to load all the Ruby libraries we need using the common Ruby require command:

```
require 'libRuby'
require 'net/http'
```

The first statement loads the ROOT Ruby module. The second one loads another Ruby library (included by default in any Ruby distribution), which will help us to fetch our data from the Web server.

[16] `http://null.edunet.uoa.gr/~elathan/rr/demo/rr.php`

Next step is to create a new TCanvas object, as well as a histogram of floats. If you are familiar with ROOT, this is probably an every-day task for you. Doing it in Ruby it is even easier than doing it in C++:

```
c1 = TCanvas.new("c1","Ruby Module Case Study",200,10,600,400)
c1.SetGrid

main = TH1F.new("main","Main",100,-4,4)
main.SetFillColor kRed
```

Notice that we can freely omit parentheses. If the above snippet makes you feel uncomfortable, we advise you to have a look at the B Appendix.

Next step, is to fetch our data, using a single line of Ruby code:

```
data = Net::HTTP.get('null.edunet.uoa.gr',
                     '/~elathan/rr/demo/rr.php')
```

That is, 'data' is a string that contains all the information we want to use. Ruby has done all of it for us. It has allocated space, it has made all the required communication with the Web server and it gave us a string that contains the 100 random numbers.

Since we are interested only in the numbers we must eliminate the 'br' tags:

```
data.gsub!(/\<br\/>/, "")
```

If you are not aware of Regular Expressions, you might feel uncomfortable with the above line, which substitutes with an empty string all the 'br' tags in our 'data' string. Regular Expressions is a common technology used in Computer Science for pattern matching. Analyzing how Regular Expressions work is beyond of this thesis. ROOT contains its own Regular Expression library. However, we use Ruby's native Regular Expression support for easiness.

Now, our 'data' string has all the numbers we want to insert in our histogram separated by spaces. It would be handy to store them in an array:

```
entries = data.split(" ")
```

'entries' is an array of 100 strings, each one holding one of our 100 random numbers in text representation. The way we got 'entries' is really quite trivial. We instructed Ruby to split the string in elements, using as a separator the space character.

Now, we are ready to iterate in our 'entries' array and fill our histogram:

```
entries.each do |entry|
    main.Fill(entry.to_f)
    main.Draw("e1p")
    c1.Update
end
```

The above snippet is a Ruby iterator. Ruby iterates in our 'entries' array. In each iteration the variable 'entry' points to the current element of the 'entries' array. In each iteration we fill the histogram with the 'entry' variable and we update the screen. Notice that we use the 'to_f' method in order to convert the string in a float representation.

Last thing is to perform the fit, to update the screen for the last time and force ROOT in loop mode, so as to be able to inspect the results:

```
main.Fit("gaus", "ql")
main.Draw("same")
c1.Modified
c1.Update
gApplication.Run
```

The whole script is presented in Appendix C. A screenshot of the output can be seen in Figure 2.

9 Aknowledgements

ROOT's Ruby Interpreter Interface could never been achieved without the valuable help and contribution of people from the scientific community. Thus, I would like to thank sincerely my Supervisor, Associate Professor of the Physics Department at the University of Athens, Dr. G. Tzanakos, who encouraged me and helped me during the whole implementation of ruby-root. Dr. G. Tzanakos was the first one to guide me in the HEP world and enlight me about the software modern HEP experiments use. He was one of

