Lizard User Manual

Dino Ferrero Merlino

CERN IT/API

May 2001

Table of Contents

Chapter 1. Overview

Table of Contents

What is Lizard?

Lizard is a new Interactive Analysis Environment (see Lizard Home Page for details) produced by the IT/API group at CERN. The aim of the Lizard project is to produce an analysis tool which can be easily integrated in a C++ based environment and provides functionalities comparable with the core of PAW.

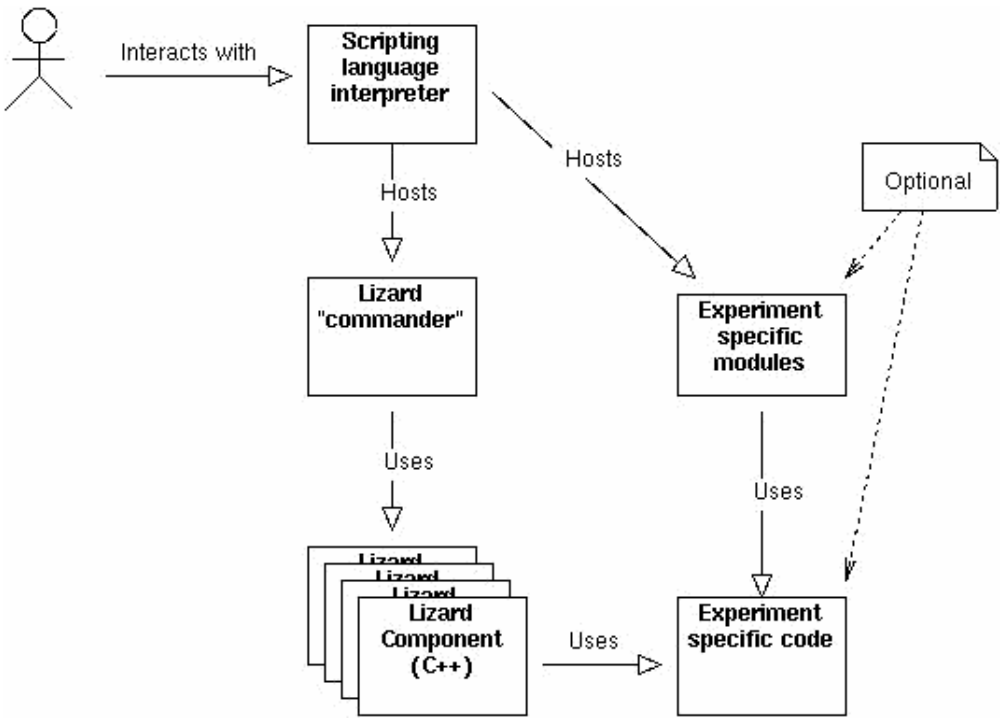
Motivation

The Lizard aim is to provide a package that:

- provides the core functionalities of an Interactive Analysis Environment (histograms,ntuple,fitting,graphics,scripting)
- can easily integrate experiment-specific code
- is based on a simple but flexible architecture
- allows to load only the components really required
- can be extended by skilled users in a relatively simple manner

In order to achieve such a degree of flexibility we opted for a *component-based* system whose binding to the scripting language are automatically generated. In Figure 1.1 the main structure of Lizard is depicted.

Figure 1.1. Relations among user, scripting, Lizard and components



For more details see the chapter on Architecture.

Chapter 2. Lizard at a glance

Table of Contents

Ntuple-like analysis
Getting help

This chapter provides an overview of Lizard features. The package comes with a set of example programs and we plan to have a step-by-step tutorial available at some point. Nevertheless this chapter should provide an overview of Lizard capabilities as well as a first introduction to its object model.

Lizard current implementation

Scripting language

Although Lizard could be easily embedded in any mainstream scripting language (see ARCHITECTURE) such as Python, Tcl, Perl, Ruby etc., the current implementation is using Python. Thus Lizard is a 'guest' of a standard Python 2.0 interpreter, which means that all the power of Python is available (see Python resources for details) to implement more complex user scripts.

Why Python

Why to chose Python over any other language? The following text, taken from the Python summary outlines Python's main features:

Python is an interpreted, interactive, object-oriented programming language. It is often compared to Tcl, Perl, Scheme or Java.

Python combines remarkable power with very clear syntax. It has modules, classes, exceptions, very high level dynamic data types, and dynamic typing. There are interfaces to many system calls and libraries, as well as to various windowing systems (X11, Motif, Tk, Mac, MFC). New built-in modules are easily written in C or C++. Python is also usable as an extension language for applications that need a programmable interface.

The Python implementation is portable: it runs on many brands of UNIX, on Windows, DOS, OS/2, Mac, Amiga...

Python is copyrighted but freely usable and distributable, even for commercial use.

From our point of view the outstanding pros are:

- Object-oriented language
- relatively simple syntax
- freely available and supported by a large community
- easy to integrate with C++
- easy to extend (dynamic loading of modules etc.)

These features overweigth the few cons we found, such as the 'original' indenting scheme. More

details on Python are provided in Chapter 3.

Components in Lizard

As briefly stated in the the section called "Motivation" the user works inside a standard Python interpreter. During the Lizard startup, its "components" are dynamically loaded and become available as Python objects to work with. We can roughly distinguish three kinds of Lizard objects:

`Data Objects`
: Vectors, Histograms, Ntuples

`Managers`
: Entities responsible for creating, bookkeping and deleting Data Objects.

`Active Components`
: Entities responsible for performing tasks, such as plotting, fitting, executing external C++ code.

During Lizard startup the default Managers and Active components are created on behalf of the user.

Working with Lizard mainly consists of asking Managers to create Data Objects and using Active components to transform them. As an example a user may:

1. ask the `HistoManager` to retrieve an
2. `Histogram` instance from the database and invoke the
3. `Plotter` to draw it on the screen.

Although it sounds complex it’s actually very easy, as in this small script:

```
# Ask histo manager to load a histogram. h is the handle of histogram object
h = hm.load1D("10")
# Transform it into a vector (suitable for plotting)
v = vm.from1D(h)
# Now ask the Plotter to draw it
pl.plot(v)

# Note we could have used a shortcut to hide the vector transformation
# Shortcuts are Python functions.
hplot(h)
```

This object-oriented approach may seem more cumbersome e.g. than "monolithic" user interfaces such as in PAW, where there’s only one "global" listener to every user request. Actually it’s not so difficult and it allows to load only the components required for different kind of analysis (and to switch implementations at run-time...).

Default Lizard environment

As explained in the previous section, during Lizard startup, a set of components is pre-loaded and default `Manager` and `Active Component` instances are created. This sections explains exactly what’s defined at Lizard startup.

Default Components

The Table 2.0 summarizes the components created at startup and their role:

Table 2.0. Default components

Identifier	Class type
hm	HistoManager
vm	VectorManager
ntm	NtupleManager
pl	Plotter

Shortcuts

Shortcuts are Python function that implement in a single call several interactions with Lizard components and objects. Their main purpose is to reduce the amount of typing required to carry out simple operations. The Table 2.1 summarizes the shortcuts defined at startup, their purpose and their return value (if any).

Table 2.1. Shortcut

Name	Purpose	Return value
help()	gives help on available methods of classes used in Lizard	
exit()	exit from Lizard	
exe(filename)	Execute a Python program	
hist()	select histogram representation for plotting ("stairs")	
err()	select error bar representation for plotting	
line()	select line representation for plotting	
ylog()	set Y axis to log	
ylin()	set Y axis to lin	
xlog()	set X axis to log	
xlin()	set X axis to lin	
xgrid()	draw "grid" lines perpendicular to X axis	
ygrid()	draw "grid" lines perpendicular to Y axis	
grid()	draw "grid" lines perpendicular to both axis	
xygrid()	draw "grid" lines perpendicular to both axis	
hplot(_histo, opt="")	plot histogram `_histo`	histogram as a VectorOfPoints
hfit(_histo, _model)	fits histogram `_histo` with model `_model` {"G", "E", "P0", "P1", ...}	the projected histogram
cplot1D(_nt, _sel, _cut="")	projects selection `_sel` of tuple `_nt` using cut `_cut` (default none) into a 1D histogram and plots the histogram	fitted curve as a VectorOfPoints
wait()	waits for user to type **`Return`** (useful in scripts between plots)	
prompt(_pr)	prompts user with question `_pr`	whatever user typed

Simple examples

In this section we'll see a few examples on how to do simple things in Lizard. In order to avoid unnecessary details most operations will be done using the shortcuts, since that's exactly their purpose.

Important

The Python syntax uses the `.` (dot) to invoke methods and requires to specify a pair of ellipses `()` after the method name, even if no argument are passed to the method (exactly as in C++). Unlike C++, Python does not require the terminating `;` (semicolon). E.g.

```
ntm.listNtuples()
```

Plotting a histogram

In order to plot an histogram we should first ‘book’ it then plot it. This is the script to do so:

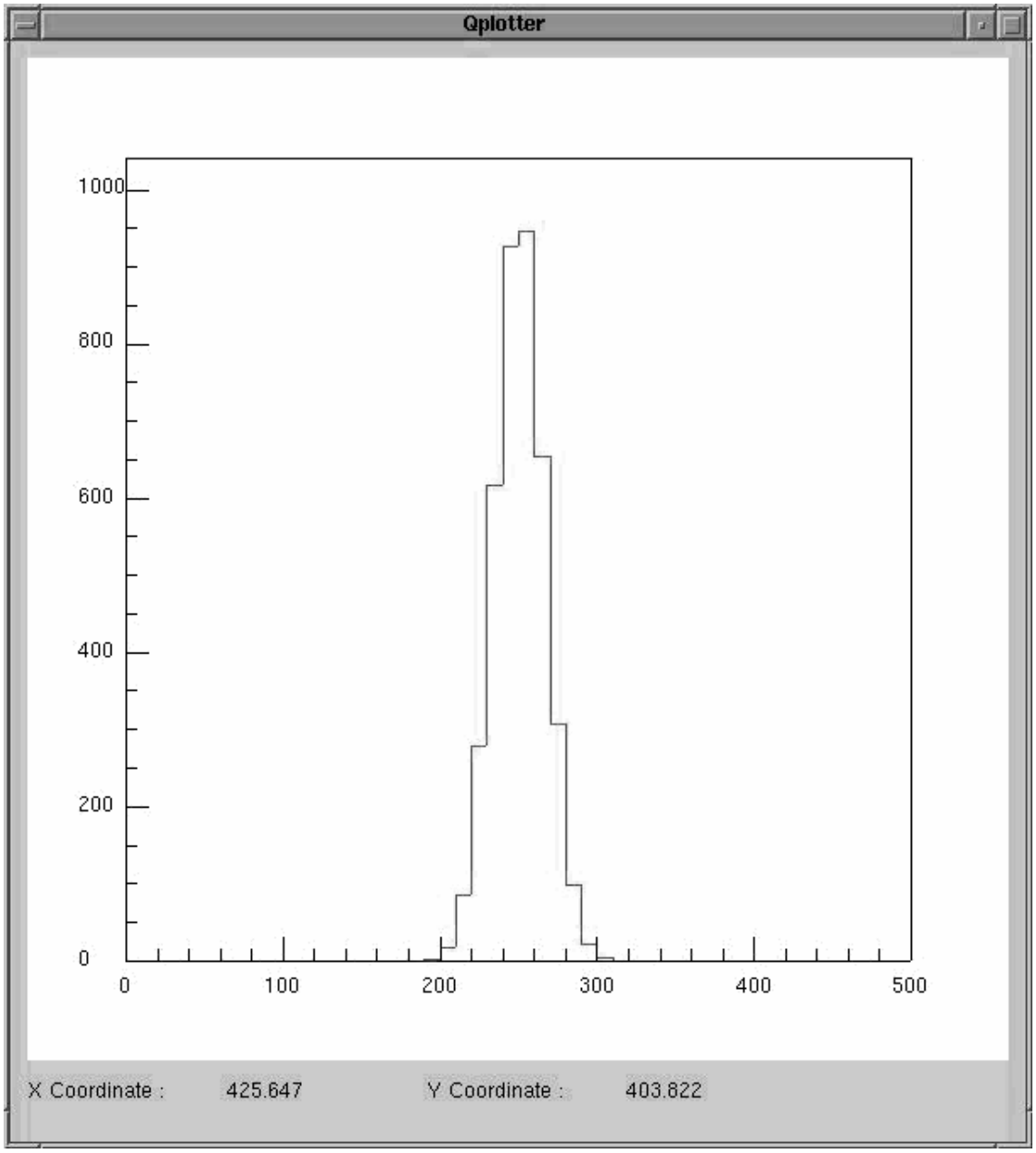
```
# Ask histo manager to create a histogram. h is the handle of histogram object
# First parameter is a short ID, then the title, the no. of bins and the range
h = hm.create1D("10","Empty histogram",10,-5.,5.)
# Plot the empty histogram using a shortcut
hplot(h)
```

Lines starting with # are comments, so we need exactly two lines of Python to book and plot an histogram. Empty histograms are not so useful, so an extra filling loop is worth:

```
# Create histogram
h1=hm.create1D(10,"test 1",50,0., 500.)
# Fill it
for i in range(0.,500.):
    h1.fill(i,100.*exp(-(i-250.)**2/500.))

# The empty line is necessary (to close the for loop)!
hplot(h1)
```

The output of the script is shown in Figure 2.1.

Figure 2.1. Plotting a 1D histogram

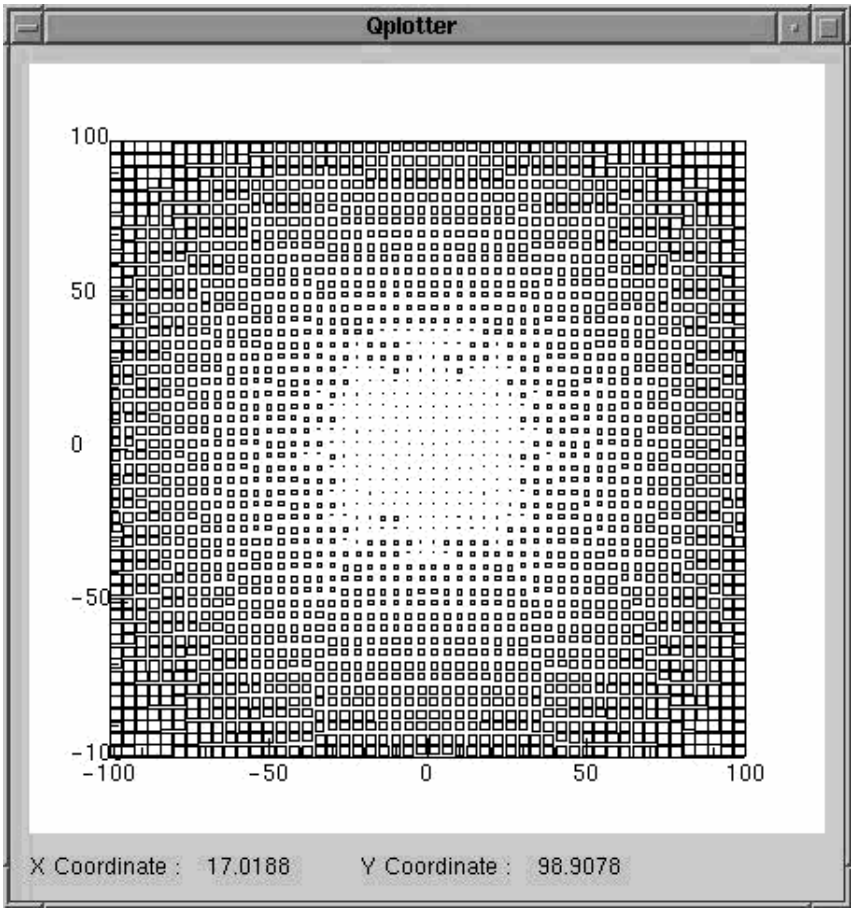


Finally an example with a 2D histogram represented as a BOX plot (the size of each box is proportional to the bin content):

```
# Book 2D histogram
h3 = hm.create2D("20","phi",50,-100,100,50,-100,100)
# Fill it
for x in range(-100,100.,1):
    for y in range(-100,100.,1):
        h3.fill(x,y,x*x+y*y)

# Plot histogram
hplot(h3)
```

The output of the script is shown in Figure 2.2.

Figure 2.2. Plotting a 2D histogram



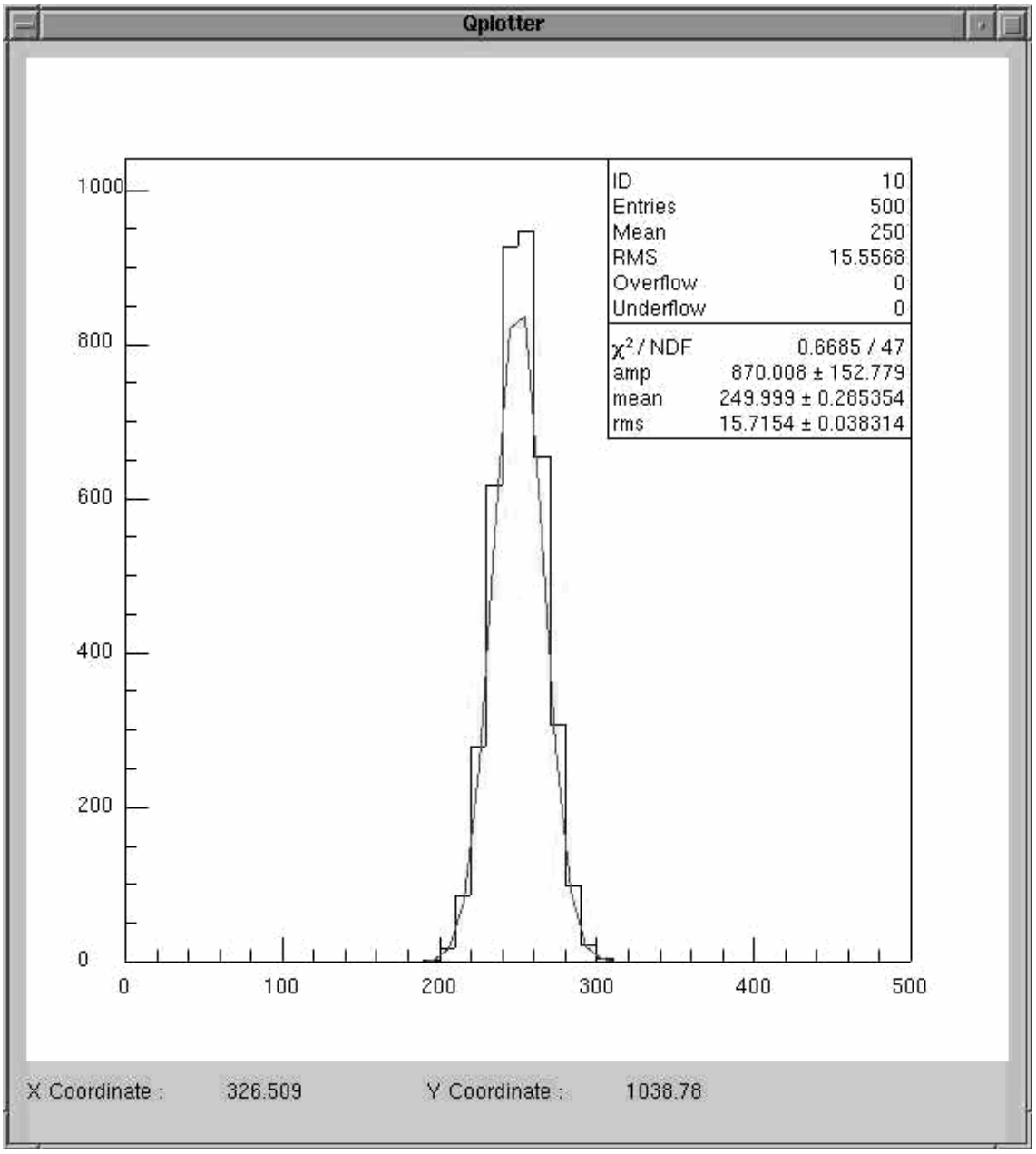
Fitting a histogram

In this case we book the histogram, fill it and then fit it with a Gaussian. The use case is very simple (the fitter can easily retrieve the initial parameters of the Gaussian from the histogram), nevertheless it represents probably the most used fit in real life:

```
# Create histogram
h1=hm.create1D(10,"test 1",50,0., 500.)
# Fill it
for i in range(0.,500.):
    h1.fill(i,100.*exp(-(i-250.)**2/500.))

# The empty line is necessary (to close the for loop)!

# Activate summary information
pl.zoneOption ("option","stats")

# Fit the histogram and overlay the results
v2=hfit(h1,"G")
```

The output of the script is shown in Figure 2.3.

Figure 2.3. Fitting a histogram



Plotting vectors using several zones

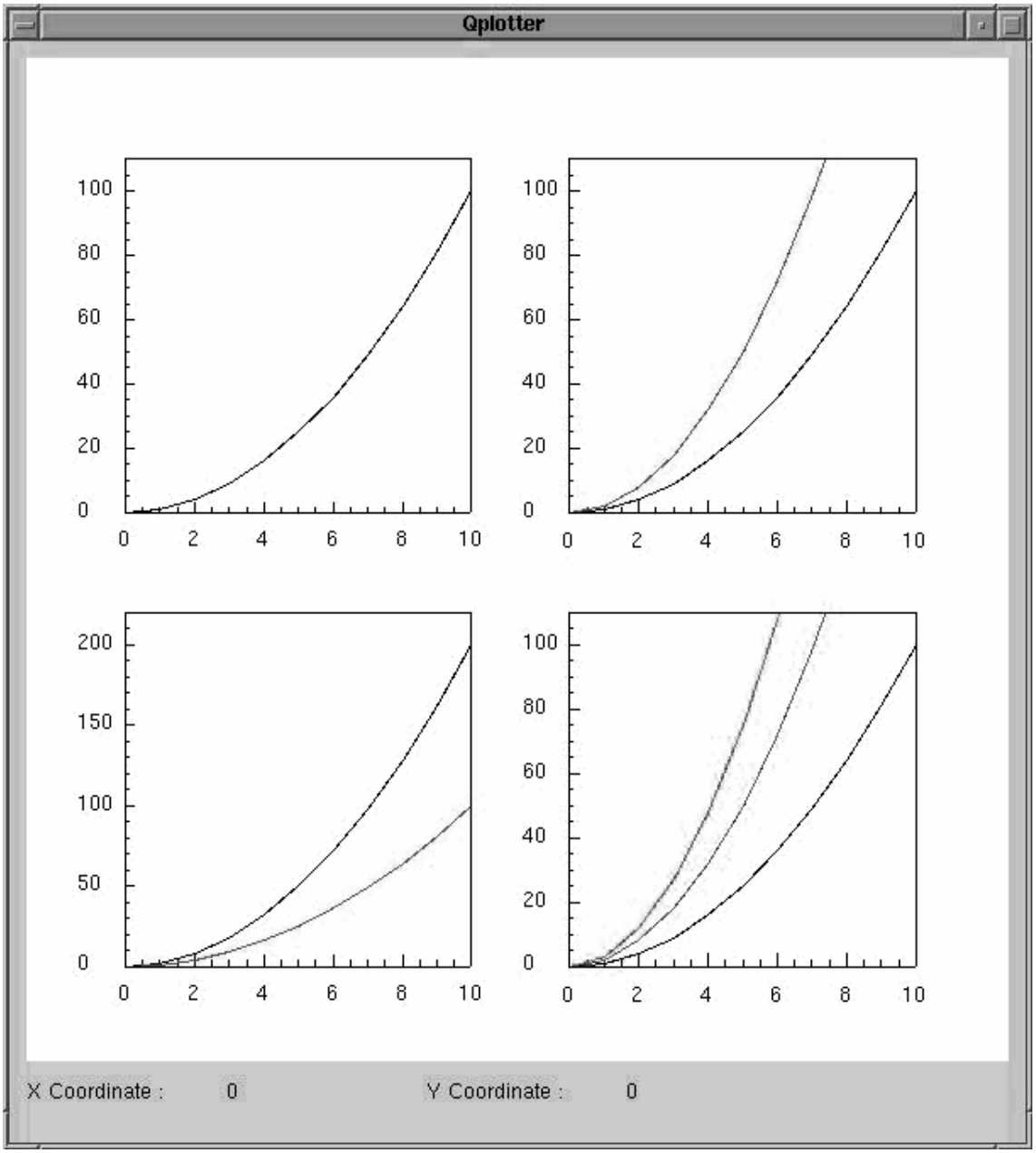
Lizard vectors are the workhorse of the package. Since the `Plotter` and `Fitter` componets work with vectors, shortcuts silently convert them in vectors on behalf of the user. In this example we'll see how to plot vectors in several zones using some of the features of the `Plotter`.

```
# these are Python lists, equivalent to arrays
xvals = [0.,1.,2.,3.,4.,5.,6.,7.,8.,9.,10.]
yvals = [0.,1.,4.,9.,16.,25.,36.,49.,64.,81.,100.]

# Create a Lizard vector
v1 = vm.fromPy(xvals, yvals)
v2 = vm.fromPy(xvals, yvals)
v2.mul(2.)

# Four zones
```

```
pl.zone(2,2)

# First zone
pl.plot(v1)

# Implicit overlay on second zone
pl.plot(v1,v2)

# Now inverted on third zone (see zone limits)
pl.plot(v2,v1)

# Explicit overlay on fourth zone
pl.plot(v1)

pl.dataStyle("linecolor","blue")
pl.overlay(v2,3)

pl.dataStyle("linecolor","green")
pl.overlay(v3,3)


# Reset color
pl.dataStyle("linecolor","")
```

The output of the script is shown in Figure 2.4.

Figure 2.4. Plotting vectors in several zones



In the first part of the script we see how to transform native Python data to a Lizard vector and how to multiply a vector by a scalar. Then we see how to plot a single vector, how to overlay two vector 'implicitly', i.e. using the second parameter of the method and finally how to overlay an arbitrary number of vectors. Notice that the first vector sets the scale for the zone (this behaviour can be changed by setting a zone option).