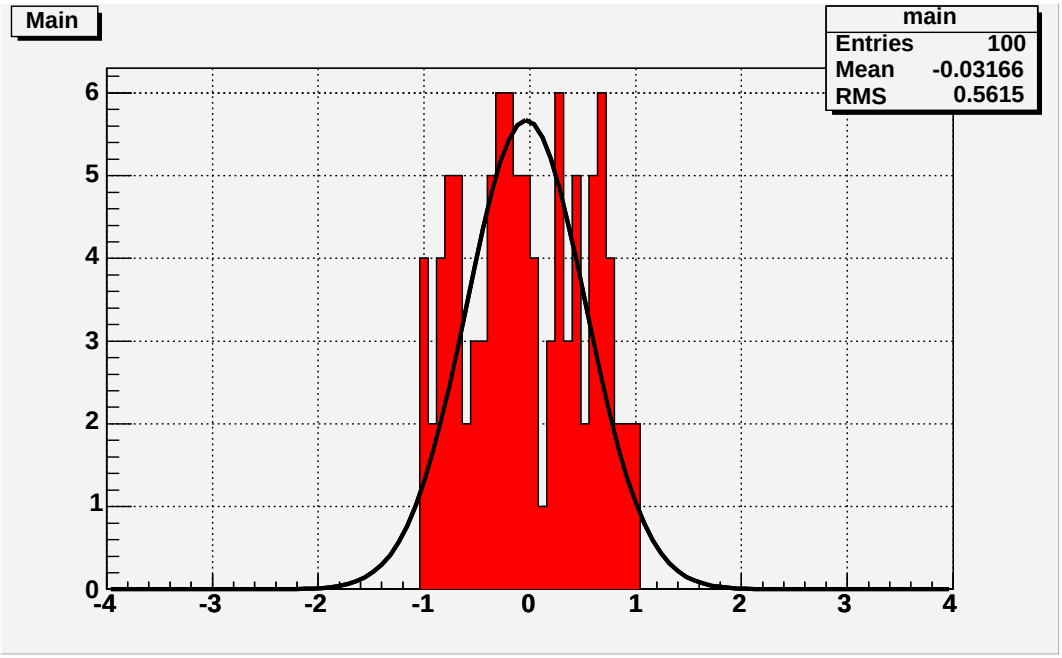

Figure 2: ROOT Ruby module case study.

the few people, who believed that I can complete this project. Also, I would like to ankowledge Rene Brune, Fons Rademakers and Masahuro Goto from the ROOT teem, who helped me with various technical issues regarding the ROOT framework, Juan Alcarez for the contribution of some benchmarks written in Ruby and Axel Naumann for porting ruby-root in the Microsoft Windows platform. Finaly, I would like to thank all the Ruby programmers at the ruby-talk mailing list for the critical answers they gave me regarding technical Ruby aspects.

10 Appendices

A References

[1] Chapter 3 of "The MINOS Technical Design Report" (#NuMI-L-337)
[2] Minos Status and Physics Goals (George S. Tzanakos)
[3] The Pragmatic Programmer's Guide (Pickaxe)

B Migrating from C/C++ to ruby-root

Assuming you are already familiar with the ROOT framework and you have used it to construct C/C++ macros or even C/C++ applications which utilize the ROOT functionality, below we will present a few rules in order to migrate easily your C/C++ work to Ruby. Each rule might be different for ruby-root and for the ROOT Ruby module.

B.1 Constructors

Ruby constructors are a little bit different than the C++ ones. The Ruby equivalent to:

```
TPad *pad = new TPad();
```

is:

```
pad = TPad.new()
```

Keep in mind, that the parentheses can be omitted.
Availability: Both ruby-root and the ROOT Ruby module support this rule.

B.2 Method Calling

In C++ there are two ways to call a member function of a class' instance:

```
TPad *pad = new TPad();
pad->Draw();
```

or:

```
TPad pad;
pad.Draw();
```

In Ruby there is a global way to call a method of an object. This can be seen also from the previous rule. However, in order to make it clear, we present the rule of calling methods in Ruby, here:

```
pad = TPad.new()
pad.Draw()
```

Keep in mind, that the parentheses can be omitted.
Availability: Both ruby-root and the ROOT Ruby module support this rule.

B.3 TApplication

If you don't want your script to end immediately after execution, but loop until you quit from it, then you have to write:

```
tapp = TApplication.new "name"

...enter your code here...

tapp.Run
```

Availability: Both ruby-root and the ROOT Ruby module support this rule. However, in the ROOT Ruby module, we advise you to use the idiom:

```
...enter your code here...

gApplication.Run
```

B.4 C++ Explicit Casts

A common practice for C++ users is to grab objects through C++ explicit casts, like:

```
TTree *t1 = (TTree*)f->Get("t1");
```

In ruby-root, you can do this in the following way:

```
t1 = f.Get("t1").to_ttree
```

The rule is to use the method to_ followed by the ROOT class in lower case.
Availability: This rule is available only in ruby-root. The equivalent rule for the ROOT Ruby module is:

```
t1 = f.Get("t1").as("TTree")
```

In the near future the second idiom will be used for both ruby-root and the ROOT Ruby module.

B.5 ROOT Collections

All ROOT Collections (TList, TClonesArray, etc.), as well as TArray Classes are implicitly converted to Ruby arrays and vice versa. Also, everything which looks in C++ as an array (i.e. Double_t *foo) is converted to a Ruby array and vice versa.

The following example demonstrates this:

```
bases = TClass.new("TPad").GetListOfBases
bases.each do |b|
    p b
end
```

For a practical use of this rule, see the multigraph.rb script which is part of the ruby-root testsuite.
Availability: This rule is available only in ruby-root. A substantial effort to implement this rule in the ROOT Ruby module has been done and in the near future there is a plan to commit the required code to the ROOT CVS tree.

B.6 #to_ary

Some times a ROOT Collection may embed another ROOT Collection in one of its slots. In this case you will have to explicit convert the second collection to a Ruby array using the #to_ary method.