Ntuple-like analysis

Lizard ntuples are currently based on HepODBMS `Tags` (although the Lizard architecture would allow to 'plug-in' other implementations). Here we assume that those `Tag` collection have been created outside Lizard (e.g. by a reconstruction program) or using the `Analyzer` component. The first step is to find out which Tags are available. This is done by asking the `NTupleManager` instance to list them:

```
ntm.listNtuples ()
```

This is the Lizard output:

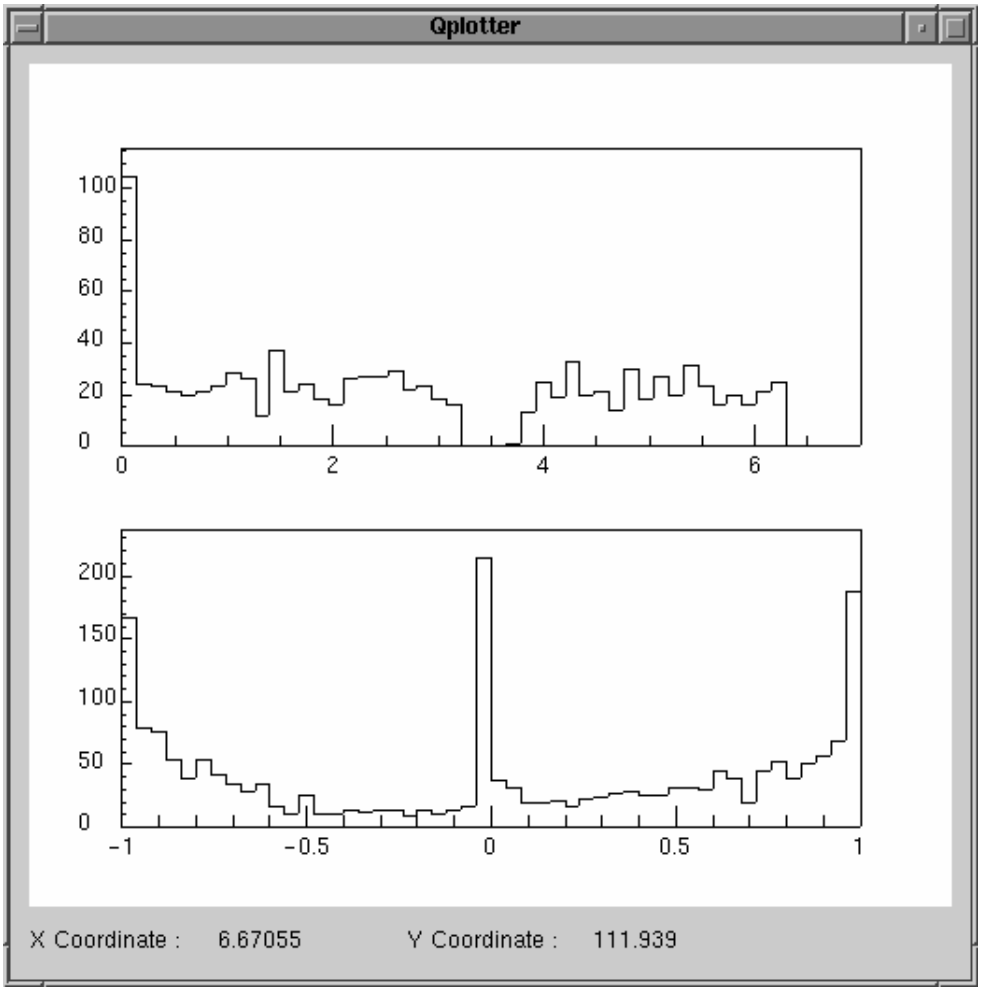
```
:-) ntm.listNtuples () Explorables present: TagCollection1 TagCollection2
TagCollection3 TagCollection4
```

It is now possible to analyze single ntuples or chains, as in the following script:

```
# Open an ntuple
nt1=ntm.findNtuple ("TagCollection1")
# Open a chain
nt2=ntm.findNtuple ("TagCollection1|TagCollection2")

# Two zones
pl.zone(1,2)

# Using ntuple shortcuts

# Plot the phi attribute without cuts
cplot1D(nt1,"phi","")
# Plot the sinus of phi with a cut
cplot1D(nt2,"sin(phi)","(phi < 100 )&& (phi > -1)")
```

The output of the script is shown in Figure 2.5.

Figure 2.5. Plotting ntuple attributes



The `cplot1D` shortcut computes the min/max of the the expression to plot, then books a default histogram, fills it and plots it). In order to see some more ntuple analysis features, let's avoid shortcuts and use the underlying basic features instead.

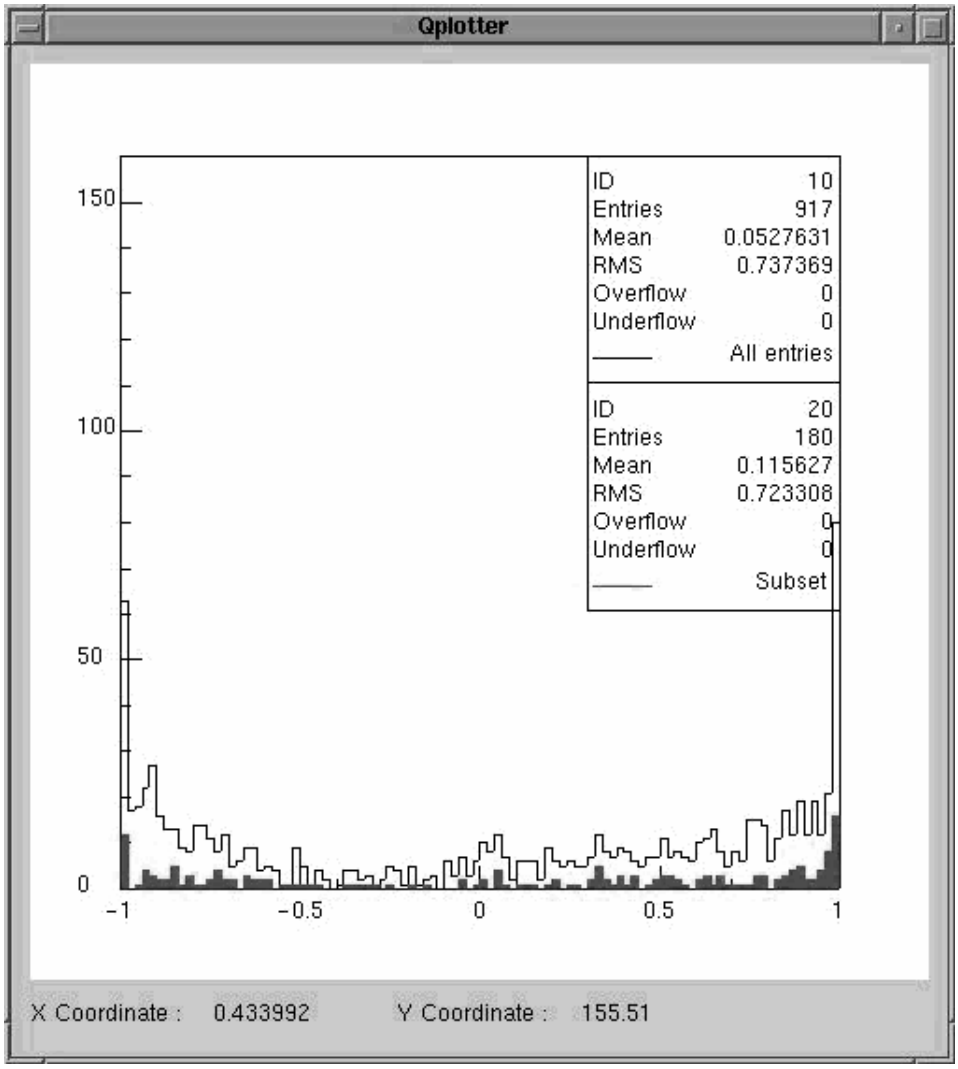
Although the `cplot1D` shortcut is useful to have a quick look at the data, most of the time users prefer to project the attributes on histogram with optimal binning and limits. Moreover it's very handy to be able to specify only a subset of the data sample, so to speed up the 'cut tuning' phase. Let's see an example how to do this:

```
################## That's the ntuple part ##################
# Open an ntuple
nt1=ntm.findNtuple ("TagCollection1")
# Book the histograms with more bins
h1 = hm.create1D("10","sin(phi) with cut",100,-1,1)
h2 = hm.create1D("20","sin(phi) with cut",100,-1,1)

# Project the sinus of phi with a cut to get rid of the fake peak on zero
nt1.cproject1D(h1,"sin(phi)","(phi < 6.29 )&& (phi > 0)")

# Project same quantity with same cut but take only entries from 200 to 500
nt1.cproject1D(h2,"sin(phi)","(phi < 6.29 )&& (phi > 0)",300,200)

################## The rest is plotting... ##################

# Set zone min/max to improve readeability
```

```
pl.setMinMaxY(0,160,1)
pl.zoneOption ("option","stats")

# Plot histogram
pl.dataOption ("legend","All entries")
# This is equivalent to shortcut hplot(h1)
v1=vm.from1D(h1)
pl.plot(v1)
# Reset options
pl.dataOption ("","")

# Plot histogram
pl.dataStyle("fillstyle","solid")
pl.dataStyle("fillcolor","red")
pl.dataStyle("linecolor","red")
pl.dataOption ("legend","Subset ")
pl.dataOption ("representation","hfilled")
v2=vm.from1D(h2)
pl.overlay(v2,0)

# Reset options
pl.zoneOption ("","")
pl.resetMinMax()
pl.dataOption ("","")
pl.dataStyle("","")
```

The script it’s a bit longer due to plotting settings which would not be really necessary and its output is shown in Figure 2.6.

Figure 2.6. Projecting ntuple attributes



Notice that you can have the statistics information for both histograms and that a legend has been defined for each of them.

Getting help

Lizard comes with an help system to provide advice on its capabilities. The main level help can be accessed using the `help()` shortcut which would produce the following output:

```
:-) help()

--------------------------------------------------------------------------------
help is available for the following classes :

        Analyzer
        Annotation
        FitParameter
        Fitter
        HistoManager
        Histogram1D
        Histogram2D
        Ntuple
        NtupleManager
        Plotter
        Point
```

```
        Scalar
        SharedLib
        Vector
        VectorManager
        shortcuts

you can specify more than one class separated by blanks in the help command
for example: help("Histogram1D HistoManager")
```

As explained by the output more detailed information is available on specific components, e.g.:

```
:-) help ("NtupleManager")
--------------------------------------------------------------------------------
methods available for NtupleManager :

   void     listNtuples( ostream&aOs = cout );
   INtuple* findNtuple( const char* aName );
```

The `news()` commands summarizes the changes happened during the most recent releases (this is most useful for development releases, where the help files may not be up-to-date).

```
:-) news()


    New features in Lizard 1.2.0.8 (based on Anaphe 3.5.8):

     o Vectors can now be created fromProfile(profHist, options)
       if options are non-empty, spread will be used as errors

     o hplot() function can plot profiles, added argument for option to fromProf
       hplot(profHist)     plots profileHisto using errors,
       hplot(profHist,"s") plots profileHisto using spread

     o included new NtupleTag/AIDA_Ntuple version (improved cut handling)

    ----------------------------------------------------------------
... etc
```

Finally the `Plotter` component has a method to list the available options:

```
:-) pl.listOptions()

Options for Zones

---------------------
Zone Option List -------------------
The following options for Zone objects are available
via the setProperty method taking two strings as parameter:
setProperty (string name, string value).
("option"," xlinear xlog stats nostats ylinear ylog xaxisgrid yaxisgrid ")
("coordinates"," locked free ")
("mirroraxis","yes no")
------------------------------------------------------------



Options for Datasets(Histograms)

---------------------
DataSet Option List -------------------
```

```
The following options for DataSet objects are available
via the setProperty method taking two strings as parameter:
setProperty (string name, string value).
("representation"," error line histo marker errormark smooth hfilled box color "
("legend","value")
("mtype"," none rect diamond triangle dtriangle utriangle ltriangle rtriangle xc
("msize","value")
-------------------------------------------------------------
```

Python provides a useful *command completion* features (similar to those available on Unix shells) which lists all the methods associated to a Python object. To do so, type the name of an object followed by the dot and then the **TAB** key, as in this example:

```
:-) pl.
pl.__class__    pl.__module__   pl.listOptions  pl.refresh      pl.setRep
pl.__del__      pl.__repr__     pl.overlay      pl.resetMinMax  pl.textStyle
pl.__doc__      pl.dataOption   pl.plot         pl.setMinMaxX   pl.this
pl.__init__     pl.dataStyle    pl.psPrint      pl.setMinMaxY   pl.thisown
```

Methods starting with __ are Python internals, the others correspond to those mentioned in the help screen..

Chapter 3. A crash course on Python

Table of Contents

Introduction

It’s obviously impossible to describe how to use Python in a few lines of text, so we’ll give just the very basic (refer to The Python Tutorial for a comprehensive introduction to the language).

Python is an interpred language: whatever you type at the prompt is submitted to the interpreter after typing **Return**. The interpreter parses the statement and executes it on the fly.

Python supports "standard" data types (numbers and strings) but also high-level built in data types , such as flexible arrays and dictionaries. It comes with a large collection of standard modules as well as built-in modules that provide things like file I/O, system calls, sockets, and even interfaces to GUI toolkits like Tk or Qt.

Python is extensible: if you know how to program in C/C++ it is easy to add a new built-in function or module to the interpreter, either to perform critical operations at maximum speed, or to link Python programs to existing libraries (such as an event reconstruction library).

Scalar variables, functions, statements

Python supports numerical variables (integer and floating-point) and strings (not to mention complex numbers...). The type of a variable is defined by the assignment of a value. Once a value is assigned, Python checks that any *operation* applied to the variable is legal. String arguments must be enclosed in double quotes. As an example the `shell()` function executes any shell command specified as a string argument:

```
shell("ls -ltr")
```

As an example let's see how to declare two variables, print them and apply some operations:

```
:-) myNumber = 1
:-) myString = "1"
:-) print myNumber,myString   # No difference in output
1 1
:-) print sin(myNumber)       # Math operation on a number is OK
0.841470984808
:-) myString2="Bla"+myString  # String concatenation on a string is OK
:-) print myString2
Bla1
:-) # Now illegal operation: sinus of a string!
:-) print sin(myString)
Traceback (most recent call last):
  File "stdin", line 1, in ?
TypeError: illegal argument type for built-in operation
:-)
:-) # Another illegal operation: concatenation of a number
:-) myString2="Bla"+myNumber
Traceback (most recent call last):
 File "stdin", line 1, in ?
TypeError: cannot add type "int" to string
```

Notice how to use the *built-in statement* `print` and a mathematical function such as `sin()`.

Important

When calling Python functions you should always specify a pair of parentheses even if no argument is required (just as in C++):

```
# Correct
exit()
# Wrong
exit
```

Important

Python does not like blanks before statements. Typing a blank in front of the statement produces a syntax error message.

Lists, control-flow statements and more

Python lists and more

Python implements several high-level data typses, such as `list`, `tuple` and `dictionarie`. Among them `list` is by far the most important, so it's worth having a closer look. A `list` is a list of

comma-separated values (items) between square brackets. List items need not all have the same type.

```
:-) # A list of God-knows-what..
:-) a = ['spam', 'eggs', 100, 1234]
:-) # A list of floating point numbers
:-) xvals = [0.,1.,2.,3.,4.,5.,6.,7.,8.,9.,10.]
:-) print xvals
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0]
```

A list can easily be transformed in another list. The following script creates a list of values from `0` to `10` (using the `range()`) and then creates a list containing the square of such values exploiting the *functional programming* features of Python:

```
# A list like the previous one
xvals = [x for x in range(0.,11.)]
# A list with the square values
yvals = [x*x for x in xvals]
print xvals
print yvals
```

The output is:

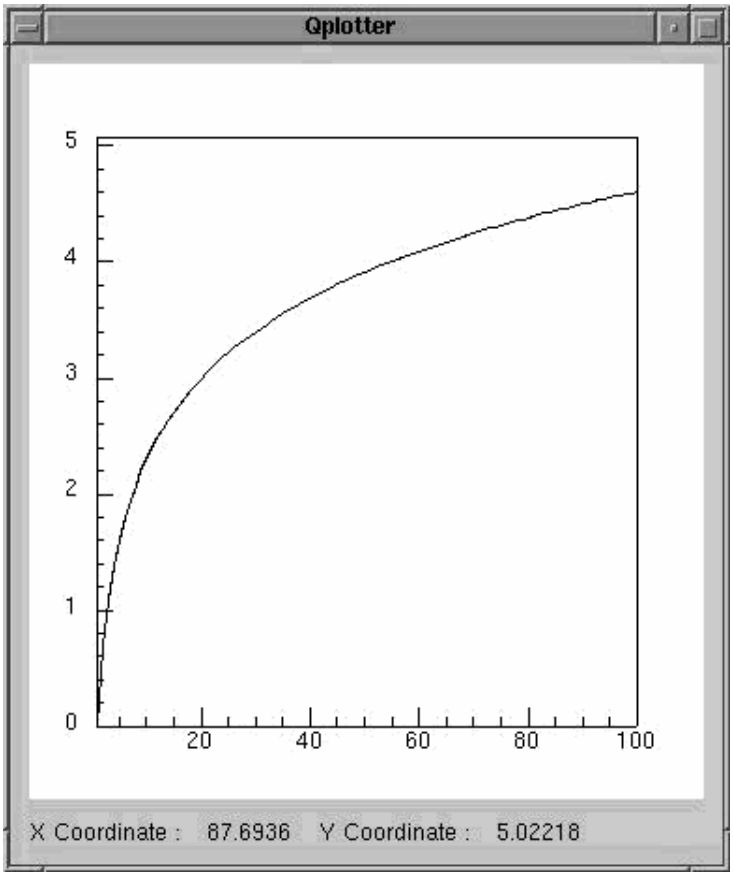
```
:-) print xvals
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
:-) print yvals
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Lizard allows to construct a `Vector` out of two Pyhton lists (one for the X values, the other for the Y values), so for instance to draw a logarithmic curve:

```
# A list of real values
xvals = [x*1. for x in range(1,101)]
# Their logarithm
yvals = [log(x) for x in xvals]
# Convert to Lizard vector
v1=vm.fromPy(xvals,yvals)
# Plot it
pl.plot(v1)
```

The output is shown in Figure 3.6.

Figure 3.6. Plotting Python lists in Lizard



Python control-flow statements

Control-flow statement is the category of statements that alter the program flow. Python provides several such statements, as `for`, `while`, `if`, `break`, `continue`. The widely used `for` statement is closer to its Unix shell equivalent rather than to it C/C++ counterpart. Its purpose is to iterate over the elements of a sequence (`list` or `string` for instance) as shown by this example:

```
# The usual list [1..10]
xvals = [x for x in range(1.,11.)]
# An empty list
yvals = []
# Iterate over all values of list xvals
for x in xvals :
   # Append to yvals the square root of the corresponding xvals
   yvals.append(sqrt(x))

# print the result
print yvals
```

If we need to iterate through a sequence of integer numbers, the `range()` function can be really handy. That's an example:

```
:-) for i in range(0.,50.,10):
...    print i
...
0
10
20
```

```
30
40
```

If the sequence requires real numbers it's better to use the `while` statement, as in this example:

```
:-) a = 0
:-) while a < 3 :
...    print a
...    a = a + 0.5
...
0
0.5
1.0
1.5
2.0
2.5
```

Notice that all iteration statements mark the beginning of the block of statements to execute by a `:` (colon). The end of the block is defined by the first empty line.

Chapter 4. Working with histograms

Table of Contents

Introduction

Lizard histograms are kept in-memory. A `HistoManager` instance takes care of creating, destroying and bookkeping all histograms. Histogram persistency, i.e. storing/retrieving histograms from a persistent store (file or database) is controlled by a `HistoFactory` instance inside the `HistoManager`. The current implementation of Lizard provides an Objectivity/DB factory.

Transient (in-memory) histograms

Creating and deleting histograms in memory

The `HistoManager` instance provide methods to create all supported kind of histograms. This code shows all the creation methods available:

```
# 1D histogram with 50 bins from 0 to 500
h1=hm.create1D(10,"test 1",50,0., 500.)

# 2D histogram with 10X10 bins from 0 to 500
h2=hm.create2D(11,"test 2d",10,0., 500., 10, 0., 500.)

# 3D histogram with 10X10X10 bins from 0 to 500
h3=hm.create3D(12,"test 3d",10,0., 500., 10, 0., 500., 10, 0., 500.)

# 1D profile histogram with 100 bins from 0 to 50
h4=hm.createProfile (14,"test profile",100,0., 50.)

# 1D histogram with 4 bins of different width
h5=hm.create1DVar(15,"test 1d variable binning",[0.,1.,5.,13.,41.])

# 2D histogram with 4X4 bins of different width
h6=hm.create2DVar(16,"test 2d variable binning",[0.,11.,12.,31.,40.],[10.,11.,12
```

It is possible to see the list of existing histograms by invoking the `HistoManager` method `list()`. To remove all histograms in one go, it's necessary to call the `delete(0)`:

```
:-) hm.list() # Histograms created using the previous script

1D histograms:
label = 10 title = 'test 1'
label = 15 title = 'test 1d variable binning'

2D histograms:
label = 11 title = 'test 2d'
label = 16 title = 'test 2d variable binning'

3D histograms:
label = 12 title = 'test 3d'

profile histograms:
label = 14 title = 'test profile'

:-) hm.deleteHisto(0) # Delete all histograms
:-) hm.list()

no histograms stored
```

Histogram IDs

Histograms in memory can be further referenced using their `ID`, i.e. the string or number which comes as first argument in the create method. Unlike in HBOOK, Lizard histogram IDs are strings and not numbers. If a number is specified it is converted to the corresponding string. By default creating a new histogram with the same ID overrides the old one and produces a warning.

```
:-) # Numeric ID
:-) h1=hm.create1D(10,"test 1",50,0., 500.)
:-)
:-) # String ID, will override
:-) h1=hm.create1D("10","test 1",50,0., 500.)
INFO: 1D-histogram with ID 10 has been deleted.
:-)
:-) # Now delete using the numericla ID
:-) hm.deleteHisto(10)
INFO: 1D-histogram with ID 10 has been deleted.
```

It is possible to exclude the overriding by using one of this methods on the `HistoManager`

```
hm.disableOverwrite()
hm.enableOverwrite ()
hm.disableWarnOverwrite()
hm.enableWarnOverwrite ()
```

as in this example:

```
:-) hm.list()

no histograms stored

:-) hm.disableOverwrite()
:-) h1=hm.create1D(10,"test 1",50,0., 500.)
:-) h1=hm.create1D("10","test 1",50,0., 500.)
ERROR: found 1D-histogram with ID 10 and overwriting is disabled.
ERROR when trying to register. No histo created.
```

To retrieve an histogram identified by ID there are methods available on the `HistoManager` as shown by this script:

```
# Get back four handles of  histograms having different type
h11 = hm.retrieveHisto1D(10)
h12 = hm.retrieveHisto2D(11)
h13 = hm.retrieveHisto3D(12)
h14 = hm.retrieveProf(14)
```

Persistent (on-disk) histograms

Important

Since the current version of the prototype supports persistency with Objectivity/DB some of the information is very specific and would not apply to using another store.

Selecting the database and creating directory structure(optional)

The first step is to select the (Objectivity/DB) database for the Histograms. By default, a DB called "test-0" will be used (the DB is created if it is not yet existing). To select another database for histograms e.g.:

```
hm.selectStore("zerbino")
```

whose resulting output is:

```
:-) hm.selectStore("zerbino")
Naming root directory has been created.
Created /usr directory
Created /usr/dinofm directory
```

Lizard has allocated a new Objectivity/DB database and created the top of a directory structure in it. The structure starts with `/usr` and then the username taken from the environment variable `$USER`,

so in this case the directory structure contains `/usr/dinofm`. It is now possible to create and remove directories as well as to navigate in the tree using the `HistoManager` methods:

```
:-) # What's current directory? The newly created /usr/dinofm
:-) hm.pwd()
/usr/dinofm
:-) # List the content of directory (nothing)
:-) hm.ls()
:-)
:-) # Create two subdirectories and list them
:-) hm.mkdir("Data")
:-) hm.mkdir("MC")
:-) hm.ls()
Data                                Wed May  9 10:19:37 2001
MC                                  Wed May  9 10:19:42 2001
:-) # Now go in the Data subdirectories
:-) hm.cd("Data")
:-) # Create two more directories
:-) hm.mkdir("2000")
:-) hm.mkdir("2001")
:-) # Current directory
:-) hm.pwd()
/usr/dinofm/Data
:-) # The new subdirectories of Data
:-) hm.ls()
2001                                Wed May  9 10:20:05 2001
2000                                Wed May  9 10:20:02 2001
```

Storing histograms in database

Let's see how to store transient histograms in the `/usr/dinofm/Data/2001` directory.

```
# Create 1D histogram with 50 bins from 0 to 500
h1=hm.create1D(10,"test 1",50,0., 500.)

# Create 2D histogram with 10X10 bins from 0 to 500
h2=hm.create2D(11,"test 2d",10,0., 500., 10, 0., 500.)

# Change directory (absolute path)
hm.cd("/usr/dinofm/Data/2001")
# Check we're really there...
hm.pwd()
# List histograms (none)
hm.ls()

# Store histograms by ID
hm.store(10)
hm.store("11")

# List them
hm.ls()
```

Such a script would produce this output on screen:

```
:-) # Create 1D histogram with 50 bins from 0 to 500
... h1=hm.create1D(10,"test 1",50,0., 500.)
:-)
:-) # Create 2D histogram with 10X10 bins from 0 to 500
... h2=hm.create2D(11,"test 2d",10,0., 500., 10, 0., 500.)
```

```
:-)
:-) # Change directory (absolute path)
... hm.cd("/usr/dinofm/Data/2001")
:-) # Check we're really there...
... hm.pwd()
/usr/dinofm/Data/2001
:-) # List histograms (none)
... hm.ls()
:-)
:-) # Store histograms by ID
... hm.store(10)
:-) hm.store("11")
:-)
:-) # List them
... hm.ls()
10          1D test 1                   Wed May  9 10:35:20 2001
11          2D test 2d                  Wed May  9 10:35:20 2001
```

Removing histograms from database

There are now two histograms in the directory. To remove them by ID:

```
# Change directory (absolute path)
hm.cd("/usr/dinofm/Data/2001")
# Remove from database
hm.scratchHisto (10)
hm.scratchHisto (11)
```

Notice how numerical and string IDs can be freely mixed.

Retrieving histograms from database

Histograms in a persistent store can be retrieved by ID:

```
# Change directory (absolute path)
hm.cd("/usr/dinofm/Data/2001")
# Retrieve histograms
h1 = hm.load1D (10)
h2 = hm.load1D (11)
print h1.title()
print h2.title()
```

Retrieving histograms with non-existing IDs or mixing histogram types would produce error messages like these:

```
:-) hm.ls()
10          1D test 1                   Wed May  9 10:35:20 2001
11          2D test 2d                  Wed May  9 10:35:20 2001
:-) h1 = hm.load1D (16)
HistoFactory::load1D> requested histo with label= 16 not found
:-) h1 = hm.load2D (10)
HistoFactory::load2D> cannot load 2D from histo with dim = 1
```

Lizard `Histogram` objects

Lizard `Histogram` objects have different types (1D,2D, Profile etc.) but they share a common

ancestor class. This section will first introduce the methods defined on all histograms, then will look into the methods that are specific for each type.

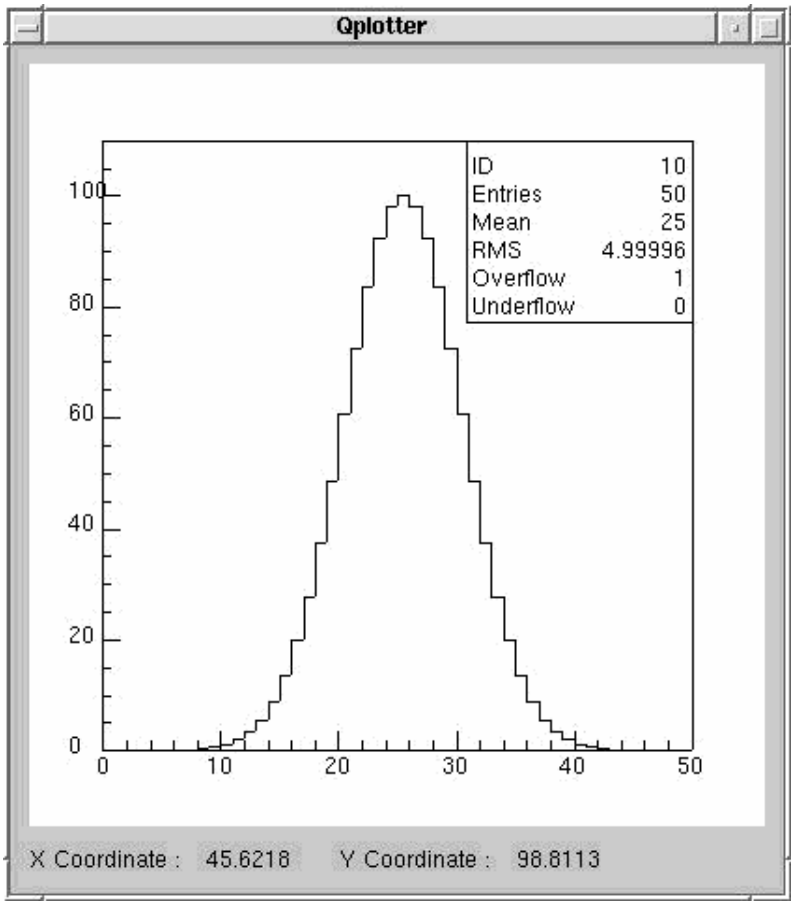
Methods common to all Lizard `Histogram` objects

This is a quick list of the methods available on each histogram independently of its specific type:

```
# Methods common to all histograms

# Reset histogram
h1.reset()
h2.reset()

# Number of dimensions
h1.dimensions()
h2.dimensions()

# Title
h1.title()

# Entries
# Print the total number of entries
h1.allEntries ()
# It’s also possible to assign the total number of entries to a variable
# a =  h2.allEntries ()

# Print the number of in-range entries
h1.entries ()
# Print the number of out-range entries (underflow/overflow)
h1.extraEntries()
# Print the number of equivalent entries (SUM[weight] ^ 2 / SUM[weight^2])
h1.equivalentBinEntries()

# Total bin content
# Sum of in-range bin contents in the whole histogram
h1.sumBinHeights()
# Sum of extra bin contents in the whole histogram
h1.sumExtraBinHeights()
# Sum of all (both in-range and extra) bin contents in the whole histogram
h1.sumAllBinHeights()
```

To show an example, we create a histogram as in Figure 4.1

Figure 4.1. A Gaussian



The next screen shows the output produced by such common methods on that histogram:

```
:-) # 1D histogram with 50 bins from 0 to 50
... h1=hm.create1D(10,"test 1",50,0., 50.)
:-) # Fill the 1D histogram
... for i in range(0.,51.):
...     h1.fill(i,100.*exp(-(i-25.)**2/50.))
...
:-)
:-) # Methods common to all histograms
:-) # Number of dimensions
... h1.dimensions()
1
:-) # Title
... h1.title()
'test 1'
:-)
:-) # Entries
... # Print the total number of entries
... h1.allEntries ()
51
:-) # Print the number of in-range entries
... h1.entries ()
50
:-) # Print the number of out-range entries (underflow/overflow)
... h1.extraEntries()
1
:-) # Print the number of equivalent entries (SUM[weight] ^ 2 / SUM[weight^2])
... h1.equivalentBinEntries()
17.724516454605929
:-)
```

```
:-) # Total bin content
... # Sum of in-range bin contents in the whole histogram
... h1.sumBinHeights()
1253.3133575714553
:-) # Sum of extra bin contents in the whole histogram
... h1.sumExtraBinHeights()
0.00037266531720786707
:-) # Sum of all (both in-range and extra) bin contents in the whole histogram
... h1.sumAllBinHeights()
1253.3137302367725
```

Methods common to Lizard 1D `Histogram` objects

Lizard 1D `Histogram`, such 1D histograms with fixed and variable binning and profile histograms, share a large set of common methods. This section presents these methods and highlights the differences related to profile histograms.

```
# Methods common to all 1D histograms

# Access to bin information
# Bin content (of bin 20)
h1.binHeight (20)
# Bin error (of bin 20)
h1. binError (20)
# It is possible to map an x value to the bin. Error of bin at x = 20.44
h1.binError(h1.coordToIndex (20.44))
# Retrieve the lowest/highest bin (useful for iteration)
h1.minBin()
h1.maxBin()
# Maximum bin content
h1.maxBinHeight ()
# Minimum bin content
h1.minBinHeight ()

# Statistics
# Mean
h1.mean()
# RMS
h1.rms()

# Methods specific to non-profile 1D histograms (1D and 1DVar)
# Fill with weight=1.
h1.fill(1.)
# Fill with weight=4.
h1.fill(1.,4.)

# Methods specific to profile 1D histograms
# Fill with weight=1.
h4.fill(1.)
# Fill with weight=4.
h4.fill(1.,4.)
# Get the "spread" of a bin
h4.binSpread (20)
```

As a real example let’s see how to put in a Python list the bins’ content of a histogram:

```
# 1D histogram with 50 bins from 0 to 50
h1=hm.create1D(10,"test 1",50,0., 50.)
# Fill the 1D histogram
```

```
for i in range(0.,51.):
    h1.fill(i,100.*exp(-(i-25.)**2/50.))

# Empty list
cont = []
# Iterate over bins
for i in range (h1.minBin(), h1.maxBin()) :
   # Put in the list the content of each bin
   cont.append(h1.binHeight(i))

# Print the list
print cont
```

Methods for Lizard 2D and 3D `Histogram` objects

Lizard 2D and 3D `Histogram` objects have their own set of specific methods, which are usually straightforward extensions of their 1D equivalent. This section will just list some of these methods and leave the user the thrill to find the other ones.

```
# 2D histo access bin content/error
h2.binHeight (5,5)
h2.binError (5,5)

# Fill with and without weight
h2.fill (1.,1.,8.)
h2.fill (1.,1.)

# Projections
hx = h2.projectionX ()
hy = h2.projectionY ()

# Slices
# Only the bin containing coordinate X=5.
hs1 = h2.sliceX (5.)
# All bins containing in the range X = 3..6
hs2 = h2.sliceX (3.,6.)

# 3D histo access bin content/error
h3.binHeight (5,5,5)
h3.binError (5,5,5)

# Fill with and without weight
h3.fill (1.,1.,1.,7.)
h3.fill (1.,1.,1.)

# Projections
hx = h3.projectionXY ()
hy = h3.projectionYX ()
```

Chapter 5. Working with Vectors

Table of Contents

Introduction

Lizard vector's are not just simple container of real values (like e.g. KUIP vectors) but objects designed to store information (value with error) at "Points" in space. Each of these points contains:

- a *value*, which is an instance of 'Point' having:
  - a *value*
  - a *positive error* on the value
  - a *negative error* on the value
- *coordinates*, which are in turn 'Points' of the same type, having:
  - a *value* of each of the coordinates
  - a positive *'error'* on each of the coordinates
  - a negative *'error'* on each of the coordinates

This information allows, for instance, to map an histogram bin completely: the height of the bin corresponds to the value, the error on the bin corresponds to the value errors, the center of the bin corresponds to the X coordinate and the distances between the center and the lower/upper edge of the bin corresponds to the 'errors' on the coordinate. This kind of vector is therefore easily mapped to any histogram and can be used e.g. for fitting or plotting. As briefly explained in the first part, Vectors are created and managed by a `VectorManager`.

Role of vectors in Lizard

Lizard vectors have a central role in connection with the `Plotter` and `Fitter` components. The `Fitter` accepts a vector as the set of points to fit the model to, the `Plotter` accepts most data (the exception being Scattter plots) as vectors. For instance the Lizard command to plot are:

```
# Plot one vector
pl.plot(v1)
# Overlay a vector
pl.overlay(v1,1)
# Overlay two vectors in one go
pl.plot(v1,v2)
```

Shortcuts such as `hplot(h1)` to plot a histogram, silently convert the histogram to a vector.

Using the `VectorManager`

Creating `Vector` from histograms

The `VectorManager` allows the user to create new vectors which will then be managed on his behalf. One way to create vectors is to exploit the methods to create a vector out of an existing histogram:

```
:-) # 1D histogram with 50 bins from 0 to 500
... h1=hm.create1D(10,"test 1",50,0., 500.)
:-)
:-) # 2D histogram with 10X10 bins from 0 to 500
... h2=hm.create2D(11,"test 2d",10,0., 500., 10, 0., 500.)
:-)
:-) v1=vm.from1D(h1)
:-) v2=vm.from2D(h2)
:-) vm.list()

1 1D vectors:
 1 size 50

1 2D vectors:
 1 size 100
```

Once the vectors are created, they are identified by a sequence number starting from 1. The output of the the `list()` method will be ordered by those sequence number.

Retrieving a `Vector` from manager

To retrieve a vector from the `VectorManager` one should use either `retrieve1D` or `retrieve2D` methods, which produce a reference to an existing `Vector`:

```
:-) vm.list()

1 1D vectors:
 1 size 50

1 2D vectors:
 1 size 100
# There are two vector. Get another reference to one of them
:-) v3=vm.retrieve1D (1)
:-) vm.list()

1 1D vectors:
 1 size 50

1 2D vectors:
 1 size 100
 # Still two vectors in the manager!
```

Removing a `Vector` from manager

To remove a `Vector` from the manager, just delete it:

```
:-) # No effect, was just a reference
:-) del v3
:-) vm.list()

1 1D vectors:
 1 size 50

1 2D vectors:
 1 size 100

:-) # This will remove the 1D vector
:-) del v1
:-) vm.list()

1 2D vectors:
 1 size 100

:-) # This will remove the 2D vector
:-) del v2
:-) vm.list()

no vectors stored
```

Copying a `Vector`

To make a copy of `Vector` use the `deepClone()` method as in this example:

```
:-) # Make a real copy of vector
:-) v2 = v1.deepClone()
```

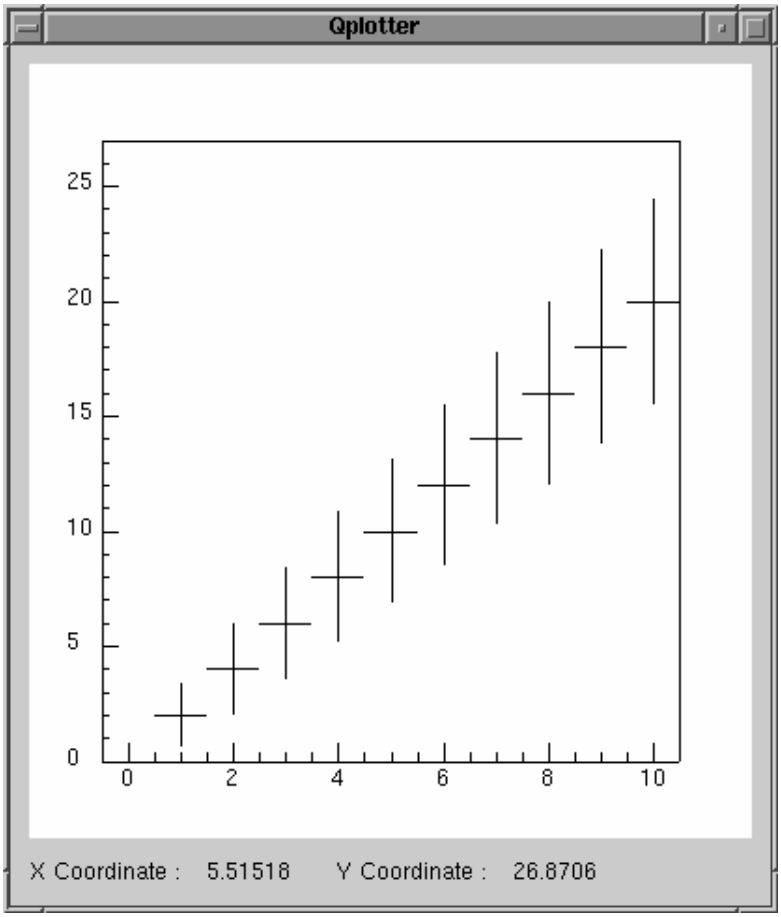
Creating a `Vector` from Python lists

In the the section called "Lists, control-flow statements and more" we saw briefly how to make a Lizard `Vector` out of two Pyhton lists. It's now time to see in detail that functionality. In this case the script will fill in not only coordinates on X and value on Y, but also the respective errors:

```
# A list from 0 to 10
xvals = [x*1. for x in range(0.,11.)]
# A list with twice the values
yvals = [2*x for x in xvals]
# 'Error' on X i.e. half binwidth
exvals = [0.5 for x in xvals]
# Error on Y, i.e. sqrt(Y)
eyvals = [sqrt(y) for y in yvals]

# Create Lizard vector from Python lists
v1=vm.fromPy(xvals,yvals,exvals,eyvals)

# Plot
pl.dataOption ("representation","error")
pl.plot(v1)
```

The output of the script is shown in Figure 5.1.

Figure 5.1. Making a Lizard vector from Python lists



Writing/reading back a `Vector` from ASCII file

A Lizard `Vector` can be saved in an ASCII file and read back later on. This allows to export/import data from other formats and to produce dumps to send to collegues or to the Lizard support team in case of bugs... Once we have a `Vector` handle there are two `VectorManager` methods to use:

```
# Save in ASCII file
v1.toAscii("v1.dat")
# Read back from ASCII file
v1.fromAscii("v1.dat")
```

The file format is plain ASCII with bits of XML-like tags. In the future it will evolve in proper XML format. As an example this is the ASCII file produced by saving the previously produced `Vector`:

```
# This and the next three lines are needed to read in the vector. Change at own
# Vector: version, dimension and size (nX, nY for 2D)
# 1 1 11   11
# <AIDAAnnotation size="0" >
# </AIDAAnnotation>
# x ex+ ex-    val ev+ ev-
0 0.5 0.5          0 0 0
1 0.5 0.5          2 1.41421 1.41421
2 0.5 0.5          4 2 2
3 0.5 0.5          6 2.44949 2.44949
4 0.5 0.5          8 2.82843 2.82843
```

```
5 0.5 0.5          10 3.16228 3.16228
6 0.5 0.5          12 3.4641 3.4641
7 0.5 0.5          14 3.74166 3.74166
8 0.5 0.5          16 4 4
9 0.5 0.5          18 4.24264 4.24264
10 0.5 0.5          20 4.47214 4.47214
```

After a header part, there are basically only ‘blank-separated’ numbers in a well defined sequence (as explained by the last comment line).

Operations on vectors

A Vector can be modified by replacing the values of its points, by applying transformations such as shifting or scaling and finally by performing operations with other vectors or scalars.

Translating and scaling a `Vector`

These transformation applies to any instance of a `Vector` object. The following code shows how to use them:

```
# scale (multiply by value) the value of each data point
v1.scaleV(0.5)
v1.scaleV(2)
# v1 as it was...
# scale (multiply by value) the coordinate index (0=x, 1=y, ...)
v1.scaleCoordinate (2,0)
v1.scaleCoordinate (0.5,0)
# shift (add value) the value of each data point
v1.shiftV(5)
# shift (add value) the coordinate index (0=x, 1=y, ...)
v1.shiftCoordinate (0,3)
```

Notice that scaling/shifting transformations can be applied to both values and coordinates. In the latter case the index of the coordinate must bespecified, since the transformation apply to vectors of any dimensionality.

Arithmetic operations with other `Vector`

It is possible to *add, subtract, multiply and divide* a `Vector` object by another `Vector`. Only values are affected, while coordinates remain unchanged. In order to avoid creating new objects, this operations modify the current object, so their behavior is like the C/C++ operators `+= -= *= /=`.

```
# Create an empty vector
v1=vm.create()
# Read the content from file
v1.fromAscii("v1.dat")
reading in 1D-vector of size 11 (11, )
# Make a copy
v2=v1.deepClone()

# Now combine v1 and v2

# The content of v1 is now 0
v1.subVector (v2)
```

```
pl.plot(v1)
# The content of v1 is now equal to v2
v1.addVector (v2)
pl.plot(v1)
# The content of v1 is 1
v1.divVector (v2)
pl.plot(v1)
# And back to the original values...
v1.mulVector (v2)
pl.plot(v1)
```

Errors are propagated according to the rules used by HBOOK histograms, so users can effectively apply these methods to implement histogram operations.

Arithmetic operations with scalars

It is possible to *add, subtract, multiply and divide* a `Vector` object by a scalar (real) value. Only values are affected, while coordinates remain unchanged. In order to avoid creating new objects, this operations modify the current object, so their behavior is like the C/C++ operators `+= -= *= /=`.

```
# Arithmetic operations with scalars
v1.add(5)
v1.sub(5)
v1.div(5)
v1.mul(5)
# original values as usual
```

The `Point` inside vectors

Retrieving single points out of a vector

A Lizard `Vector` is a container of `Point` objects. It is possible to access individual points contained in a vector either by retrieving special points (e.g. the point corresponding to minimum/maximum value) or by iterating over all points, as shown by this script:

```
############## Vector creation ####################
# A list from 0 to 10
xvals = [x*1. for x in range(0.,11.)]
# A list with twice the values
yvals = [2*x for x in xvals]
# 'Error' on X i.e. half binwidth
exvals = [0.5 for x in xvals]
# Error on Y, i.e. sqrt(Y)
eyvals = [sqrt(y) for y in yvals]

# Create Lizard vector from Python lists
v1=vm.fromPy(xvals,yvals,exvals,eyvals)
############## End of Vector creation ####################

# Retrieving points and accessing their content
# get the point containing the maximum value
pMax = v1.maxValue ()
# print point dimension
print "Maximum point, Dimension =",pMax.dimension()
```

```
# print value and errors
print "Value =",pMax.value(), " (-" , pMax.vMinus() ,  "+" , pMax.vPlus() ,")"

# get the point containing the minimum value
pMin = v1.minValue ()
print "Maximum point, Dimension =",pMin.dimension()
# print value and errors
print "Value =", pMin.value(), " (-" , pMin.vMinus() ,  "+" , pMin.vPlus() ,")"

# Now loop on all points
nPoints = v1.nPoints()
print "All points"
print "Value X Binwidth"
for i in range(0,nPoints):
   curP = v1.point(i)
   print curP.value(), curP.coordinate (0), (curP.coordPlus (0)+curP.coordMinus
```

The output of this script is as follows:

```
Maximum point, Dimension = 1
Value = 20.0  (- 4.472135955 + 4.472135955 )
Maximum point, Dimension = 1
Value = 0.0  (- 0.0 + 0.0 )
All points
Value X Binwidth
0.0 0.0 1.0
2.0 1.0 1.0
4.0 2.0 1.0
6.0 3.0 1.0
8.0 4.0 1.0
10.0 5.0 1.0
12.0 6.0 1.0
14.0 7.0 1.0
16.0 8.0 1.0
18.0 9.0 1.0
20.0 10.0 1.0
```

The `Vector` methods to retrieve special points or to iterate over all of them are extremely useful (imagine looking for peaks in a large vector to find out starting values for fitting). Once a `Point` object is selected it is straightforward to retrieve its value and coordinates (and corresponding uncertainties) by using methods such as `value()`.

Modifying points in a vector

It is possible to modify a Lizard `Vector` by adding or removing single points or by changing individual points. This is an example on how to remove one point:

```
############## Vector creation ####################
# A list from 0 to 10
xvals = [x*1. for x in range(0.,11.)]
# A list with twice the values
yvals = [2*x for x in xvals]
# 'Error' on X i.e. half binwidth
exvals = [0.5 for x in xvals]
# Error on Y, i.e. sqrt(Y)
eyvals = [sqrt(y) for y in yvals]

# Create Lizard vector from Python lists
v1=vm.fromPy(xvals,yvals,exvals,eyvals)
############## End of Vector creation ####################
```

```
# there are 11 points
print "No of points=",v1.nPoints()
# Remove the first point
v1.removePoint(0);
# One point less ...
print "No of points=",v1.nPoints()
```

It is possible to use arithmetic operations to change the value or the coordinate(s) of a point:

```
:-) p1 = v1.point(0)
:-) p1.coordinate (0)
1.0
:-) p1.value()
2.0
:-) p1.shiftCoordinate(0,-1)
:-) p1.add(3.0)
:-) p1.coordinate (0)
0.0
:-) p1.value()
5.0
```

The vector's `Annotation`

Each Lizard vector contains an `Annotation`. An instance of `Annotation` is a set of *key* , *value* pairs that can be used to store properties and corresponding values. As an example both histogram statistics and fit results are kept as annotations attached to the corresponding vector. It's easy to see this if we create an new histogram, convert it into a vector and dump it in an ASCII file

```
# Create histo
histo=hm.create1D(10,"test 1",10,0., 500.)
# Fill it
for i in range(0.,500.):
    histo.fill(i,100.*exp(-(i-250.)**2/500.))