The technique is illustrated in the FirstContour.rb script, which is part of the ruby-root testsuite.
Availability: This rule is available only in ruby-root. A substantial effort to implement this rule in the ROOT Ruby module has been done and in the near future there is a plan to commit the required code to the ROOT CVS tree.

B.7 C++ Enumerations

Not all of the enumerations are supported, but you have access to the most frequently used ones, in the most non-surprising way. I.e. the following is acceptable:

```
foo.SetColor kRed
```

Availability: Both ruby-root and the ROOT Ruby module support this rule.

B.8 C++ Globals

Heavily used globals such as gStyle, gROOT and others are supported, with an exception: do not use gBenchmark, since there is a bug. Instead use:

```
gBenchmark = TBenchmark.new.Start("bench")
```

Availability: Both ruby-root and the ROOT Ruby module support this rule.

B.9 C++ References

C++ References are packed and returned as a Ruby array. So a C++ call:

```
gRandom->Rannor(&x,&y);
```

will be in ruby-root:

```
x, y = gRandom.Rannor
```

If the C++ method returns a value, the latter is the last element of the returned Ruby array.
Availability: This rule is available only in ruby-root. A substantial effort to implement this rule in the ROOT Ruby module has been done and in the near future there is a plan to commit the required code to the ROOT CVS tree.

B.10 Function Pointers

A C/C++ pointer to function is handled in ruby-root as a Ruby user defined method. So, whenever you want to pass a pointer to function, you can pass the symbol of your Ruby method in the following fashion:

```
def background(x, par)
    return par[0] + par[1]*x[0] + par[2]*x[0]*x[0]
end
    ...
backFcn = TF1.new("backFcn", :background, 0, 3, 3)
```

Notice the leading ":" when the Ruby method is passed to the TF1 constructor.
Availability: This rule is available only in ruby-root. A substantial effort to implement this rule in the ROOT Ruby module has been done and in the near future there is a plan to commit the required code to the ROOT CVS tree.

B.11 ROOT Trees and TTree#via

At the time of writing, ruby-root supports the construction of ROOT Trees with doubles, strings and integers. In the latter case there is a bug which leaks the memory. So, use integers in TTrees with care.

ruby-root introduces a new TTree#via method in order to fill a TTree. The following example demonstrates this:

```
# fill the tree
r = TRandom.new
10000.times do |i|
    px, py = r.Rannor
    t1.via :SetBranchAddress, :Fill, { "px" => px,
                                       "py" => py,
                                       "pz" => px*px + py*py }
end
```

For a full example see the treerr.rb and cernbuild.rb scripts, which are part of the ruby-root testsuite. Constructs like TTree#via will be heavily used in ruby-root in the near future.

Availability: This rule is available only in ruby-root. A substantial effort to implement this rule in the ROOT Ruby module has been done and in the near future there is a plan to commit the required code to the ROOT CVS tree.

B.12 Floating values and arithmetic

Ruby's representation of floats is 'a.b', where 'a' is the integer part and 'b' the decimal one. Thus, '1.' or '.1' is not acceptable. However ruby-root is smart to accept '1' when the original C++ function requires a float argument, but reject '1.0' when the original C++ function requires an integer argument.
Availability: Both ruby-root and the ROOT Ruby module support this rule.

B.13 Boolean checks

All C++ boolean types are converted to Ruby booleans, but remember that in Ruby false is only 'false' and 'nil'. Thus, '0' is true! So, keep in mind that you have to explicit check if a method returned '0':

```
if (i && (i%kUPDATE) == 0) # if 0 is true!
```

Availability: Both ruby-root and the ROOT Ruby module support this rule.