# Transform histo in a vector and save it in ASCII format
v1=vm.from1D (histo)
v1.toAscii ("v1.dat")
# Show the content of the ASCII format
shell ("cat v1.dat")
```

The resulting output is something like that:

```
# This and the next three lines are needed to read in the vector. Change at own
# Vector: version, dimension and size (nX, nY for 2D)
# 1 1 10   10
# <AIDAAnnotation size="7" >
#   <AnnotationAttribute key="ID" val="10" vis="true" />
#   <AnnotationAttribute key="title" val="test 1" vis="false" />
#   <AnnotationAttribute key="Entries" val="500" vis="true" />
#   <AnnotationAttribute key="Mean" val="250" vis="true" />
#   <AnnotationAttribute key="RMS" val="12.7234" vis="true" />
#   <AnnotationAttribute key="Overflow" val="0" vis="true" />
#   <AnnotationAttribute key="Underflow" val="0" vis="true" />
# </AIDAAnnotation>
# x ex+ ex-    val ev+ ev-
48.202 1.79799 48.202          1.45888e-33 9.04371e-34 9.04371e-34
97.8238 2.17623 47.8238          3.43057e-18 1.86941e-18 1.86941e-18
147.089 2.91084 47.0892          4.07065e-07 1.84461e-07 1.84461e-07
```

```
195.214 4.78585 45.2141          2.77643 0.934482 0.934482
237.122 12.8777 37.1223          1928.89 367.592 367.592
262.23 37.7696 12.2304          2028.21 380.951 380.951
303.851 46.1488 3.85122          3.45022 1.15207 1.15207
351.932 48.0676 1.93239          6.1318e-07 2.76603e-07 2.76603e-07
401.186 48.8137 1.18634          6.29309e-18 3.41888e-18 3.41888e-18
450.804 49.1963 0.803698          3.26373e-33 2.01876e-33 2.01876e-33
```

It's easy to see that this vector contains an annotation with seven attributes. The first attribute has a key `ID`, a corresponding value `10` (the histogram ID) and a visibility flag equal `true`. The second attribute has a key `title`, a corresponding value `test 1` (the histogram title) and a visibility flag equal `false`. The first attribute will appear in the summary information on the screen (since its visibility is true), the second will not be shown, since its visibility is false.

While the purpose of *visible* attributes is obvious (the key and the value will appear as an entry in the summary information), invisible attributes can store special information that is used by other Lizard components, user programs or just kept for human eyes.

Retrieving vector's `Annotation` as a Lizard object

It is possible to retrieve the annotation attached to a vector using the `annotation()` method of the vector:

```
############## Vector creation ####################
histo=hm.create1D(10,"test 1",10,0., 500.)
# Fill it
for i in range(0.,500.):
    histo.fill(i,100.*exp(-(i-250.)**2/500.))

v1=vm.from1D (histo)
############## End of Vector creation ####################

# Retrieve annotation
a1 = v1.annotation()
# Print its size
print a1.size()
```

Modifying vector's `Annotation`

Once got a handle on the annotation it is possible to modify it by hiding unuseful information and adding new attributes:

```
############## Vector creation ####################
histo=hm.create1D(10,"test 1",10,0., 500.)
# Fill it
for i in range(0.,500.):
    histo.fill(i,100.*exp(-(i-250.)**2/500.))

v1=vm.from1D (histo)

# Create two zones and set limits to leave space for summary information
pl.zone(2,1)
pl.setMinMaxY(0,3000,1)
pl.setMinMaxY(0,3000,2)

# Plot the original vector on the first zone
```

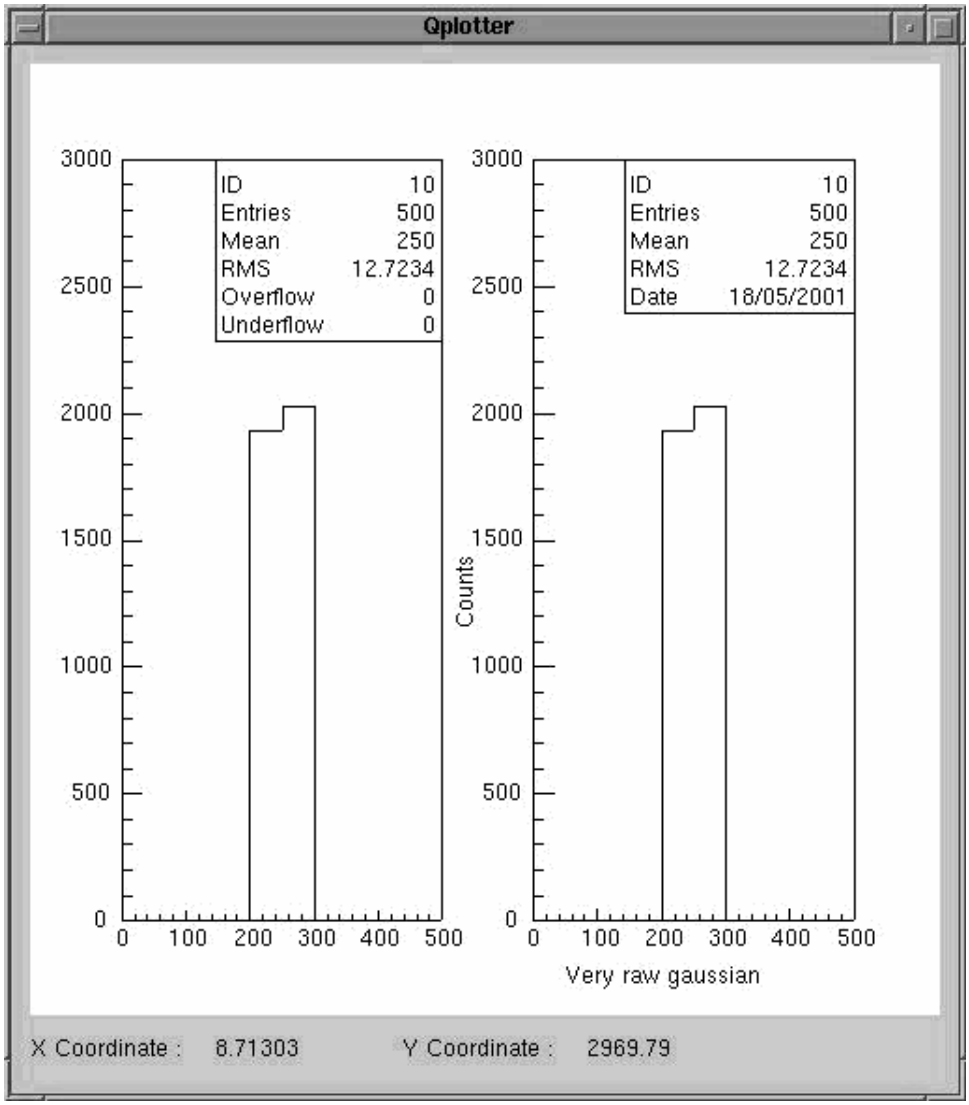
```
pl.plot(v1)
##############                  ####################

# Retrieve annotation
a1 = v1.annotation()
# Hide Overflow/Underflow by overriding the visibility (the 0 at the end)
a1.add ("Overflow","0",0)
a1.add ("Underflow","0",0)

# Add two "invisible" attribute which are understood by Lizard plotter as axis t
a1.add ("XAxisTitle","Very raw gaussian",0)
a1.add ("YAxisTitle","Counts",0)

# Add a "visible" user attribute (today’s date) by giving 1 as last parameter
a1.add ("Date",strftime("%d/%m/%Y",localtime(time())),1)

# Add an "invisible" user attribute (the 0 at the end)
a1.add ("Remark","Should improve the binnning",0)

# Plot the vector with modified annotation on the second zone
pl.plot(v1)

# Print the "invisible" remark
print
print "For your eyes only:"
print a1.find("Remark")
print
```

In this example the original annotation is modified in several ways:

- The original `Overflow , Underflow` pairs are hidden by switching off the visibility flag (last argument to the method)
- Two "invisible" attributes coding the axis titles are added to the annotation *the Lizard plotter takes care of decoding the information). Notice that their visibility is turned off (0 as last argument to the method).
- One "visible" attribute containing today’s formatted date is added (visible since 1 is passed as last argument to the method).
- Finally an "invisible" attribute is appended. It won’t appear anywhere on the plot but it can be printed if the user knows the name of the key...

The output of the script is shown in Figure 5.2.

Figure 5.2. Editing annotation



Chapter 6. Working with Ntuples

Table of Contents

Caching expressions
Using parameters to avoid compiling code

Introduction

Lizard ntuples are currently based on HepODBMS Explorable Collections stored in a Objectivity/DB database. In the rest of this section it is assumed that such collections have been already created outside Lizard (although they can be created from inside using the `Analyzer`). Future versions of Lizard will very likely implement the same functionalities on top of other persistency systems.

The `NtupleManager` component

As explained in the section called " Components in Lizard" the user interacts with Lizard managers to obtain other objects to work with. Ntuple analysis is no exception, since the "standard" way of doing it is:

- Ask the `NtupleManager` component to return a handle to an ntuple (or a chain of ntuples).
- Invoke methods on the `Ntuple` object to carry out analysis tasks (e.g. scanning, projecting etc.).

The default `NtupleManager`

During startup, a default instance of `NtupleManager` called `ntm` is created on behalf of the user. Such instance allows the user to list the available ntuples, to retrieve one ntuple (or a chain of similar ntuples), to define parameters for cuts etc. The following script shows some `NtupleManager` functionalities:

```
# List available ntuples
ntm.listNtuples ()
# Open an ntuple
nt1=ntm.findNtuple ("TagCollection1")
# Open a chain
nt2=ntm.findNtuple ("TagCollection1|TagCollection2")
# List ntuples in memory
ntm.review ()
```

This is the resulting output:

```
:-) # List available ntuples
... ntm.listNtuples ()

Explorables present:

        TagCollection1
        TagCollection2
        TagCollection3
        TagCollection4

:-) # Open an ntuple
... nt1=ntm.findNtuple ("TagCollection1")
:-) # Open a chain
... nt2=ntm.findNtuple ("TagCollection1|TagCollection2")
```

```
:-) # List ntuples in memory
... ntm.review ()

1.      TagCollection1
2.      TagCollection1|TagCollection2
```

Finding ntuples

The `NtupleManager` will lookup ntuples in the `HepExplorable` container of the `System` database attached to the current Objectivity/DB Federated Database. This is place where the `listNtuples ()` method would retrieve the names to show to the user. It is possible to locate ntuples whose description is stored in other databases or containers by prepending to the ntuple name respectively the database and container names, separated by colons, e.g.:

```
# Open an ntuple using only the name (and default database:container)
nt1=ntm.findNtuple ("TagCollection1")
# Equivalent call specifying explicitly database:container
nt2=ntm.findNtuple ("System:HepExplorable:TagCollection1")
```

Defining chains

A ‘chain’ of ntuples is an ordered set of ntuples sharing the same structure but containing different data. The Lizard `NtupleManager` allows to define a chain of ntuples as a sequence of names separated by the | character as in this example:

```
# Open an chain using only names (and default database:container)
nt1=ntm.findNtuple ("TagCollection1|TagCollection2")
# Equivalent call specifying explicitly database:container
nt2=ntm.findNtuple ("System:HepExplorable:TagCollection1|System:HepExplorable:Ta
```

Operations on ntuples

Once got a handle to an ntuple, it is possible to scan, probe or project attributes (or functions of attributes) while applying cuts and possibly restricting the set of "rows" taken into account. Lizard allows to specify both the quantitiy to "sample" (i.e. what ends up in the histogram) and the cut as C++ expressions that are compiled on the fly and dynamically loaded. The Table 6.0 summarizes the most important ntuple commands:

Table 6.0. Ntuple methods

Method	Arguments	Comment
cscan	string aAttrs,string aCut, aFirst = 0, aRows = LONG_MAX, std::ostream& aOs = std::cout	Scan attributes for rows satisfying the cut. It's possible to restrict the range of rows and to specify an alternative output stream
probe1D	const string aFunc,string aCut,double xMin, double xMax, long aMatcNum	Evaluate the minimum/maximum of an attribute and the number of rows satisfying the cut.
cproject1D	IHistogram1D* aHist,string aFunc,string aCut, aFirst = 0, aRows = LONG_MAX	Project the attribute onto the histogram for rows satisfying the cut. It's possible to restrict the range of rows taken into account.
cproject2D	IHistogram2D* aHist,string aFunc1,string aFunc2,string aCut, aFirst = 0, aRows = LONG_MAX	Project the attributes onto the 2D histogram for rows satisfying the cut. It's possible to restrict the range of rows taken into account.
scatter2D	IScatter2D* aPlot, string aFunc1 ,string aFunc2,string aCut, aFirst = 0, aRows = LONG_MAX	Produce a scatter plot of the attributes for rows satisfying the cut. It's possible to restrict the range of rows taken into account.

Whenever the name `aFunc` is mentioned, it means any mathematical function combining one or more ntuple attributes, such as `sin(phi*0.99)` or `sqrt(pt*pt-Energy)`.

Scanning ntuples

Scanning an ntuple is the process of printing on a stream (by default the terminal window) the value(s) of one or more ntuple attributes. It is possible to specify the attribute(s) either by giving their names separated by blanks or by specifying the position of an attribute in the ntuple, e.g.

```
# Open an ntuple
nt1=ntm.findNtuple ("TagCollection1")
# List ntuple attributes
nt1.listAttributes ()
# Print the values of the first two attributes starting from row 0 to row 4
nt1.cscan("eventNo pt","",0,5)
# The same thing specifying the position of the last attribute (2)
nt1.cscan(2,"",0,5)
```

The output would be something like that:

```
:-) # Open an ntuple
... nt1=ntm.findNtuple ("TagCollection1")
:-) # List ntuple attributes
... nt1.listAttributes ()
eventNo         signed long int
pt              double
phi             double
Energy          double

:-) # Print the values of the two first attributes starting from row 0 to row 4
... nt1.cscan("eventNo pt","",0,5)

eventNo         pt

0               7.36183
```

```
1               9.97665
2               7.80474
3               3.06194
4               13.1003

:-) # The same thing specifying the position of the last attribute
... nt1.cscan(2,"",0,5)

eventNo         pt

0               7.36183
1               9.97665
2               7.80474
3               3.06194
4               13.1003
```

Notice that the row count starts from `0`. It is possible to restrict the data sample by specifying a cut as the second argument to `cscan()`. A cut is a string containing any C++ expression that evaluates to a boolean. The string is compiled and the cut condition applied to every row: if the condition is false, the row is discarded, as shown by this example:

```
# Open an ntuple
nt1=ntm.findNtuple ("TagCollection1")
# Now a simple cut
nt1.cscan("phi","phi>4",0,5)
# A cut using the ceil() math function (round upwards to nearest integer)
nt1.cscan("phi","ceil(phi)>4",0,5)
# A cut using the floor() math function (round downwards to nearest integer)
nt1.cscan("phi","floor(phi)>4",0,5)
```

The corresponding output is:

```
:-) # Open an ntuple
... nt1=ntm.findNtuple ("TagCollection1")
:-) # Now a simple cut
... nt1.cscan("phi","phi>4",0,5)

phi

4.55469
4.42411

:-) # A cut using the ceil() math function (round upwards to nearest integer)
... nt1.cscan("phi","ceil(phi)>4",0,5)

phi

4.55469
4.42411

:-) # A cut using the floor() math function (round downwards to nearest integer)
... nt1.cscan("phi","floor(phi)>4",0,5)

phi
```

The last cut did not select any row (no values of phi greater than 5 were in the first 5 rows). It is possible to redirect the output of the `cscan()` method by specifying as last parameter a Python stream, as in this example:

```
# Open an ntuple
nt1=ntm.findNtuple ("TagCollection1")
# Create a Python output stream associated to a file
os=ofstream("scanout.dat")
# Re-direct the cscan() output to the stream
nt1.cscan("phi pt","phi > 0",0,10,os)
# Delete the stream
del os
# Show the file content
shell("cat scanout.dat")
```

Probing ntuples

Probing an ntuple is the process of computing the minimum and maximum value of a function of one or more ntuple attributes. The typical use-case for probing is to compute histogram limits in order to book a histogram, as in this example:

```
# Open an ntuple
nt1=ntm.findNtuple ("TagCollection1")
# Probe1D returns three values which we assign to a Python list in one go!
limits = []
# Probe the values of phi
limits = nt1.probe1D ("phi","")
print limits
# book histogram
h1 = hm.create1D("10","phi",50,limits[0],limits[1])
```

If the user executes the previous script, the output will look like this:

```
:-) # Open an ntuple
... nt1=ntm.findNtuple ("TagCollection1")
:-) # Probe1D returns three values which we assign to a Python list in one go!
... limits = []
:-) # Probe the values of phi
... limits = nt1.probe1D ("phi","")
:-) print limits
(0.0, 6.2782335752515044, 1000)
:-) # book histogram
... h1 = hm.create1D("10","phi",50,limits[0],limits[1])
```

Lizard tells that the `phi` attribute has a minimum value of `0.0` and a maximum value of `6.27823357525` (the number of rows sampled is `1000`). The `probe1D()` method accepts a cut as well, as shown in this modified version of the previous script:

```
# Open an ntuple
nt1=ntm.findNtuple ("TagCollection1")
# Probe1D returns three values which we assign to a Python list in one go!
limits = []
# This time specify the cut phi>1.0
limits = nt1.probe1D ("phi","phi>1.0")
print limits
# book histogram with limits according to the cut
h1 = hm.create1D("10","phi",50,limits[0],limits[1])
```

Now the output will reflect the use of the cut:

```
:-) # Open an ntuple
... nt1=ntm.findNtuple ("TagCollection1")
:-) # Probe1D returns three values which we assign to a Python list in one go!
... limits = []
:-) # This time specify the cut phi>1.0
... limits = nt1.probe1D ("phi","phi>1.0")
:-) print limits
(1.0007097441033108, 6.2782335752515044, 762)
:-) # book histogram with limits according to the cut
... h1 = hm.create1D("10","phi",50,limits[0],limits[1])
```

Finally it is possible to define the number of rows to take into account when probing. The `probe1D()` would compute (by default) the limits on the first 1000 rows matching, starting from row 0. It is possible to reduce or enlarge the range by specifying two additional arguments as in this example:

```
# Open an ntuple
nt1=ntm.findNtuple ("TagCollection1")
# Probe1D returns three values which we assign to a Python list in one go!
limits = []
# Probe the values of phi on the first 100 rows only
limits = nt1.probe1D ("phi","",0,100)
# Probe the values of phi on up to 1000000 rows starting from row 50
limits = nt1.probe1D ("phi","",50,1000000)
```

Plotting ntuple attributes or attributes' functions

Plotting ntuple attributes (or functions of such attributes) means actually three distinct operations:

- Booking (implicitly or explicitly) an histogram. In the implicit case the system first scans the data to compute histogram limits, then books a default histogram.
- Examine the ntuple data by checking on each row if the cut is satisfied. If so fill the histogram.
- Plot the resulting histogram on screen or in a Postscript file.

In PAW it was possible to perform the three steps in one go using the `NTUPLE/PLOT` command or to break down the process using the commands `HISTO/CREATE/1D`, `NTUPLE/PROJECT` and finally `HISTO/PLOT`. The first command was used typically to get a rough estimate of the distribution, then users would explicitly book the histogram and project the data on it, so to have more control on binning and to speed up the analysis (no need to scan data to get histogram limits). Lizard takes a similar approach, although the user is explicitly given a method to compute min/max of an attributes' function (see the section called "Probing ntuples"). This section will present all the methods and strategies that could be used in Lizard to carry out the task.

Ntuple plot shortcuts

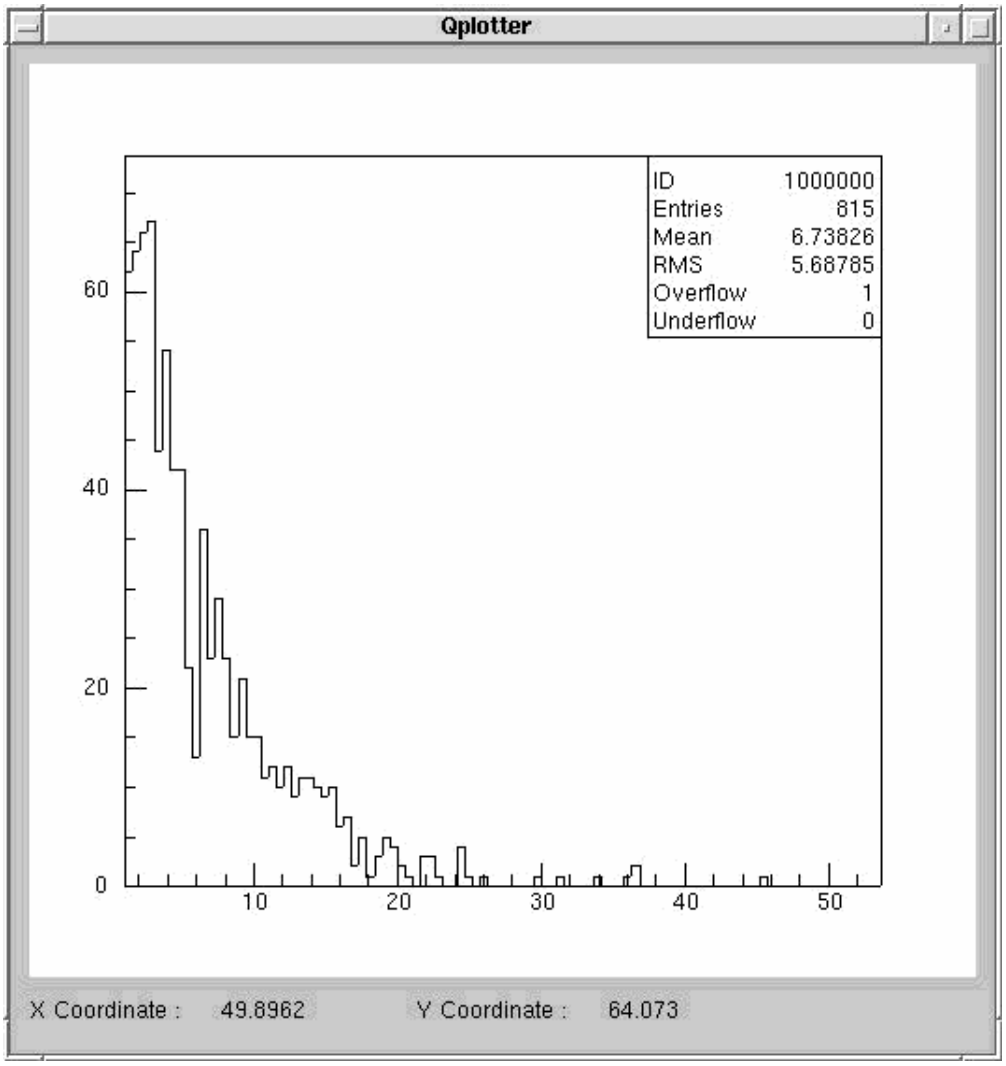
The easiest way to plot ntuple data is to use the `cplot1D()` shortcut, which is a Python function taking aa parameters an ntuple handle, a function to sample in the histogram and a cut as in this example:

```
# Open an ntuple
nt1=ntm.findNtuple ("TagCollection1")
```

```
# Plot the pt distribution with a pt>1 cut
h1=cplot1D(nt1,"pt","pt>1")
# The shortcut returns an histogram handle that can be used e.g. for fitting
# hfit(h1,"E")
```

Lizard would book a suitable histogram, fill it with all `pt` values greater than one and finally plot it. The output would be like in Figure 6.2.

Figure 6.2. Plotting ntuple attribute using shortcut



Important

The shortcut returns the handle of a ‘default’ histogram which will be overridden on the next call to `cplot1D()`. Do not rely on the handle being valid after this happens, e.g.:

```
# Plot the pt distribution with a pt>1 cut
h1 = cplot1D(nt1,"pt","pt>1")
# The shortcut returns an histogram handle that can be used e.g. for fitting
# Now another call to cplot1D() returning the result in a different handle
h2 = cplot1D(nt1,"pt","pt>1")
```

```
# From now on h1 is invalid, because a new default histogram has overridden
# the previous one.
```

Projecting over a `DynamicHistogram`

Dynamic histograms are objects that looks like real histograms but actually keep a copy of the filled entries. This means users need only to specify the number of bins, not the minimum and maximum of the range. Such an histogram can then be used to project ntuple values without knowing the limits on the data, as in this example:

```
# Open an ntuple
nt1=ntm.findNtuple ("TagCollection1")
# Create dynamic 1D histogram 1ith 100 bins
h10 = hm.createDynamic1D(10,"Dynamic 1D histo",100)
# Project the pt distribution with a pt>1 cut
nt1.cproject1D (h10,"pt","pt>1")
# Plot the histogram
hplot(h10)
```

The output would exactly as in the previous example, but now the user has more control on the histogram (e.g. can select the no. of bins). The only drawback of `DynamicHistogram` is the memory footprint: since a dynamic histogram keeps the individual points filled in, it may grow very much when large data samples are analyzed (i.e. more than 100000 entries).

Projecting over a known `Histogram`

This is the way to have full control and maximum efficiency in analysis. In order to do so we need to know in advance the histogram limits, then book a ‘normal’ (i.e. non-dynamic one) histogram and finally project it, as in this example:

```
# Open an ntuple
nt1=ntm.findNtuple ("TagCollection1")
# Retrieve histogram limits for this cut
limits = []
# Probe1D returns three values which we assign to a Python list in one go!
limits = nt1.probe1D ("pt*pt","pt>1")
# book histogram with limits according to the cut
hPt = hm.create1D("10","pt square",50,limits[0],limits[1])
# Project the pt distribution with a pt>1 cut
nt1.cproject1D (hPt,"pt*pt","pt>1")
# Plot the histogram
hplot(hPt)
```

This procedure is completely equivalent to the PAW ‘book, project and plot’ sequence.

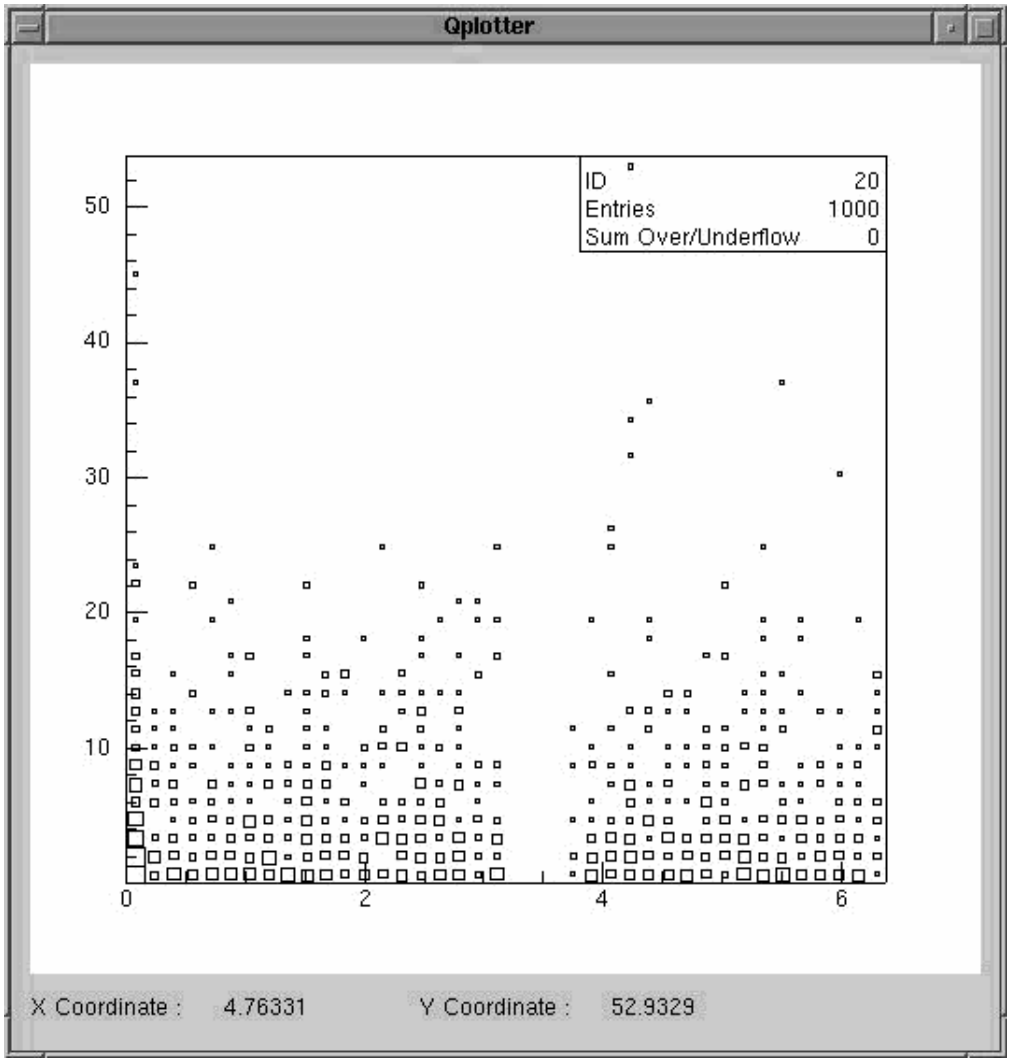
Projecting over a 2D `Histogram`

For the time being there’s no equivalent shortcut to `cplot1D()`, but this could be easily implemented using the method outlined in the the section called "Projecting over a known `Histogram`". Lizard provides a `probe2D()` to allow users to probe two attributes (or functions of attributes) in one go, so it’s straightforward to modify the previous script in this way:

```
# Open an ntuple
nt1=ntm.findNtuple ("TagCollection1")
# Find histogram limits on phi and pt
lims = []
lims = nt1.probe2D("phi","pt","phi>-1")
# Book histogram. Notice the small quantity added to high limits to
# avoid missing the maximum values
h20=hm.create2D(20,"Phi vs. pt",40,lims[0],lims[1]+0.1,40,lims[2],lims[3]+0.1)
# Project the attributes with the fake cut "phi>-1"
nt1.cproject2D (h20,"phi","pt","phi>-1")
# Plot the histogram with BOX option
hplot(h20)
```

The output of the script is shown in Figure 6.3.

Figure 6.3. Projecting over a 2D histogram



Scatter plots

Scatter plots are diagrams in which each point in the 2D phase-space is visually represented by a marker (usually a dot) in the corresponding X-Y position on the output device. There are two typical use cases for scatter plot:

- X-Y plot of a small data sample where it’s very important to have accurate information about the position of each point
- ‘Cloud’ plot of a very large dataset. The exact position of the points is less important, since the main information is the overall correlation among the variables.

Lizard defined two types of scatter plot objects:

- ‘dynamic’ scatter plot, where the original X-Y coordinates are kept as pairs of values
- ‘bitmap’ scatter plot, where the phase space is divided in a very thin grid (e.g. 1024X1024 cells) and each cell is either `on` or `off` depending on having being ‘hit’ by an X-Y point.

The first type of scatter plot gives always the exact position of the points, but it consumes memory if the number of points is very high. The second type of scatterplot uses always the same amount of memory, even for a billion points dataset , but has a "quantization" effect of the order of 1/1000 (i.e. points closer than 1/1000 of the range cannot be separated). This is not such a big problem as it may seem: most screen can’t anyway separate two such points and if this precison is required for points belonging to very large datasets, it’s still possible to reduce the range of the scatter plot in the desired region of the phase-space.

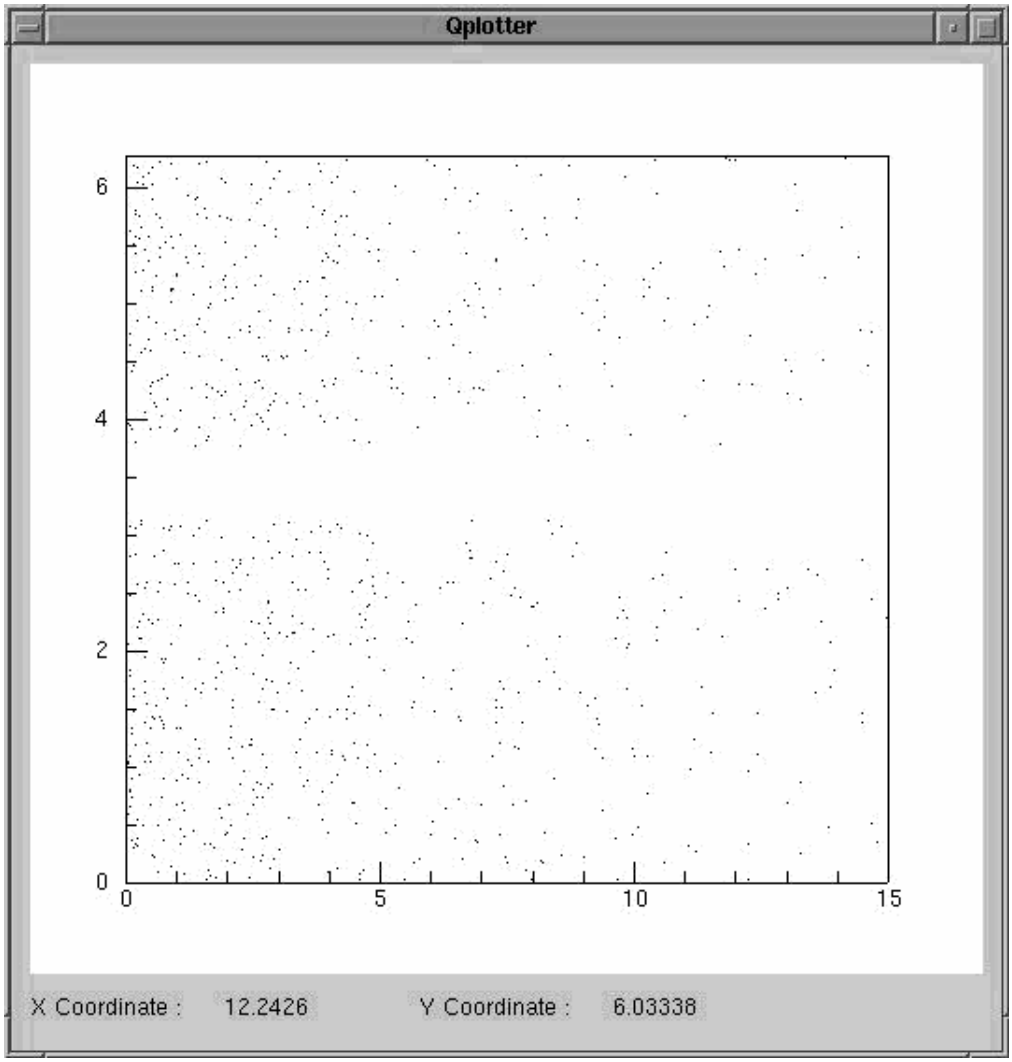
The main advantage of these data structures is that they allow manipulations fo the plot (i.e. zooming) without going back to the original data (something that was not available in PAW).

In the current version of Lizard scatter plots are created by default as 1024X1024 bitmaps and they can be plotted as shown in the next script:

```
# Open an ntuple
nt1=ntm.findNtuple ("TagCollection1")
# Create a factory for scatter plots
scatFact = createIScatterFactory()
# Create a scatter plot with x range [0,20] and Y range [0,6.28]
scat = scatFact.create(0,15.,0.,6.28)
# Fill the scatte rplot
nt1.scatter2D (scat,"pt","phi","")
# Plot it
pl.plot(scat)
```

The resulting scatter plot is shown in Figure 6.4.

Figure 6.4. Scatter plot



More on C++ expressions used by ntuple methods

As explained briefly in the section called " Operations on ntuples ", Lizard does not interpretate the cuts or the functions accepted by the ntuple methods and expressed using the C++ syntax. The C++ code is compiled, loaded in memory and "glued" with the `Ntuple` component. Lizard then loops over the ntuple entries and execute the cuts and functions as part of its algorithm. This section will give a few more details about this topic.

Caching expressions

Although the compilation of the code on the fly is pretty fast (thanks to the wide use of abstract interfaces), a few optimizations have been devised in order avoid as much as possible even this small overhead. If the user specifies an ntuple methods accepting one or more C++ expressions, the code is compiled and the resulting shared library is cached for possible reuse, i.e. if the same cut or function is reused, no compilation will take place anymore. The following script would show such behaviour when run the first time in Lizard:

```
# Open an ntuple
nt1=ntm.findNtuple ("TagCollection1")
# Code is compiled
```

```
print "Compile & execute code"
start = time()
h1 = cplot1D(nt1,"pt","pt>1")
first = time()-start
# Code is not compiled anymore: expressions are in cache
print "Do not compile & execute code"
start = time()
# Compute ratio
h1 = cplot1D(nt1,"pt","pt>1")
print "Execution is ", first/(time()-start), " times faster"
```

The output shows clearly the advantage of not recompiling code.

```
Compile & execute code
Do not compile & execute code
INFO: 1D-histogram with ID 1000000 has been deleted.
Execution is  14.9392915407  times faster
```

In the current implementation the cache stores the 100 most recently used expressions.

Using parameters to avoid compiling code

Although caching is useful when the cuts or expressions are ‘stable’ (e.g. when running an already tuned analysis script over new datasets), it’s no help when the user is still in the ‘exploratory’ phase, playing with cuts and struggling with corrections. As an example let’s imagine the user wants to try four different cuts in `pt` just to see the quality of the resulting fit. The simplest approach would be to project with 4 different cuts, as in the following script:

```
# Open an ntuple
nt1=ntm.findNtuple ("TagCollection1")
# Create histogram
h1 = hm.create1D("10","phi",50,0,50)

# Project 4 times
nt1.cproject1D(h1,"pt","pt<30.0")
hfit(h1,"E")
h1.reset()

nt1.cproject1D(h1,"pt","pt<35.")
hfit(h1,"E")
h1.reset()

nt1.cproject1D(h1,"pt","pt<40.")
hfit(h1,"E")
h1.reset()

nt1.cproject1D(h1,"pt","pt<45.")
hfit(h1,"E")
```

This would mean four code compilations, since the cut is syntactically different every time. In order to avoid this, Lizard allows to define ntuple parameters that can be used in the cut and updated at execution time, thus avoiding to recompile just because a numeric value has changed:

```
# Open an ntuple
nt1=ntm.findNtuple ("TagCollection1")
# Create histogram
h1 = hm.create1D("10","phi",50,0,50)
```

```
# Create ntuple parameter set
pars = ntm.createParameters ()

cuts = [30.0,35.,40.,45.]
for i in range (0,4) :
   # Set parameter ptMin
   pars.set("ptMin",cuts[i])
   # Use parameter ptMin
   nt1.cproject1D(h1,"pt","pt<ptMin")
   hfit(h1,"E")
   h1.reset()
```

Chapter 7. Using the `Fitter` component

Table of Contents

Introduction

Lizard fit components are based either on the MINUIT or on the NAG C Numerical Libraries backend minimizer engines. It is possible to switch from one to the other at startup time, by specifying a parameter on the command line. Whichever engine is used, Lizard exposes uniform fit interfaces both for "simple" fitting (i.e. fitting with well known model functions, such as Gaussian, Exponential or Polynomial) and for "complex" fitting with complex model functions.

Fitting histograms using a shortcut

The `hfit()` shortcut can be used to fit a histogram using a few simple functions, i.e. `Gaussian`, `Exponential`, `Polynomial(0)`,`Polynomial(1)`. For all this functions it's possible to figure out reasonable starting values for the parameters, so that the user doesn't have to provide additional information as long as the dataset is close enough to the model function. Higher degree polynomial and combination of functions are not allowed with the shortcut, since it's difficult to figure out such starting values for the parameters (see NEXT SECTION for more general simple fitting). The use of the shortcut has already been introduced in a previous example:

```
# Open an ntuple
nt1=ntm.findNtuple ("TagCollection1")
# Create histogram
h1 = hm.create1D("10","phi",50,0,50)
# Project
nt1.cproject1D(h1,"pt","pt<30.0")
# Fit: this will work (pt is exponential)
hfit(h1,"E")
# Fit: these will not work!
hfit(h1,"G")
hfit(h1,"P0")
hfit(h1,"P1")
```

Fitting with simple functions and sum of functions (general case)

If the `hfit()` shortcut is not suitable for the fitting task at hand, Lizard can fit any dataset contained in a `Vector` object. The conversion from histogram to vector can either be done behind the scene (as in the case of the shortcut) or explicitly by the user. If the data to fit are already in a `Vector`, no further conversion is required.

The basic course of action for fitting can be summarized in the following list:

- Create an instance of `Fitter` component
- Give the `Fitter` the set of points to fit
- Specify the model function to fit (`Gaussian`, `Exponential`,`Polynomial(N)` or a sum of those)
- Set the starting values for the function parameters
- Start the fitting algorithm (so far only chi square fitting is exposed to the user, although the underlying package is capable of Maximum Likelihood fitting as well).
- Collect the result
- Delete the `Fitter` component

Tip

Be aware that even if the dataset is properly distributed according to the model function, most fitting algorithm would not converge if no suitable starting values for parameters are defined, so this step is almost mandatory.

The most important methods of the `Fitter` component are listed in Table 7.1.

Table 7.1. Most important `Fitter` methods

Method	Arguments	Remark
chiSquareFit()	None	Execute a chi square fit on the given set of points using the model function previously defined.
fittedVector()	None	Retrieve a Lizard`Vector` containing the values of the model function computed in the corresponding input points (usually to overlay it to the set of input points).
printParameters()	None	Print the current values of the fit parameters
printResult()	None	Print the results of the fit
setData()	Vector	Define the set of input points as a Lizard`Vector`
setModel()	string	Define the model function (combination of {G,E,Pn})
setParameter()	string,double	Set the starting value of the parameter to a given quantity

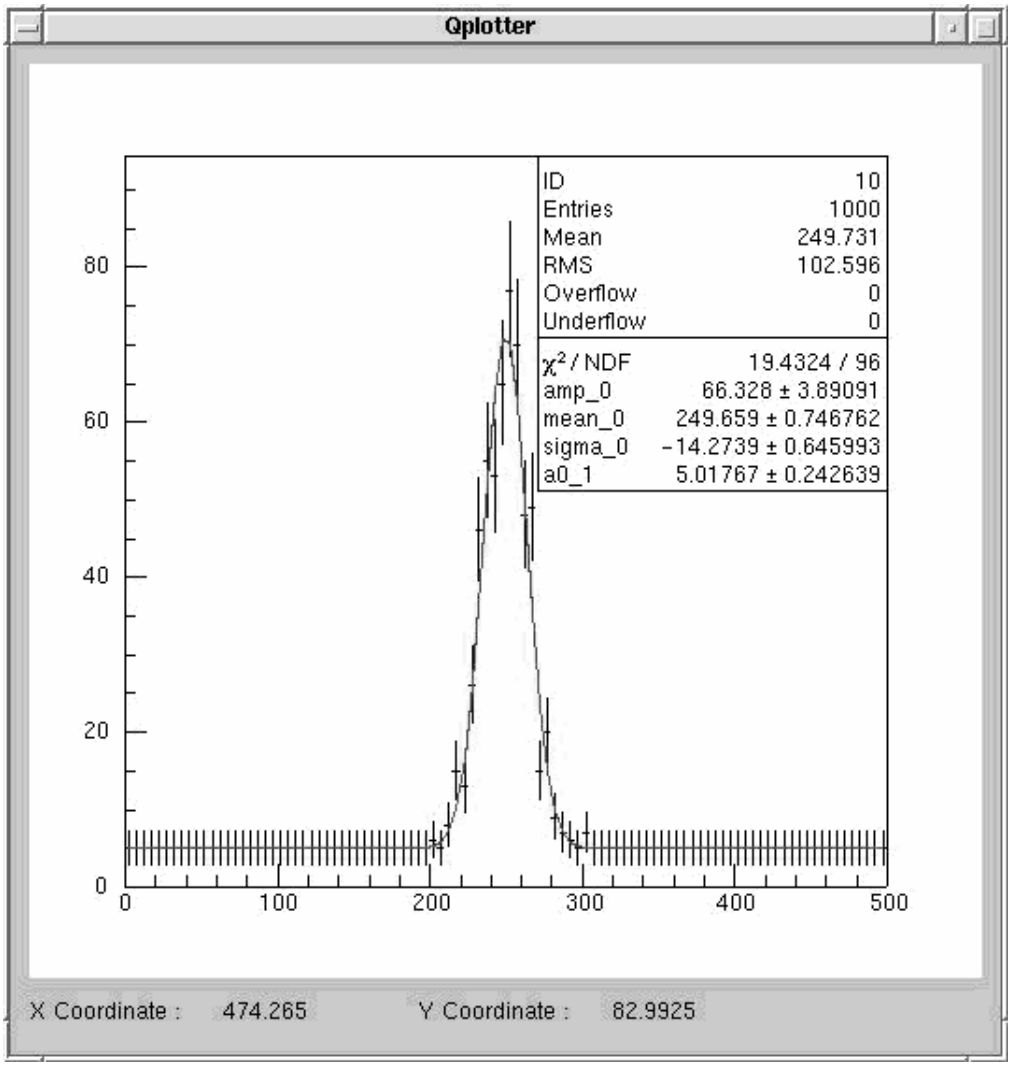
The following script shows how to fit a histogram with a combination of Gaussian plus Polynomial(0). Doing so it introduces most of the methods defined beforehand:

```
# Create an histogram
```

```
histo=hm.create1D(10,"test 1",100,0., 500.)
# Fill it with a gaussian + pedestal
for i in range(0.,500.):
    histo.fill(random.gauss (250,15))
    histo.fill(i,1)

# The empty line is necessary (to close the for loop)!
# Declare a Fitter
fit=Fitter()
# Convert histogram to vector
v1=vm.from1D(histo)
# Define the dataset
fit.setData(v1)
# Set the model
fit.setModel("G+p0")
# Define starting values for parameters
fit.setParameter("mean_0",240)
fit.setParameter("amp_0",180)
fit.setParameter("sigma_0",20)
fit.setParameter("a0_1",30)
# Print starting values
#fit.printParameters ()
# Execute chi square fit
fit.chiSquareFit()
# print results
fit.printResult()
# Get back the fitted curve
_vfit=fit.fittedVector()
# Overlay points and fit
pl.dataOption ("representation","error")
pl.plot(v1,_vfit)
pl.reset()
del fit
```

Notice the use of Python's random number generator `random.gauss()`. The output is shown in Figure 7.3.

Figure 7.3. Fitting with two functions



Parameters are named according to well known abbreviations depending on the model. When functions are composed in a sum, the position of the term in the overall function (from 0 to N-1) is appended with an underscore before it, as summarized in Table 7.2.

Table 7.2. Name of fit parameters

Function	Arguments	Remark
G	mean amp sigma	Single Gaussian curve
E	amp slope	Single Exponential curve
Pn	a0 a1 ... an-1	Single Polynomial curve of degree n
G+G	mean_0 amp_0 sigma_0 mean_1 amp_1 sigma_1	Two Gaussian curves
G+E+P0	mean_0 amp_0 sigma_0 amp_1 slope_1 a0_2	Gaussian plus Exponential plus Polynomial(0)
P0+G+E	a0_0 mean_1 amp_1 sigma_1 amp_2 slope_2	Polynomial(0) plus Gaussian plus Exponential

The easiest way to get the names of the parameters is to declare a new fitter and set the model:

Lizard will print out the parameter names with the proper naming, e.g.:

```
fit=Fitter()
fit.setModel("G+E+p0")
del fit
```

would produce this output:

```
:-) fit=Fitter()
:-) fit.setModel("G+E+p0")

ExtendedFitter setup

*data not defined*

Parameters defined in the model

   Index            Name

       0           amp_0
       1          mean_0
       2         sigma_0
       3         slope_1
       4           amp_1
       5            a0_2
:-) del fit
```

More about fit parameters

Fit parameters are objects, so they can be manipulated using their methods, summarized in Table 7.3.

Table 7.3. Methods on `FitParameter` objects

Method	Arguments	Remark
value()	None	Return parameter value (meaningful only after fitting)
error()	None	Return Hessian error on the parameter(meaningful only after fitting)
start()	None	Return starting value
step()	None	Return step (change between iterations)
isFixed()	None	Return whether the parameter is fixed
isBound()	None	Return whether the parameter is bound
lowerBound()	None	Return parameter's lower bound
upperBound()	None	Return parameter's upper bound
setStart()	double	Change starting point
setStep()	double	Change step
setBounds()	double,double	Set parameter boundaries. Equal boundaries make the parameter fixed
noUpperBound()	None	Return a value corresponding to +infinity
noLowerBound()	None	Return a value corresponding to -infinity

The following scripts makes a fit, then retrieve one parameter (the `mean`), prints its attributes and then change them using the `FitParameter` methods:

```
####### Preliminary fit to get a reasonable value for parameter #######
# Create an histogram
histo=hm.create1D(10,"test 1",50,0., 500.)
# Fill it with a gaussian
for i in range(0.,500.):
    histo.fill(random.gauss (250,15))

# The empty line is necessary (to close the for loop)!
# Declare a Fitter
fit=Fitter()
# Convert histogram to vector
v1=vm.from1D(histo)
# Define the dataset
fit.setData(v1)
# Set the model
fit.setModel("G")
# Define starting values for parameters
fit.setParameter("mean",240)
fit.setParameter("amp",180)
fit.setParameter("sigma",20)
# Execute chi square fit
fit.chiSquareFit()
####### End of preliminary fit #######


# A Python function to examine a parameter
def examinePar (fitPar) :
   print "Value=",mean.value() , " Error=",mean.error()
   print "Start=",mean.start() , " Step=",mean.step()
   if ( mean.isFixed()) :
      print "Parameter is fixed"
   if ( mean.isBound()) :
      print "Parameter bound between ",mean.lowerBound()," and ",mean.upperBound
```

```
   else :
      print "Parameter is free"


# Retrieve the mean parameter
mean = fit.fitParameter ("mean")

# Print out all information about the mean parameter
print "Examine mean parameter"
examinePar(mean)

# Now change the parameter
print ""
print "Change starting value,step and make it fixed"
mean.setStart(100.)
mean.setStep(0.5)
mean.setBounds(100.,100.)

# Print out all information about the mean parameter
print "Examine mean parameter"
examinePar(mean)

# Release fixed parameter
print ""
print "Release fixed parameter"
mean.release()
examinePar(mean)

# Set open boundaries
mean.setBounds(mean.noLowerBound(),110)
print ""
print "Bound between [-infinity,110]"
examinePar(mean)

# Set open boundaries
mean.setBounds(90,mean.noUpperBound())
print ""
print "Bound between [90,infinity]"
examinePar(mean)

del fit
```

Ignoring the fit output, that is not relevant in this context, the result of the script would be something like that:

```
...
Examine mean parameter
Value= 248.624079318  Error= 0.63102261826
Start= 240.0  Step= 1.0
Parameter is free

Change starting value,step and make it fixed
Examine mean parameter
Value= 248.624079318  Error= 0.63102261826
Start= 100.0  Step= 0.5
Parameter is fixed
Parameter bound between  100.0  and  100.0

Release fixed parameter
Value= 248.624079318  Error= 0.63102261826
Start= 100.0  Step= 0.5
Parameter is free
```

```
Bound between [-infinity,110]
Value= 248.624079318  Error= 0.63102261826
Start= 100.0  Step= 0.5
Parameter bound between  1e+20  and  110.0

Bound between [90,infinity]
Value= 248.624079318  Error= 0.63102261826
Start= 100.0  Step= 0.5
Parameter bound between  90.0  and  1e+20
```

Notice the use of a Python function to print out information about a generic parameter.

Changing the fit range

The `Fitter` component allows to exclude individual points or sets of points (ranges) from the input dataset. The methods available for this purpose are listed in the section called "Changing the fit range".

Table 7.4. `Fit` methods to adjust the input dataset

Method	Remark
excludePoint(*i*); integer *i*;	Exclude the given point from fitting
excludeRange(*low*, *high*); integer *low*; integer *high*;	Exclude points in the range from fitting
includePoint(*i*); integer *i*;	Include the given point for fitting
includeRange(*low*, *high*); integer *low*; integer *high*;	Include points in the range for fitting

The following scripts show how to do this. First a Gaussian distribution is sampled in a histogram, then a spike is injected to decrease its quality:

```
# Create an histogram
histo=hm.create1D(10,"test 1",100,0., 500.)
# Fill it with a gaussian
for i in range(0.,500.):
   histo.fill(random.gauss (250,15))

# Insert a spike at coordinate X=230
for i in range(0.,50.):
   histo.fill(230)

# Plot it
hplot(histo)
```

We now need to locate the spike in order to remove it: in this case we know it’s in coordinate X=230, so we can simply use the histogram methods that maps a coordinate to the corresponding bin:

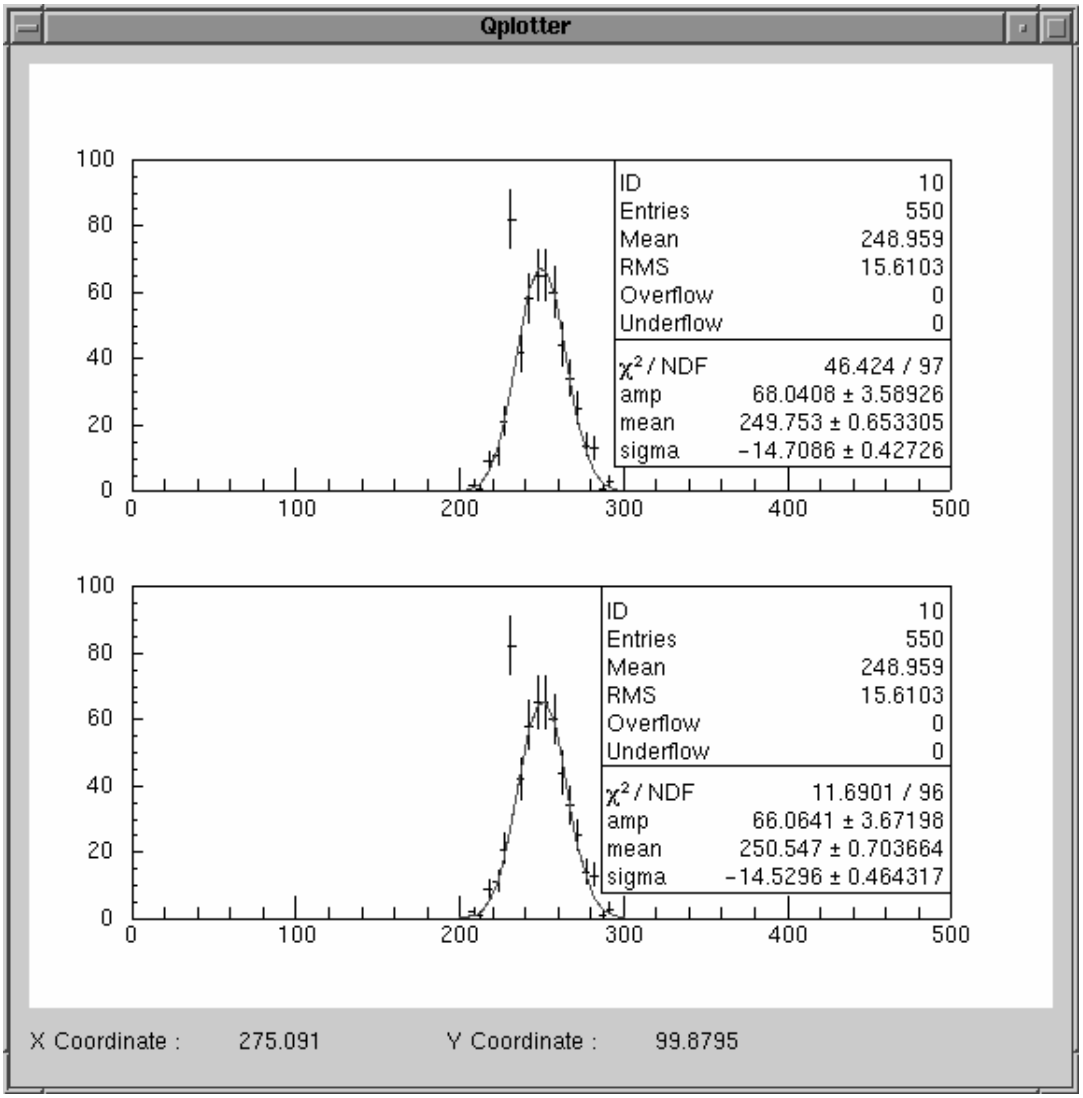
```
:-) histo.coordToIndex(230)
46
```

(in a real case it would suffice to move the cursor to the spike and read the coordinate on the `Plotter` window). This script will first fit the histogram with the spike, the redo the fit after removing the point `46`.