C Scripts

C.1 Benchmark Scripts

stress16.rb:

```
# Original port of stress16 ROOT benchmark for RubyRoot
# Author: Juan Alcaraz <Juan.Alcaraz@cern.ch>
#
# Minor adjustments for ruby-root: elathan

def stress16
```

```
=begin
  Prototype trigger simulation for the LHCb experiment
  This test nested loops with the interpreter.
  Expected to run fast with the compiler, slow
  with the interpreter.
  This code is extracted from an original
  macro by Hans Dijkstra (LHCb)
  The program generates histograms and profile histograms.
  A canvas with subpads containing the results is sent
  to Postscript.
  We check graphics results by counting the number of
  lines in the ps file.
=end

   nbuf    = 153     # buffer size
   nlev    = 4       # number of trigger levels
   nstep   = 50000   # number of steps
   # time needed per trigger
   itt     = [1000, 4000, 40000, 400000]
   # acceptance/trigger (last always 0)
   a       = [ 0.25, 0.04, 0.25, 0.0 ]

   #-->int   i, il, istep, itim[192], itrig[192], it, im, ipass;
   #-->float dead, sum[10];

   # create histogram and array of profile histograms
   gRandom.SetSeed
   pipe = TH1F.new("pipe", "free in pipeline",
                   nbuf+1, -0.5, nbuf+0.5)

   hp = []
   TProfile.Approximate
   for i in 0..nlev
      s = "buf%d" % i
      hp[i] = TProfile.new(s, "in buffers", 1000, 0,
                           nstep, -1.0, 1000.0)
   end
```

```
dead   = 0
sum    = [nbuf] + [0]*nbuf
itrig  = [0]*nbuf
itim   = [0]*nbuf

nsteps = 0...nstep
nbufs  = 0...nbuf
nlevs  = 0...nlev

for istep in nsteps
   # evaluate status of buffer
   pipe.Fill(sum[0])
   if (istep+1)%10 == 0
      for i in 0..nlev
         hp[i].Fill(Float(istep), sum[i], 1.0)
      end
   end

   ipass = 0
   for i in nbufs
      it = itrig[i]
      if it >= 1
         # add 25 ns to all times
         itim[i] += 25
         im = itim[i]
         # level decisions
         for il in nlevs
            if it == il+1 and im > itt[il]
               if gRandom.Rndm > a[il]
                  itrig[i]  = -1
                  sum[0]    += 1
                  sum[il+1] -= 1
               else
                  itrig[i]  += 1
                  sum[il+1] -= 1
                  sum[il+2] += 1
               end
            end
```

```
            end
         elsif ipass == 0
            itrig[i] = 1
            itim[i]  = 25
            sum[0]  -= 1
            sum[1]  += 1
            ipass   += 1
         end
      end

      dead += 1 if ipass == 0

   end

end


gbench = TBenchmark.new
gbench.Start("stress16")
stress16
gbench.Show("stress16")
```

hsum.rb:

```
# ruby-root testsuite
# port of the original $ROOT/hsum.C tutorial
# (20/01/2004)  --elathan  <elathan@phys.uoa.gr>
#
# original header:
# To see the output of this macro,
# click begin_html <a href="gif/hsum.gif" >here</a> end_html
# Simple example illustrating how to use the C++ interpreter
# to fill histograms in a loop and show the graphics results

gROOT.Reset

c1 = TCanvas.new("c1","The HSUM example",200,10,600,400)
    c1.SetGrid
```

```
gBenchmark = TBenchmark.new.Start("hsum")

# Create some histograms.
total  = TH1F.new("total","This is the total distribution",100,-4,4)
main   = TH1F.new("main","Main contributor",100,-4,4)
s1     = TH1F.new("s1","This is the first signal",100,-4,4)
s2     = TH1F.new("s2","This is the second signal",100,-4,4)

    total.Sumw2   # this makes sure that the sum of
                  # squares of weights will be stored
    total.SetMarkerStyle(21)
    total.SetMarkerSize(0.7)
    main.SetFillColor(16)
    s1.SetFillColor(42)
    s2.SetFillColor(46)
    slider = nil

# Fill histograms randomly

rnd = TRandom.new.SetSeed
kUPDATE = 500
10000.times do |i|
    xmain = rnd.Gaus(-1,1.5)
    xs1   = rnd.Gaus(-0.5,0.5)
    xs2   = rnd.Landau(1,0.15)
    main.Fill(xmain)
    s1.Fill(xs1,0.3)
    s2.Fill(xs2,0.2)
    total.Fill(xmain)
    total.Fill(xs1,0.3)
    total.Fill(xs2,0.2)
    if (i && (i%kUPDATE) == 0)
        if (i == kUPDATE)
            total.Draw("e1p")
            main.Draw("same")
            s1.Draw("same")
            s2.Draw("same")
```

```
            c1.Update
            slider = TSlider.new("slider","test",
                    4.2,0,4.6,total.GetMaximum,38)
            slider.SetFillColor(46)
        end
        slider.SetRange(0,i/10000.0) if slider
        c1.Modified
        c1.Update
     end
end

slider.SetRange(0,1)
total.Draw("sameaxis") # to redraw axis hidden by the fill area
c1.Modified
gBenchmark.Show("hsum")
gApplication.Run
```

C.2 Case Study Script

```
require 'libRuby'
require 'net/http'

c1 = TCanvas.new("c1","Ruby Module Case Study",200,10,600,400)
c1.SetGrid

main = TH1F.new("main","Main",100,-4,4)
main.SetFillColor kRed

data = Net::HTTP.get('null.edunet.uoa.gr',
                     '/~elathan/rr/demo/rr.php')
data.gsub!(/\<br\/>/, "")

entries = data.split(" ")

entries.each do |entry|
    main.Fill(entry.to_f)
    main.Draw("e1p")
    c1.Update
```

```
end

main.Fit("gaus", "ql")
main.Draw("same")
c1.Modified
c1.Update
gApplication.Run
```