```
pl.zone(1,2)
# Declare a Fitter
fit=Fitter()
# Convert histogram to vector
v1=vm.from1D(histo)
# Define the dataset
fit.setData(v1)
# Set the model
fit.setModel("G")
# Define starting values for parameters
fit.setParameter("mean",240)
fit.setParameter("amp",180)
fit.setParameter("sigma",20)
# Execute chi square fit
fit.chiSquareFit()
# Get back the fitted curve
_vfit=fit.fittedVector()
# Overlay points and fit
pl.dataOption ("representation","error")
pl.plot(v1,_vfit)

# Now re-fit excluding the point
fit.excludePoint(46)
# Execute chi square fit
fit.chiSquareFit()
# Get back the fitted curve
_vfit=fit.fittedVector()
# Overlay points and fit
pl.dataOption ("representation","error")
pl.plot(v1,_vfit)
pl.reset()
del fit
```

The result of the script is shown in Figure 7.4.

Figure 7.4. Excluding points from fit



Notice the improved chi square and the decrement in the number of degrees of freedom.

Chapter 8. Using the `Plotter` component

Table of Contents

Introduction

Lizard `Plotter` component is based on the Qplotter package (which in turn uses the Qt toolkit) and allows users to draw graphics on screen and in Postscript files. Although the underlying Qplotter package is completely object oriented, the `Plotter` component tries to expose a simpler user interface which is somehow "similar" to PAW sequential approach. This is done in the hope to ease the transition of former PAW users to the new system and may eventually evolve in a more modern interface.

The main features of the `Plotter` component are summarized in this list:

- The output on the screen is like the output on the page (i.e. no special treatment for printing).
- The screen/page can be divided in `Zones`. So far only zones with the same dimensions are allowed in a page (although the underlying Qplotter package supports any type of zones).
- Each zone can contain several dataset (curves)
- Datasets are instances of either `Vector` or `IScatter2D` classes.
- It is possible to select individual zones at any time (or rely on the default strategy that advances the current zone at each new plot).
- Several options/properties are available ot the page/zone/dataset level.
- The screen window contains a scene graph of the diagrams, thus it is possible to resize the window without losing the output.
- Postscript output can be produced at any moment during the preparation of the graphics (no special commands to start Postscript output, just take a Postscript snapshot of the current screen).

Working with zones

The `Plotter` component allows to define the number of zones in a page. This is implemented by the `zone()` method which takes as arguments the number of zones along X and the number of zones along Y, as shown in the following script:

```
# Just one zone (default)
pl.zone(1,1)
# Two zones on X, one on Y
pl.zone(2,1)
# One zone on X, two on Y
pl.zone(1,2)
# Four zones
pl.zone(2,2)
```

After the execution of the `zone()` method, the first zone becomes the current one.

Plotting vectors on zones

As explained beforehand, the objects users can plot are instances of either `Vector` or `IScatter2D` classes. Of course shortcuts such as `hplot()` extend this by silently converting other objects (histograms in this case) to a vector, but this does not change the rule. Since vectors are more used than Scatter plots, most scripts in this chapter will create vector(s) and plot those. Lizard provides two methods to draw a vector in a zone, as summarized in Table 8.4.

Table 8.4. `Plotter` methods to draw a vector

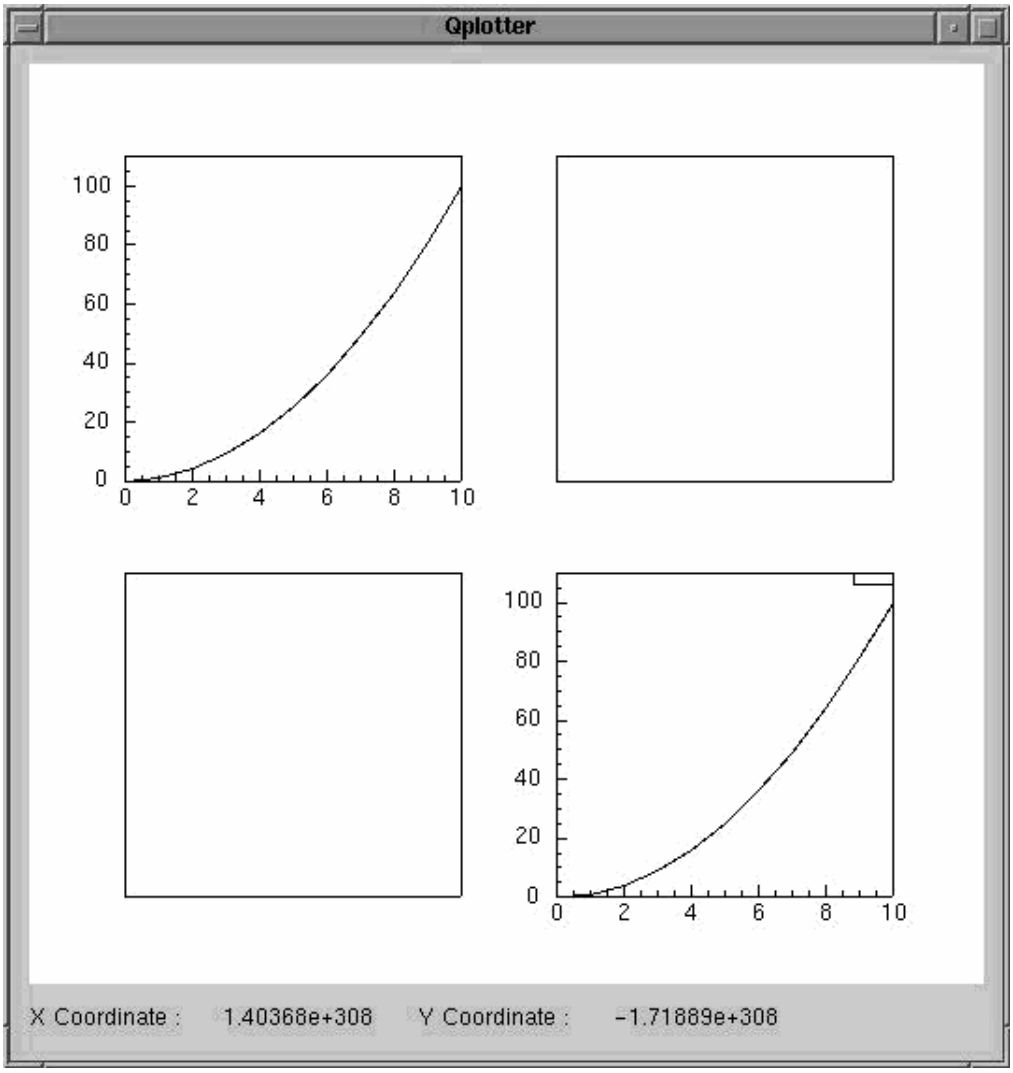
Method	Remark
plot(*v1*, *v2*); Vector *v1*; Vector *v2* == 0;	Plot one or two vectors on the current zone. The second vector will be drawn as a red line (handy for showing fit results). The second vector is optional. The current zone is incremented after plotting.
overlay(*v1*, *selectedZone*); Vector *v1*; integer *selectedZone* == -1;	Plot one vector on the selected zone. The selected zone is optional, default being the last zone used. The current zone is not incremented after plotting.

As an example the following script will create a `(2,2)` zone layout and plot one vector on the current zone and the same vector on the fourth zone:

```
# These are Python lists
xvals = [0.,1.,2.,3.,4.,5.,6.,7.,8.,9.,10.]
yvals = [0.,1.,4.,9.,16.,25.,36.,49.,64.,81.,100.]
# Create a Lizard vector
v1 = vm.fromPy(xvals, yvals)

# Four zones
pl.zone(2,2)
# Plot on the current zone (1)
pl.plot(v1)
# Plot on  zone 4
pl.overlay(v1,4)
```

The result of the script is shown in Figure 8.2.

Figure 8.2. Plotting vectors



The `overlay()` method can be used in a more general way to draw different vectors on the same zone as explained in the section called "Plotting vectors using several zones". To plot a Scatter plot rather than a vector, see the section called "Scatter plots".

Data representations and properties

Lizard supports several representations for 1D and 2D datasets (histogram, line, box to name a few). Each curve in a zone can have his own representation and it is possible to customize its appearance by changing properties between calls of the `plot()` method.

Plotter properties are defined according to the entity they apply to (e.g. page, zone or curve) and to their "domain", i.e. text or graphics, as summarized in this list:

- Zone properties (e.g. logarithmic scales)
- Dataset (curve) properties (e.g. marker shape)
- Draw properties (e.g line color)
- Text properties (e.g. font info)

Properties are usually defined by key/value pairs, where the key identifies the property and the value defines the attribute, e.g.:

```
# Show histogram statistics
pl.zoneOption ("option","stats")
# Hide histogram statistics
pl.zoneOption ("option","nostats")
```

The `Plotter` method `listOptions()` enumerates all available properties:

```
:-) pl.listOptions ()

Options for Zones

---------------------- Zone Option List -------------------
The following options for Zone objects are available
via the setProperty method taking two strings as parameter:
setProperty (string name, string value).
("option"," xlinear xlog stats nostats ylinear ylog xaxisgrid yaxisgrid ")
("coordinates"," locked free ")
("mirroraxis"," yes no ")
-----------------------------------------------------------



Options for Datasets (curves)

No Dataset available

Draw style options

---------------------- DrawStyle Option List -------------------
The following options for DrawStyle objects are available
via the setProperty method taking two strings as parameter:
setProperty (string name, string value).
("linecolor","value")
("fillcolor","value")
("linewidth","value")
("lineshape"," none solid dash dot dashdot dashdotdot ")
("fillstyle"," none solid dense94 dense88 dense63 dense50 dense37 dense12
dense06 horiz vert cross bdiag fdiag diagcross ")
-----------------------------------------------------------



Text style options

---------------------- TextStyle Option List -------------------
The following options for TextStyle objects are available
via the setProperty method taking two strings as parameter:
setProperty (string name, string value).
("fontname","value")
("fontsize","value")
("color","value")
("bold"," yes no ")
("italic"," yes no ")
-----------------------------------------------------------


----------------------  Available fonts  -------------------

adobe-courier
adobe-helvetica
adobe-new century schoolbook
adobe-times
```

```
application
avantgarde
bitstream-courier
bitstream-terminal
...
zapf chancery
------------------------------------------------------------
```

Zone properties

Zone properties are defined by calling the method `zoneOption()` method which accepts two arguments, a key and a value:

Table 8.5. Zone properties

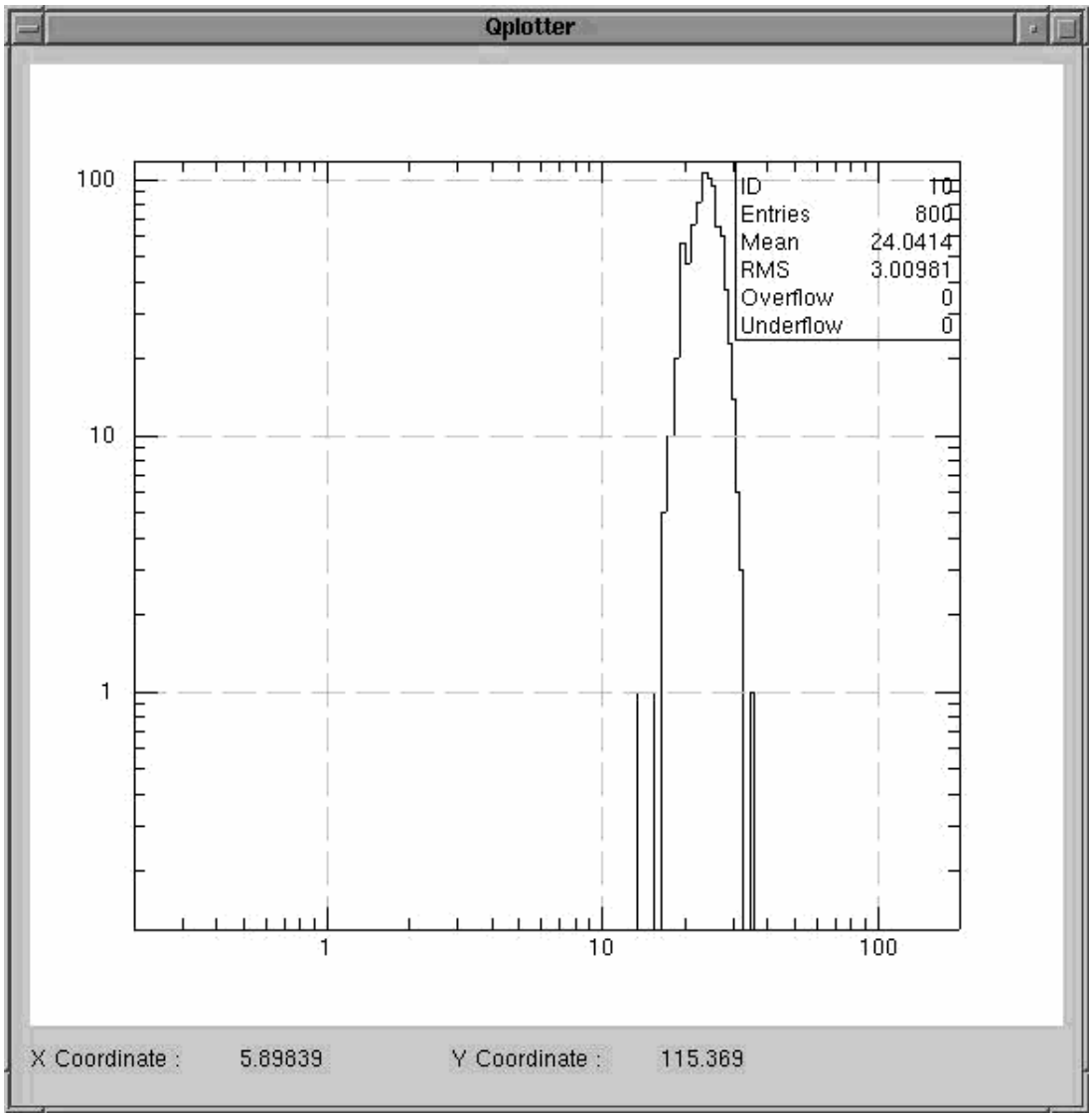
Key	Value	Remark
option	xlinear xlog ylinear ylog	Define whether one of the axis is linear(default) or logarithmic.
option	xaxisgrid yaxisgrid	Define whether the grid is activated (default no)
option	stats nostats	Define whether the summary information is shown (default stats, i.e. shown).
coordinates	locked free	Define whether the coordinate systems is locked by the first curve (default) or changes according to each new curve overlayed.
mirroraxis	yes	Define whether the the ticks on the axis should be mirrored on the top and right side of the zone (default no mirroring).

As an example, the following script exercises some zone properties:

```
# Create an histogram
histo=hm.create1D(10,"test 1",50,0., 48.)
# Fill it with a gaussian
for i in range(0.,800.):
    histo.fill(random.gauss (24,3))

# Set min/max on zone 1
pl.setMinMaxX (0,200,1)
# Enable statistics
pl.zoneOption ("option","stats")
# Log scale on both axis
pl.zoneOption ("option","xlog")
pl.zoneOption ("option","ylog")
# Grid on both axis
pl.zoneOption ("option","xaxisgrid")
pl.zoneOption ("option","yaxisgrid")
# Mirror tickmarks on top and right-hand edges
pl.zoneOption ("mirroraxis","yes")
# plot the histogram
hplot(histo)
# Reset properties
pl.reset()
```

The script produces the output in Figure 8.3.

Figure 8.3. Using zone options



Dataset (curve) properties

Dataset properties are defined by calling the method `dataOption()` method which accepts two arguments, a key and a value:

Table 8.6. Dataset properties

Key	Value	Remark
representation	error line histo marker errormark smooth hfilled	Select the representation for the 1D dataset
representation	box color	Select the representation for the 2D dataset
legend	string	The legend of the curve in the summary information (e.g. `"Monte Carlo"` ,`"Data 2001"`).
mtype	none rect diamond triangle dtriangle utriangle ltriangle rtriangle xcross ellipse	The symbol used for the marker
msize	integer	The size of the marker

Dataset representations

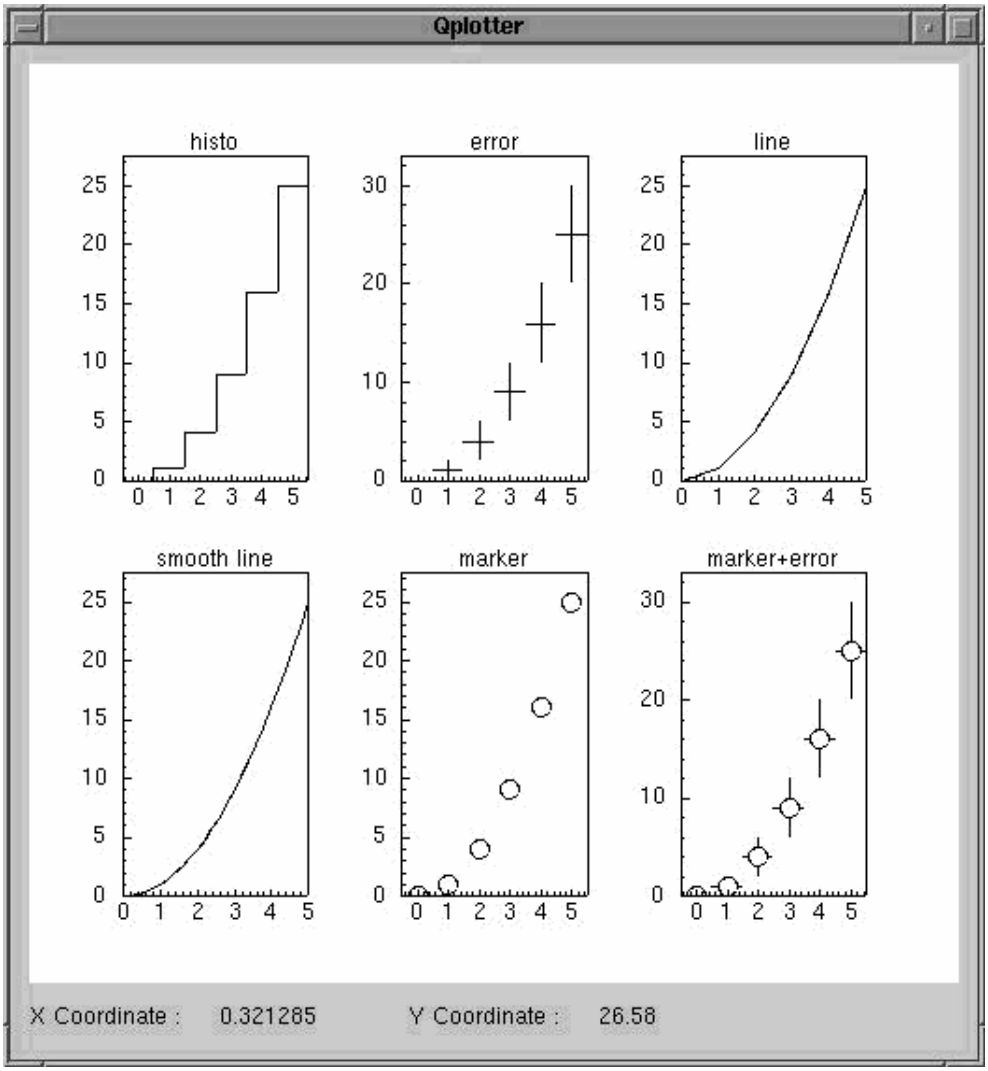
By changing the `representation` property, it is possible to select the way a set of points will be represented on the screen. The following script shows how to select different representations for the same 1D dataset:

```
# These are Python lists
# A list from 0 to 5
xvals = [x*1. for x in range(0.,6.)]
# A list with values squared
yvals = [x*x for x in xvals]
# 'Error' on X i.e. half binwidth
exvals = [0.5 for x in xvals]
# Error on Y, i.e. sqrt(Y)
eyvals = [sqrt(y) for y in yvals]

# Create Lizard vector
v1=vm.fromPy(xvals,yvals,exvals,eyvals)

# Zone settings
pl.zone(3,2)
pl.zoneOption ("option","nostats")

# That's the default representation
pl.dataOption("representation","histo")
pl.plot(v1)
pl.zoneTitle("histo")
# Error
pl.dataOption("representation","error")
pl.plot(v1)
pl.zoneTitle("error",2)
# Line
pl.dataOption("representation","line")
pl.plot(v1)
pl.zoneTitle("line",3)
# Smooth line
pl.dataOption("representation","smooth")
pl.plot(v1)
pl.zoneTitle("smooth line",4)
# Marker
pl.dataOption("representation","marker")
pl.plot(v1)
pl.zoneTitle("marker",5)
# Marker + error
```

```
pl.dataOption("representation","errormark")
pl.plot(v1)
pl.zoneTitle("marker+error",6)
```

The script outcome is shown in Figure 8.4.

Figure 8.4. Different representations for 1D datasets



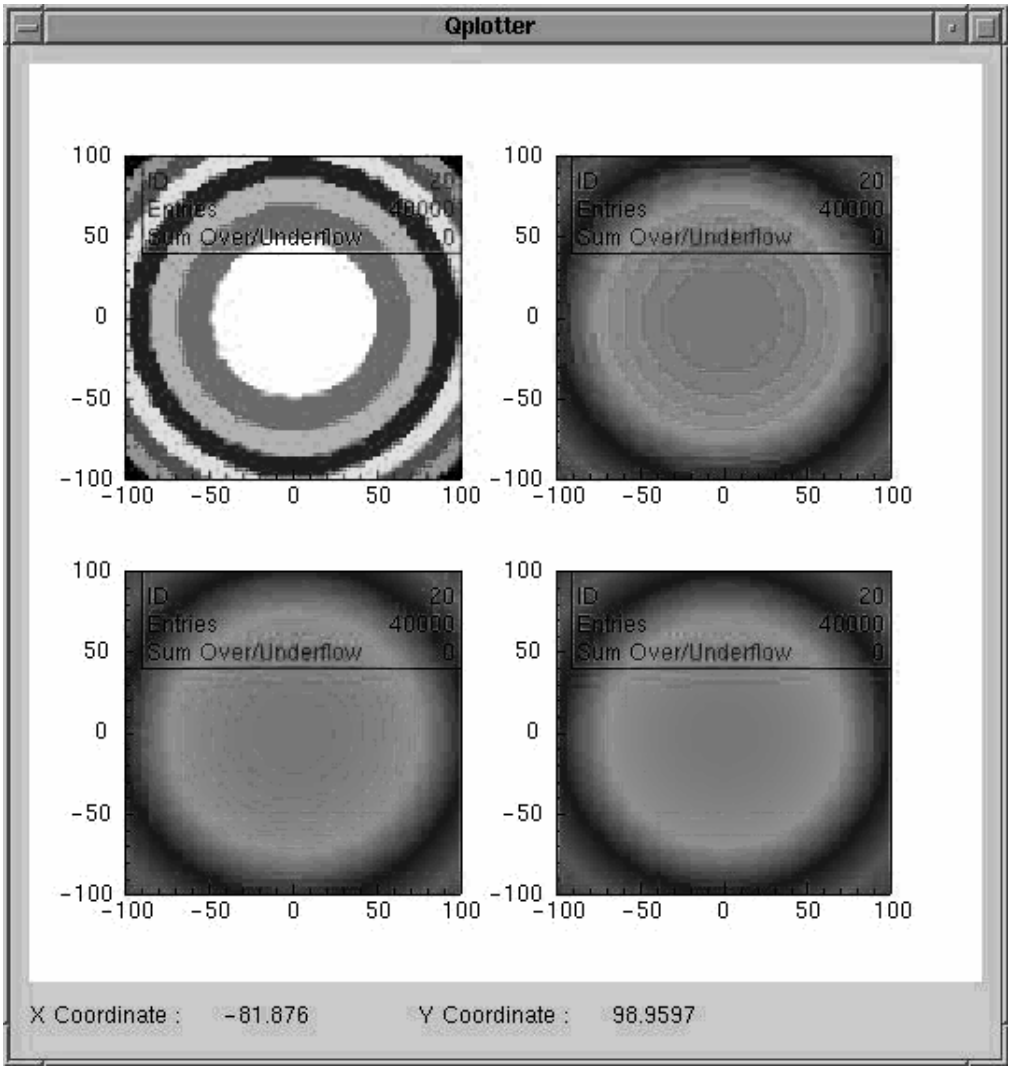
Similarly we can change the representation for a 2D dataset, as in this example which plots the same dataset using the color representation first with eigth basic colors, then with geographical palettes with increasing resolution:

```
# Show the Color plot feature
# book histogram
h3 = hm.create2D("20","phi",100,-100,100,100,-100,100)

for x in range(-100,100.,1):
    for y in range(-100,100.,1):
        h3.fill(x,y,x*x+y*y)

# Two zones
pl.zone(2,2)
```

```
# Plot histogram as color plot
# Select color representation
pl.dataOption ("representation","color")
# Plot with eight basic colors
hplot(h3)
# Plot with geographical palette of sixteen colors
pl.dataOption ("colorlevels","16")
hplot(h3)
# Plot with geographical palette of thirty-two colors
pl.dataOption ("colorlevels","32")
hplot(h3)
# Plot with geographical palette of 240 colors (maximum)
pl.dataOption ("colorlevels","240")
hplot(h3)
# Reset
pl.reset()
pl.dataOption ("representation","box")
```

The script output is shown in Figure 8.5.

Figure 8.5. Different representations for 2D datasets



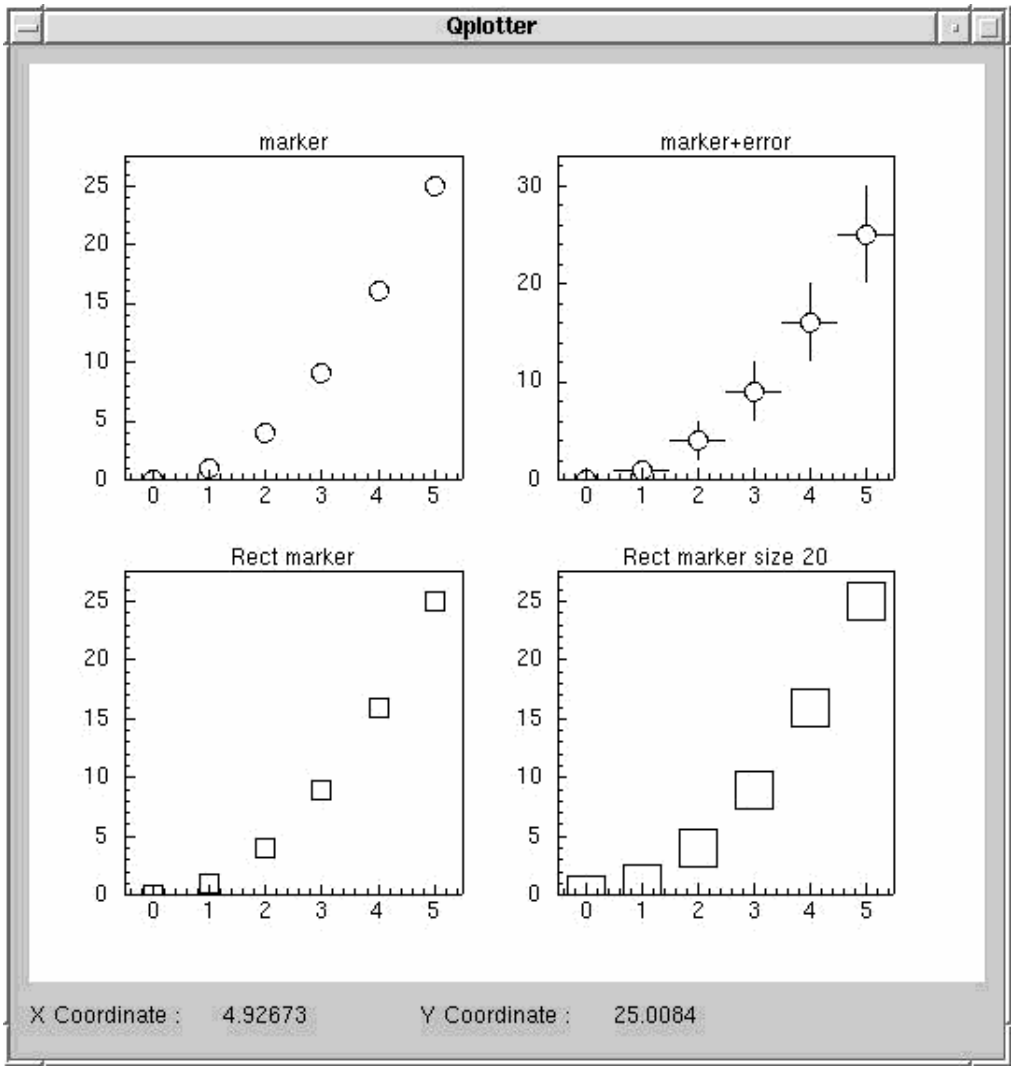
Changing markers

Lizard allows the user to change the marker symbol and its size via the `mtype` and `msize` properties. The marker type can be chosen among the list given in Table 8.6. The marker size is defined in pixels (...), the default being `10`. The following script shows how to use these features:

```
# These are Python lists
# A list from 0 to 5
xvals = [x*1. for x in range(0.,6.)]
# A list with values squared
yvals = [x*x for x in xvals]
# 'Error' on X i.e. half binwidth
exvals = [0.5 for x in xvals]
# Error on Y, i.e. sqrt(Y)
eyvals = [sqrt(y) for y in yvals]

# Create Lizard vector
v1=vm.fromPy(xvals,yvals,exvals,eyvals)

# Zone settings
pl.zone(2,2)
pl.zoneOption ("option","nostats")

# Marker
pl.dataOption("representation","marker")
pl.plot(v1)
pl.zoneTitle("marker",1)
# Marker + error
pl.dataOption("representation","errormark")
pl.plot(v1)
pl.zoneTitle("marker+error",2)

# Rectangular marker
pl.dataOption("representation","marker")
pl.dataOption ("mtype","rect")
pl.plot(v1)
pl.zoneTitle("Rect marker",3)

# Rectangular marker size 20
pl.dataOption ("msize","20")
pl.plot(v1)
pl.zoneTitle("Rect marker size 20",4)

pl.reset()
```

The resulting output is shown in Figure 8.6.

Figure 8.6. Markers



Notice that it is possible to have filled markers as well, as explained in the section called "Fill area properties".

Style properties

Before showing more properties of the dataset, it’s time to introduce so called "Style properties". These properties allow the user to modify the appearence of a curve in terms of line shape, line color, line width, fill color etc. Style options are changed via the `dataStyle()` method of the plotter. The following table summarizes the key/value pairs accepted by such method:

Table 8.7. Data style properties

Key	Value	Remark
linecolor	blue,white,black,red,green,yellow,magenta,cyan,darkgray,lightgray,gray darkred,darkgreen,darkblue,darkcyan,darkmagenta,darkyellow	Color of the line used to draw the curve
linewidth	value	Width of the line used to draw the curve. Width is in pixels, default is 1.
lineshape	none solid dash dot dashdot dashdotdot	Shape of the line used to draw the curve.
fillcolor	blue,white,black,red,green,yellow,magenta,cyan,darkgray,lightgray,gray darkred,darkgreen,darkblue,darkcyan,darkmagenta,darkyellow	Color of the fill area (for a filled representation)
fillstyle	none solid dense94 dense88 dense63 dense50 dense37 dense12 dense06 horiz vert cross bdiag fdiag diagcross	Style of the filled area (density or hatch style)

Line properties

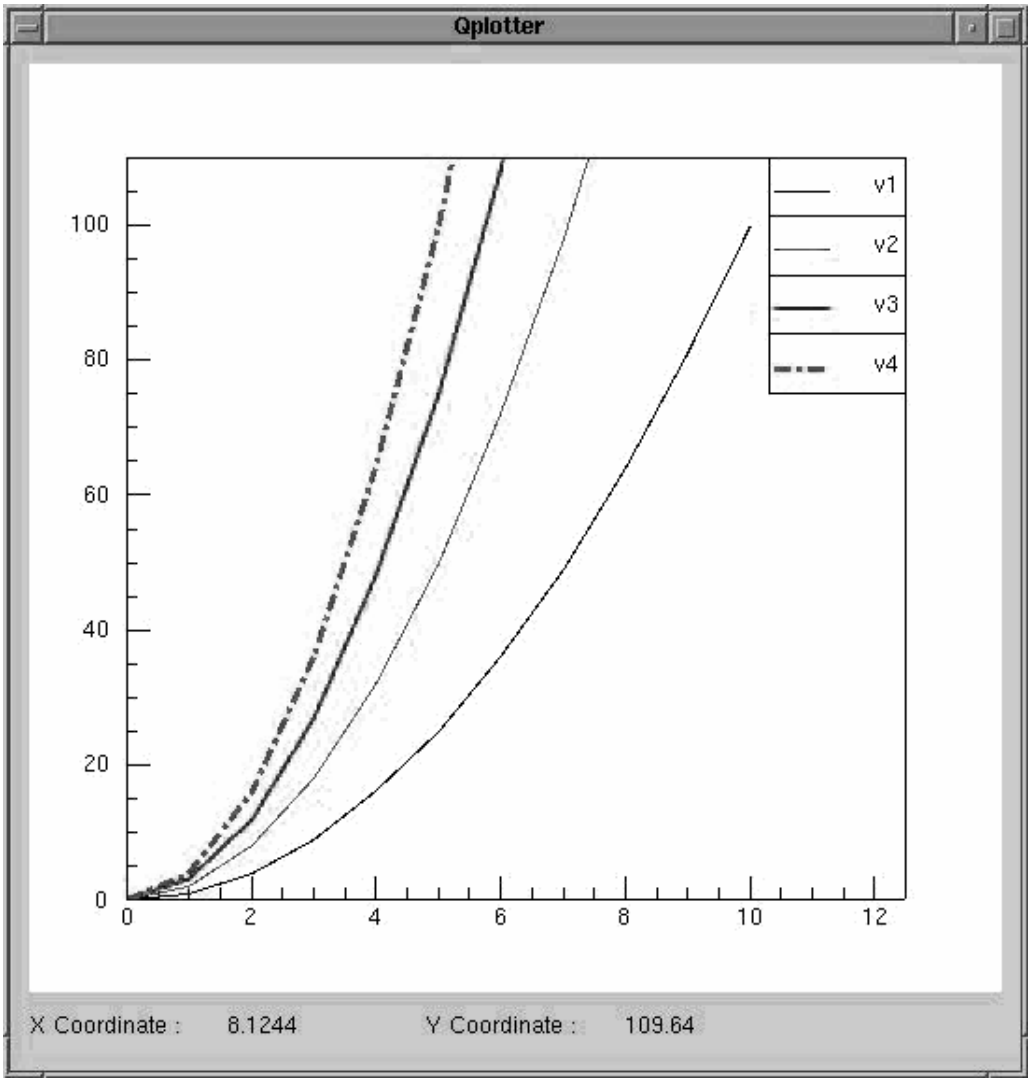
This script shows how to use style options to change the way a curve is drawn. Notice the use of another property of the curve, the `legend` property mentioned in Table 8.6:

```
# These are Python lists
xvals = [0.,1.,2.,3.,4.,5.,6.,7.,8.,9.,10.]
yvals = [0.,1.,4.,9.,16.,25.,36.,49.,64.,81.,100.]
# Create Lizard vectors
v1 = vm.fromPy(xvals, yvals)
v2 = vm.fromPy(xvals, yvals)
v3 = vm.fromPy(xvals, yvals)
v4 = vm.fromPy(xvals, yvals)
v2.mul(2.)
v3.mul(3.)
v4.mul(4.)

# Use line representation
pl.dataOption("representation","line")
# Set zone min max to make legends more readable
pl.setMinMaxX(0,12.5,1)

# First curve (default style)
pl.dataOption ("legend","v1")
pl.plot(v1)

# Change line color
pl.dataStyle("linecolor","blue")
pl.dataOption ("legend","v2")
pl.overlay(v2)

# Change line color, width
```

```
pl.dataStyle("linecolor","green")
pl.dataStyle("linewidth","2")
pl.dataOption ("legend","v3")
pl.overlay(v3)

# Change line color, width, shape
pl.dataStyle("linecolor","red")
pl.dataStyle("linewidth","3")
pl.dataStyle("lineshape","dashdot")
pl.dataOption ("legend","v4")
pl.overlay(v4)

# Reset data style
pl.reset()
```

The script output is shown in Figure 8.7.

Figure 8.7. Use of data style and legend properties



The effect of the `legend` property is to associate a given comment to a specimen of the line used to draw the curve. A similar behaviour is implemented for markers as well (see the section called "Fill area properties".

Fill area properties

There are basically two properties related to a fill area:

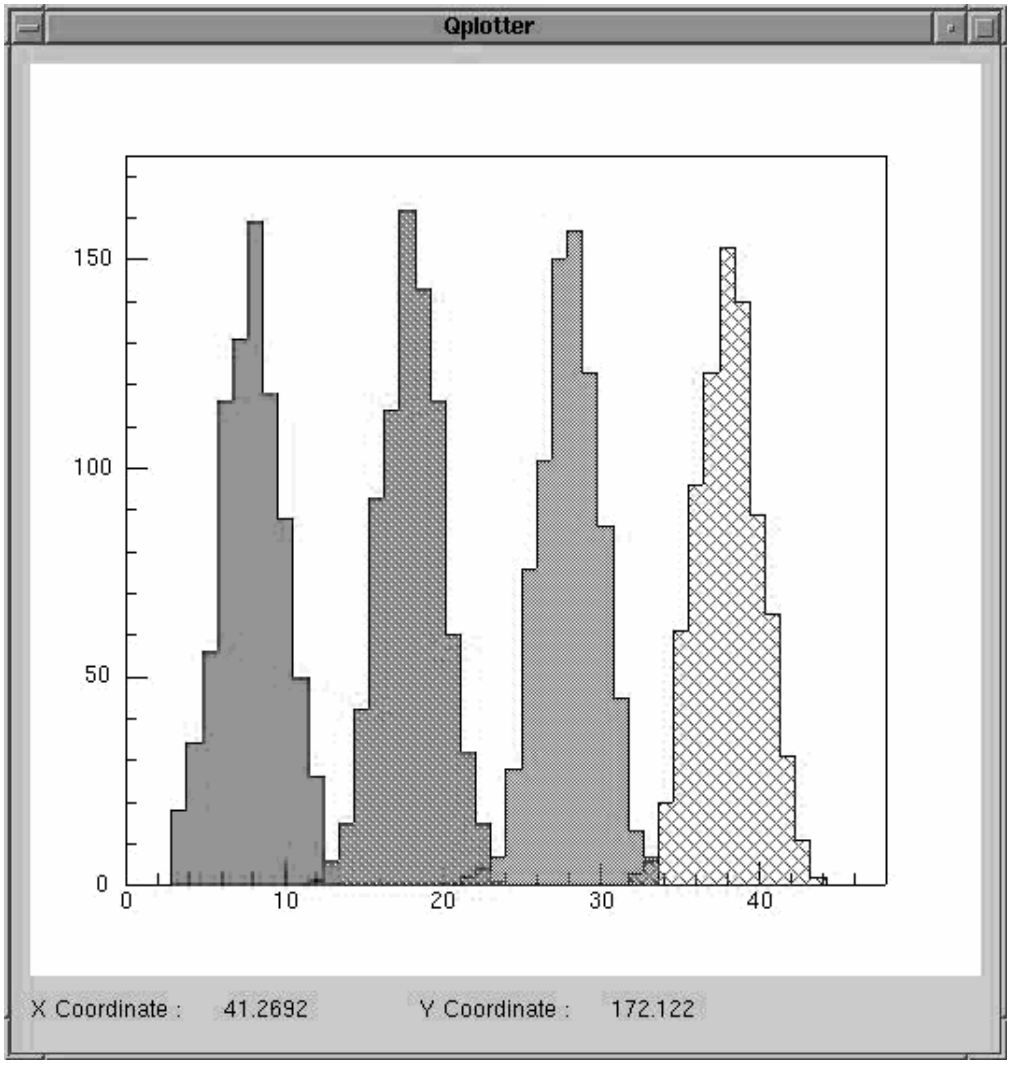
- The fill color (e.g. blue,red,green,etc.)
- The fill style (e.g. none,solid,horiz ,etc.)

The default values are `white` color and `none` pattern, i.e. a fill area is not visible by default. In order to make the area visible, both properties should be set to meaningful values, as in the following script taht shows four similar histograms filled with the same color and different styles:

```
# Create histograms
h1=hm.create1D(10,"Curve 1",50,0., 48.)
h2=hm.create1D(20,"Curve 2",50,0., 48.)
h3=hm.create1D(30,"Curve 3",50,0., 48.)
h4=hm.create1D(40,"Curve 4",50,0., 48.)
# Fill them with gaussians
for i in range(0.,800.):
    h1.fill(random.gauss (8,2))
    h2.fill(random.gauss (18,2))
    h3.fill(random.gauss (28,2))
    h4.fill(random.gauss (38,2))

# No statistics
pl.zoneOption ("option","nostats")

# Plot as filled histograms
pl.dataOption("representation","hfilled")
# Fill color is blue for all of them
pl.dataStyle("fillcolor","blue")

# Overlay histograms with different patterns (black is the line color)
pl.dataStyle("fillstyle","solid")
hplot(h1)
pl.dataStyle("fillstyle","dense88")
hplot(h2,"o","black")
pl.dataStyle("fillstyle","dense50")
hplot(h3,"o","black")
pl.dataStyle("fillstyle","diagcross")
hplot(h4,"o","black")

# Reset properties
pl.reset()
```

The script output is shown in Figure 8.8.

Figure 8.8. Use of fill area properties



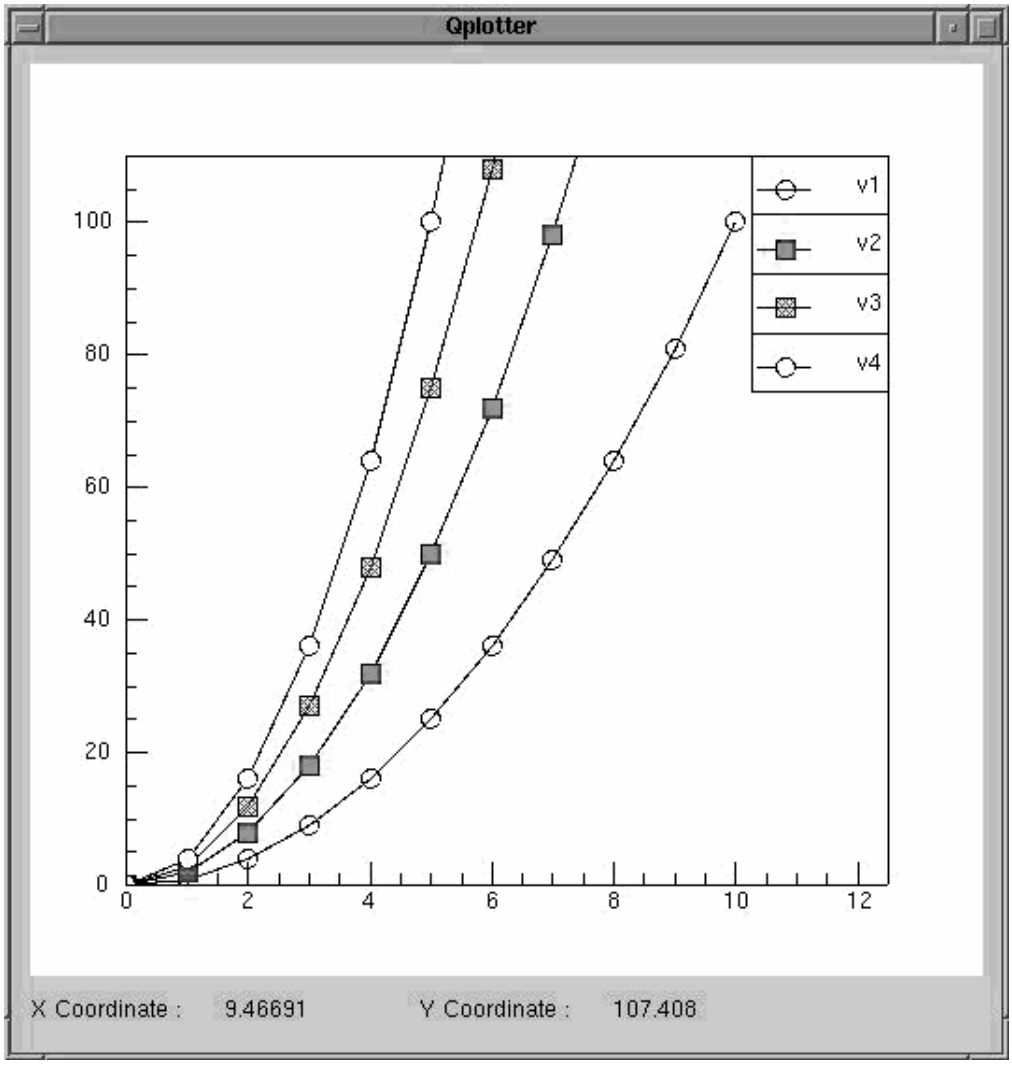
In this case the `hfilled` representation has been used to fill the area inside the histograms, but the same techniques can be used to fill markers:

```
# these are Python lists
xvals = [0.,1.,2.,3.,4.,5.,6.,7.,8.,9.,10.]
yvals = [0.,1.,4.,9.,16.,25.,36.,49.,64.,81.,100.]
# Create Lizard vectors
v1 = vm.fromPy(xvals, yvals)
v2 = vm.fromPy(xvals, yvals)
v3 = vm.fromPy(xvals, yvals)
v4 = vm.fromPy(xvals, yvals)
v2.mul(2.)
v3.mul(3.)
v4.mul(4.)

# Set zone min max to make legends more readable
pl.setMinMaxX(0,12.5,1)

# Connecting lines
pl.dataOption("representation","line")
pl.plot(v1)
pl.overlay(v2)
pl.overlay(v3)
pl.overlay(v4)
```

```
# Use marker representation
pl.dataStyle("lineshape","solid")
pl.dataOption("representation","marker")
# First curve (default style)
pl.dataOption ("legend","v1")
pl.overlay(v1)

# Change marker shape, fill color, fill style
pl.dataOption ("mtype","rect")
pl.dataOption ("legend","v2")
pl.dataStyle("fillcolor","blue")
pl.dataStyle("fillstyle","solid")
pl.overlay(v2)

# Change fill color, fill style
pl.dataOption ("legend","v3")
pl.dataStyle("fillcolor","red")
pl.dataStyle("fillstyle","dense37")
pl.overlay(v3)

# Default marker white filled
pl.dataOption ("mtype","")
pl.dataStyle("fillcolor","white")
pl.dataStyle("fillstyle","solid")
pl.dataOption ("legend","v4")
pl.overlay(v4)

# Reset data style
pl.reset()
```

The new output is presented in Figure 8.9.

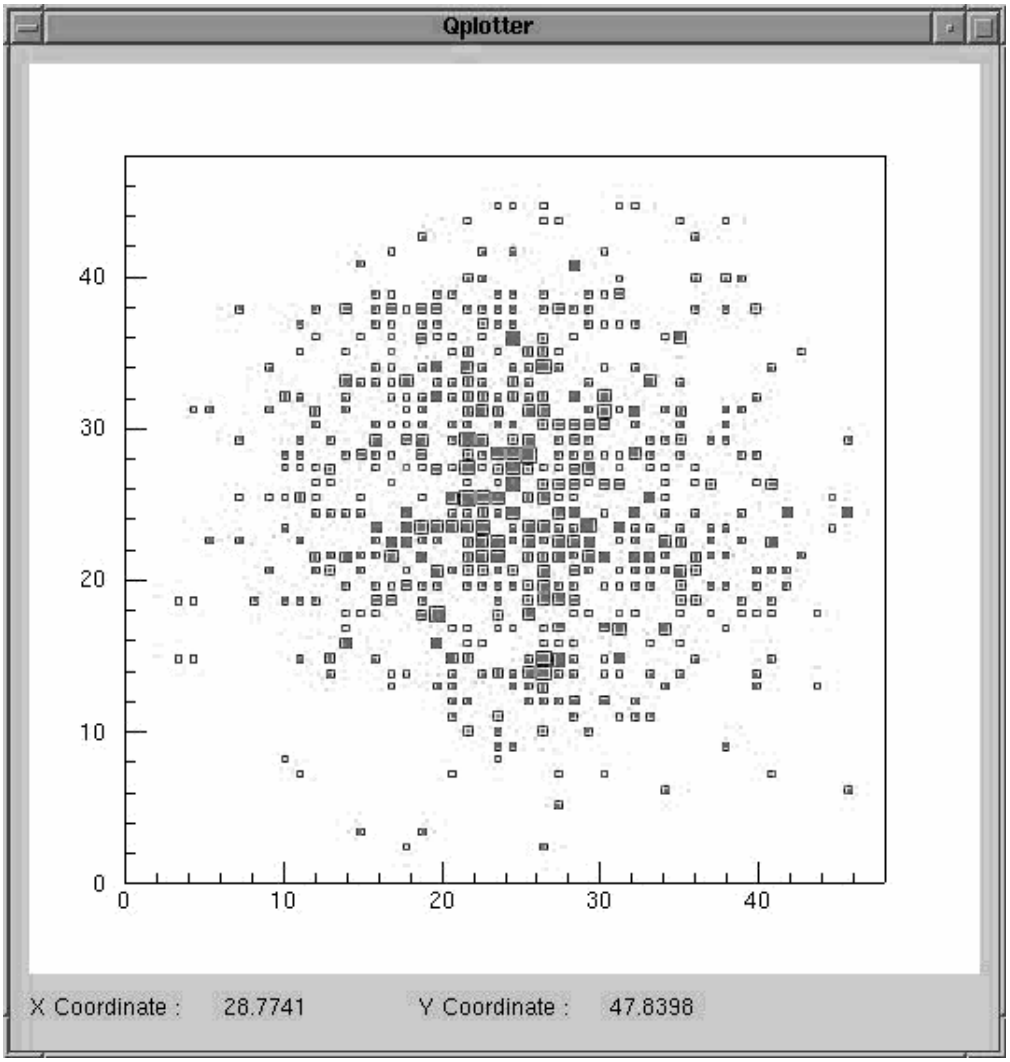
Figure 8.9. Filled markers



Finally the same properties can be used with the box 2D representation, as done by this script:

```
# Create histogram
h1=hm.create2D(10,"2D Gaussian",50,0., 48.,50,0., 48.)
# Fill it with a gaussian
for i in range(0.,800.):
    h1.fill(random.gauss (25,8),random.gauss (25,8))

# No statistics
pl.zoneOption ("option","nostats")

# Fill color is blue
pl.dataStyle("fillcolor","yellow")
pl.dataStyle("linecolor","blue")

# Filled box plot
pl.dataStyle("fillstyle","solid")
hplot(h1)

# Reset properties
pl.reset()
```

Figure 8.10. Filled boxes



Working with text

Lizard allows users to add titles to a page or a zone, as well as to put text in arbitrary positions. Moreover it's possible to change the appeareance of text by modifying plotter's `TextStyle`.

Coordinates' spaces

In order to place text on the page or zone, it is important to understand the basics of the coordinate systems in Lizard. From the user point of view there are basically two coordinate systems:

- Page coordinate system (210X210, unit is millimeter)
- Zone coordinate system (according to the zone coordinates)

The Page coordinate system starts from the bottom left corner of the drawing area (the area with white background color). Coordinates go left-to-right and bottom-to-top from `0` to `210` (`210` is the size of the drawing area in millimeter when printed on A4 paper).

The Zone coordinate systems starts from the bottom left corner of the zone (i.e. the origin of axis). Coordinates go left-to-right and bottom-to-top from minimum to the maximum values in X and Y. This means that the coordinates are affected by the change on the min/max value on one of the axis

(exactly as in PAW).

Adding titles and text

The relevant methods are summarized in Table 8.8.

Table 8.8. `Plotter` methods to add text

Method	Arguments	Remark
`zoneTitle`(*`title`*, *`zone`*); `string` *`title`*; `integer` *`zone`*;	Set the title of the given zone (default is zone 1)	
`pageTitle`(*`title`*); `string` *`title`*;	Set the title of the page	
`zoneText`(*`Xcoordinate`*, *`Ycoordinate`*, *`text`*, *`zone`*); `double` *`Xcoordinate`*; `double` *`Ycoordinate`*; `string` *`text`*; `integer` *`zone`*;	Add text at position (Xcoordinate,Ycoordinate) on the given zone (default is zone 1)	
`pageText`(*`Xcoordinate`*, *`Ycoordinate`*, *`text`*); `integer` *`Xcoordinate`*; `integer` *`Ycoordinate`*; `string` *`text`*;	Add text at position (Xcoordinate,Ycoordinate) on the page (default is zone 1)	

The following script shows how to use those methods:

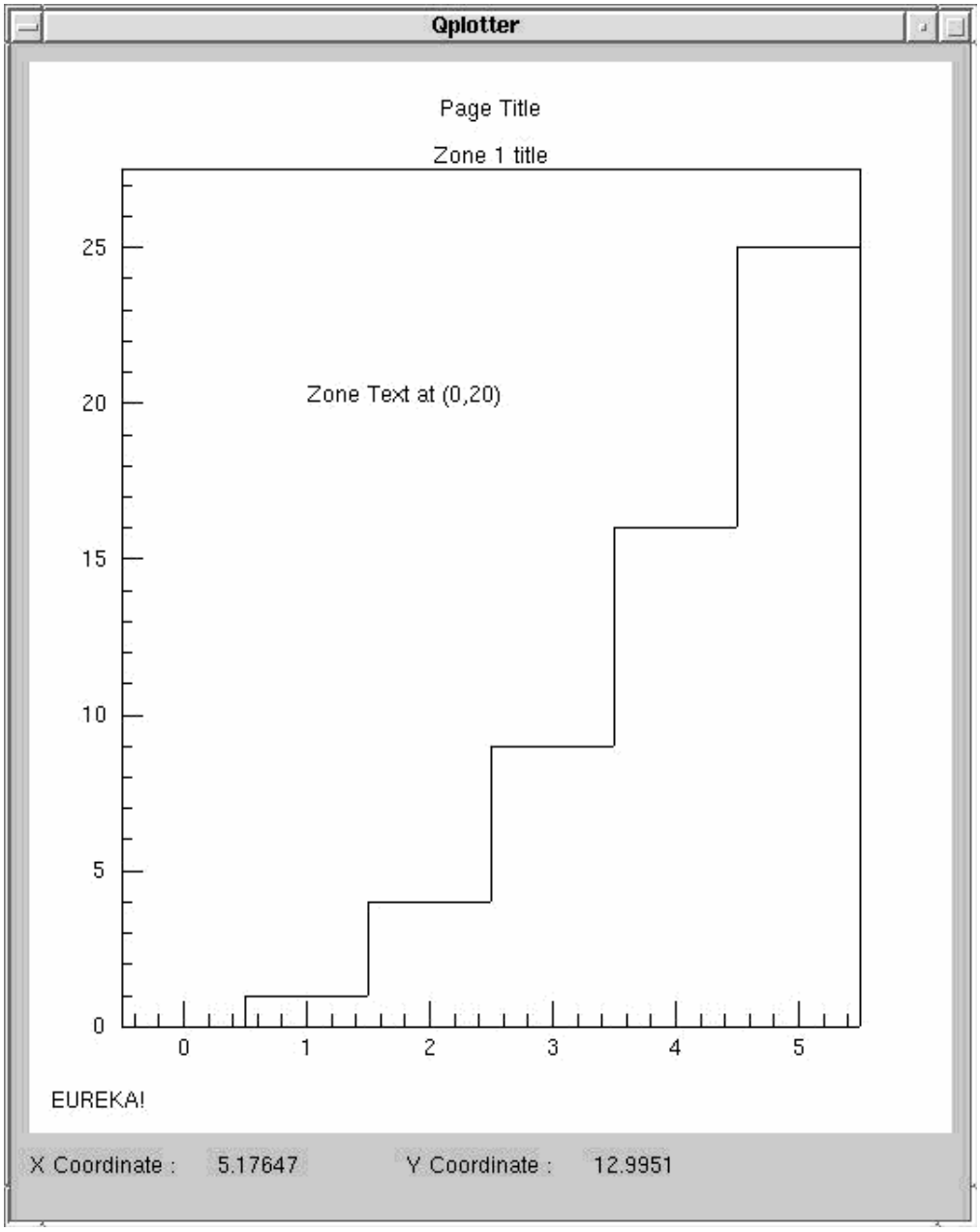
```
# These are Python lists
# A list from 0 to 5
xvals = [x*1. for x in range(0.,6.)]
# A list with values squared
yvals = [x*x for x in xvals]
# 'Error' on X i.e. half binwidth
exvals = [0.5 for x in xvals]
# Error on Y, i.e. sqrt(Y)
eyvals = [sqrt(y) for y in yvals]
# Create Lizard vector
v1=vm.fromPy(xvals,yvals,exvals,eyvals)

# Plot data
pl.plot(v1)
# Page title
pl.pageTitle("Page Title")
pl.zoneTitle("Zone 1 title",1)
# Put additional text one zone 1
pl.zoneText  (1,20,"Zone Text at (0,20)",1)
# Put text on the page
pl.pageText  (5,5,"EUREKA!")
pl.reset()
```

Important

The call to `pageTitle()` should happen after the first call to `plot()`, since the `plot()` method may cleanup the page.

The script produces the output in Figure 8.11.

Figure 8.11. Text and titles



Showing text in Zone coordinates

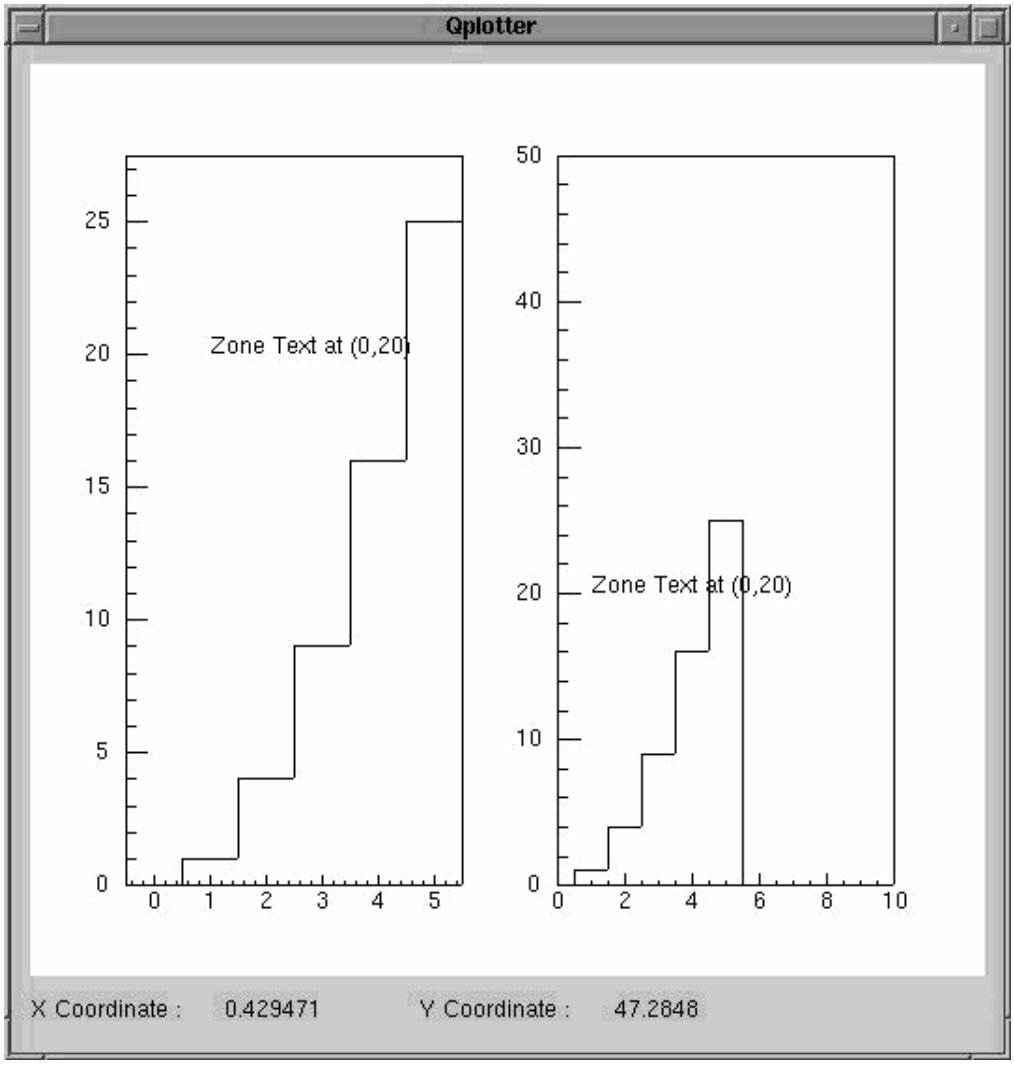
As mentioned in the the section called "Coordinates' spaces", text placed according to the Zone coordinate system 'moves' depending on the limits defined by the user. The following script draws the same curve and the same text in two zones, the latter with user-defined minimum and maximum limits:

```
# These are Python lists
# A list from 0 to 5
xvals = [x*1. for x in range(0.,6.)]
# A list with values squared
```

```
yvals = [x*x for x in xvals]
# 'Error' on X i.e. half binwidth
exvals = [0.5 for x in xvals]
# Error on Y, i.e. sqrt(Y)
eyvals = [sqrt(y) for y in yvals]

# Create Lizard vector
v1=vm.fromPy(xvals,yvals,exvals,eyvals)

# Zone settings
pl.zone(2,1)

# Zone 1
pl.plot(v1)
# Put additional text one zone 1
pl.zoneText  (1,20,"Zone Text at (0,20)",1)

# Reset all properties
# Zone 2
# Change min/max to show impact on Zone coordinates
pl.reset ()
pl.setMinMaxX(0,10,2)
pl.setMinMaxY(0,50,2)
pl.plot(v1)
# Put additional text at the same coordinates on zone 2
pl.zoneText  (1,20,"Zone Text at (0,20)",2)
pl.reset ()
```

The output in Figure 8.12 shows how the text position changes according to zone limits.

Figure 8.12. Placing text in Zone coordinates



Using TextStyle to change text appeareance

The visual appeareance of a piece of text (e.g. font, color etc.) can be customized by using the `textStyle()` method of the plotter. The Table 8.9 summarizes the key/value pairs accepted by such method:

Table 8.9. Text style properties

Key	Value	Remark
color	blue,white,black,red,green,yellow,magenta,cyan,darkgray,lightgray,gray darkred,darkgreen,darkblue,darkcyan,darkmagenta,darkyellow	Color of the text
fontname	string	Name of the font family (default is Helvetica)
fontsize	integer	Height of the font in points (1/72 inch)
bold	yes no	Bold attribute
italic	yes no	Italic attribute

As a real example of setting text attribute, the following snippet of Python code shows how to declare a function that sets all text attributes in one go:

```
# A Python function to set text attributes
def textProp(name,size,bold,italic,color="black") :
   pl.textStyle ("fontname",name)
   pl.textStyle ("fontsize",size)
   pl.textStyle ("bold",bold)
   pl.textStyle ("italic",italic)
   pl.textStyle ("color",color)
```

Such function could be used, e.g. to set the current font to a 18 points Courier with bold and italic attributes:

```
textProp("Courier","18","yes","yes")
```

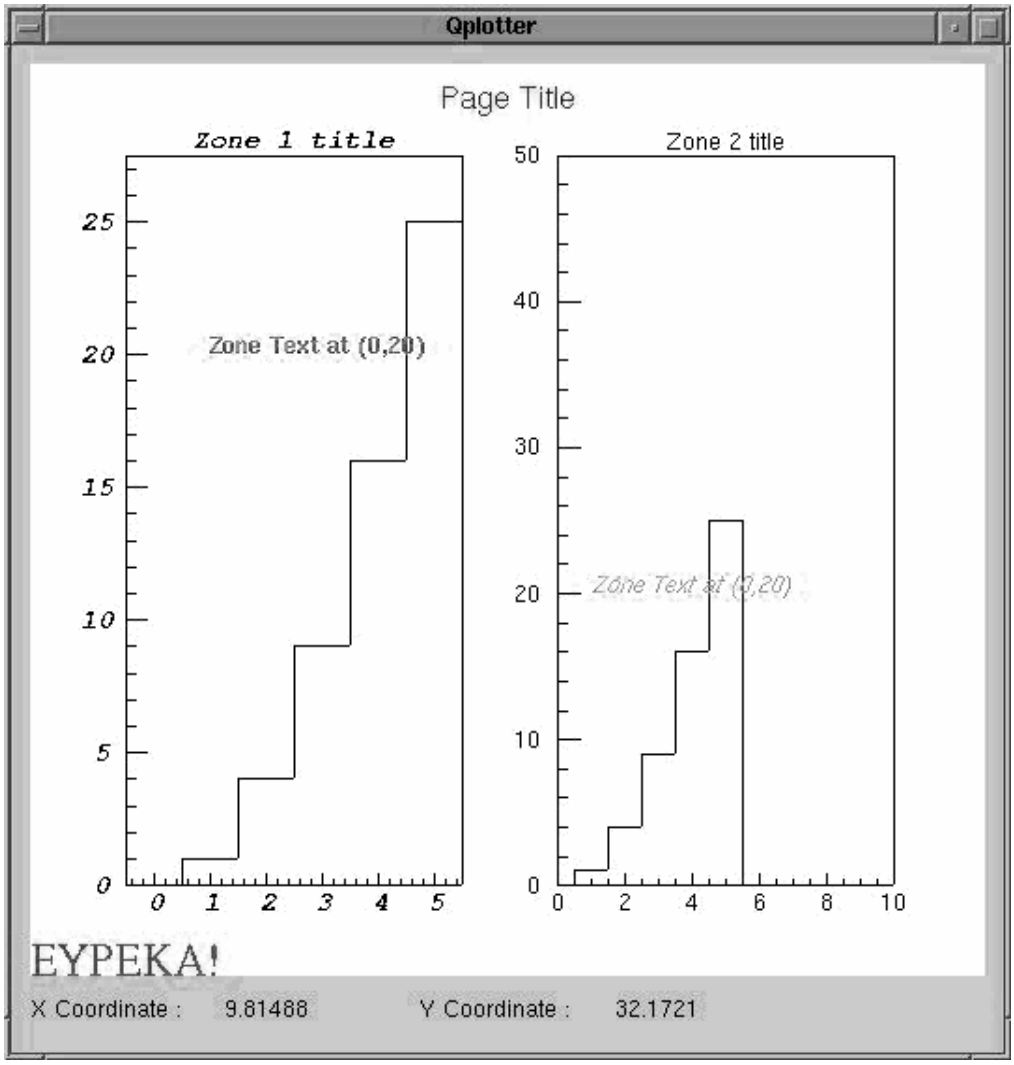
Text attributes can be defined independently for each piece of text in the page, as in this ‘wrap-up’ script:

```
# A Python function to set text attributes
def textProp(name,size,bold,italic,color="black") :
   pl.textStyle ("fontname",name)
   pl.textStyle ("fontsize",size)
   pl.textStyle ("bold",bold)
   pl.textStyle ("italic",italic)
   pl.textStyle ("color",color)


# These are Python lists
# A list from 0 to 5
xvals = [x*1. for x in range(0.,6.)]
# A list with values squared
yvals = [x*x for x in xvals]
# ’Error’ on X i.e. half binwidth
exvals = [0.5 for x in xvals]
# Error on Y, i.e. sqrt(Y)
eyvals = [sqrt(y) for y in yvals]

# Create Lizard vector
```

```
v1=vm.fromPy(xvals,yvals,exvals,eyvals)

# Zone settings
pl.zone(2,1)
pl.zoneOption ("option","nostats")

# Zone 1 global font : Courier 14 points, bold, italic
textProp("Courier","14","yes","yes")
pl.plot(v1)
pl.zoneTitle("Zone 1 title",1)
# Zone 1 font for additional text : Helvetica 12 points, bold,  blue
textProp("Helvetica","12","yes","no","blue")
# Put additional text one zone 1
pl.zoneText  (1,20,"Zone Text at (0,20)",1)

# Reset all properties
# Change min/max to show impact on Zone coordinates
pl.reset ()
pl.setMinMaxX(0,10,2)
pl.setMinMaxY(0,50,2)
pl.plot(v1)
# Zone 2 font for additional text : Helvetica 12 points, italic,  magenta
textProp("Helvetica","12","no","yes","magenta")
# Put additional text at the same coordinates on zone 2
pl.zoneText  (1,20,"Zone Text at (0,20)",2)
pl.reset ()
pl.zoneTitle("Zone 2 title",2)

# Some text in Page coordinate system using Symbol font
textProp("symbol","24","yes","no","red")
pl.pageText  (0,0,"EUREKA!")
# Last setting
textProp("Helvetica","16","no","no","darkgreen")
pl.pageTitle("Page Title")

pl.reset()
```

The output in Figure 8.13 is similar to what done before, but now each piece of text has its own attributes:

Figure 8.13. Changing text properties



Mathematical formulas and special symbols

Lizard allows users to show mathematical formulas or text including math symbols wherever plain text is accepted. The markup of the formulas is based on a subset of MathML, the Mathematical Markup language endorsed by the W3C Consortium (see [*W3CMathHome*] for details). Apart from using a markup syntax in the text, there's no difference on the methods' signature, e.g.:

```
# A legend with the methane chemical formula using subscript
pl.dataOption ("legend","<qpmath><msub><mi>CH</mi><mn>4</mn></msub></qpmath>")
# A legend with the methane chemical formula whithout subscript
pl.dataOption ("legend","CH4")
```

Notice how the MathML text is identified by the pair of tags `<qpmath>` and `</qpmath>`. If no such tag is used, the content is rendered as plain text.

A quick introduction to MathML

Taken from the W3C Math Home Page. ' MathML is intended to facilitate the use and re-use of mathematical and scientific content on the Web, and for other applications such as computer

algebra systems, print typesetting, and voice synthesis. MathML can be used to encode both the presentation of mathematical notation for high-quality visual display, and mathematical content, for applications where the semantics plays more of a key role such as scientific software or voice synthesis. MathML is cast as an application of XML. As such, with adequate style sheet support, it will ultimately be possible for browsers to natively render mathematical expressions. For the immediate future, several vendors offer applets and plug-ins which can render MathML in place in a browser. Translators and equation editors which can generate HTML pages where the math expressions are represented directly in MathML will be available soon. ’

On first sight, MathML looks much like XHTML, i.e. :

```
<mrow>
        <mrow>
                <msup> <mi>x</mi> <mn>2</mn> </msup> <mo>+</mo>
                <mrow>
                        <mn>4</mn>
                        <mo>&InvisibleTimes;</mo>
                        <mi>x</mi>
                </mrow>
                <mo>+</mo>
                <mn>4</mn>
        </mrow>
        <mo>=</mo>
        <mn>0</mn>
</mrow>
```

but the set of tags is of course targeting mathematical formula representation (the tags in the previous example belongs to the so-called ‘presentation markup’. An equivalent ‘content markup’ is supported as well, but this goes far beyond the scope of this document). Notice how elements are properly nested (start tag followed by end tag), according to XML requirements of well-formedness. The set of elements (tags) supported in Lizard is summarized in the next table:

Table 8.10. MathML elements

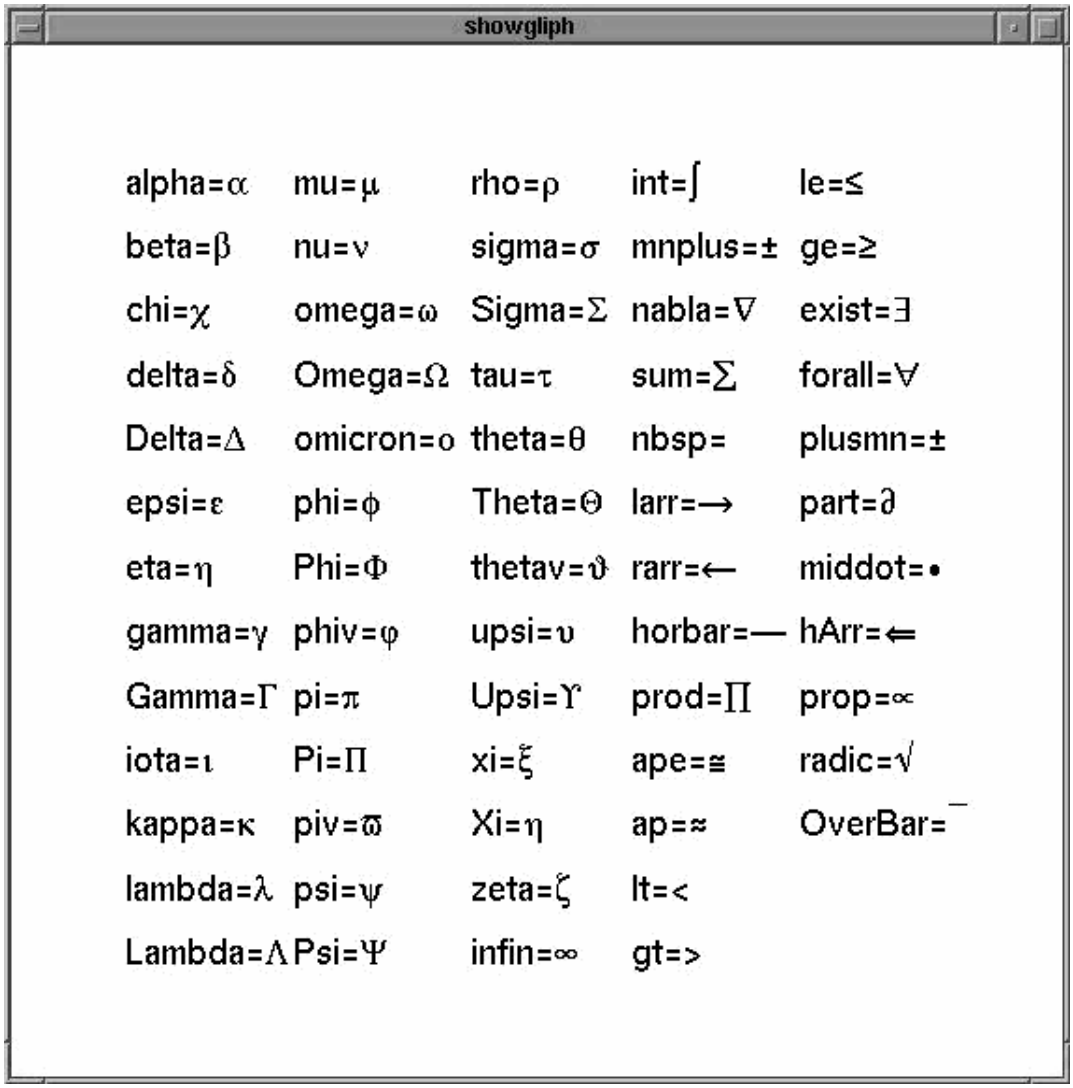
Name	Meaning
<mi>	identifier
<mn>	number
<mo>	operator, fence, or separator
<mtext>	text
<ms>	string literal
<mrow>	group any number of subexpressions horizontally
<msqrt>	form a square root sign (radical without an index)
<msub>	attach a subscript to a base
<msup>	attach a superscript to a base
<msubsup>	attach a subscript-superscript pair to a base
<munder>	attach an underscript to a base
<mover>	attach an overscript to a base
<munderover>	attach a underscript-overscript pair to a base

Symbols (including greek letters) are marked-up as `entities` i.e. identifiers enclosed between `&` and `;`, such as `&alpha;`. Although MathML allows to mix symbols in the text, Lizard requires that symbols (including greek letters) should be in an element of their own, e.g.:

```
# Correct
pl.pageText (30,160,"<qpmath><mi>&gamma;</mi><ms> rays shower</ms></qpmath>")
# Wrong
pl.pageText (30,150,"<qpmath><ms>&gamma; rays shower</ms></qpmath>")
```

This is a limitation that will certainly be overcome in future releases. A complete lists of all entities supported is shown in Figure 8.14

Figure 8.14. Entities available in Lizard



Examples of MathML use in Lizard

As stated beforehand, text marked-up with MathML can be used wherever plain text is accepted (e.g. titles, labels, arbitrary text, legends). In these examples the MathML text will be placed in the page coordinate system using the `pageText()` method of the plotter.

Showing a single Greek letter

The Greek letter represented by an entity having the same name. Greek letters in capital are entities having the same name with the first letter capitalized (i.e. `&delta;` vs. `&Delta;`). The symbol entity is enclosed in an `identifier` element (`<mi>`), then surrounded by the element that identifies the text as MathML (`<qpmath>`):

```
# Greek letter
pl.pageText(50,180,"<qpmath><mi>&alpha;</mi></qpmath>")
```

Using subscript, superscript

There are three elements that can be used: `<msub>`, `<msup>` and `<msubsup>`, as in the following code:

```
# Superscript: alpha squared
pl.pageText(50,170,"<qpmath><msup><mi>&alpha;</mi><mn>2</mn></msup></qpmath>")
# Subscript: oxygen
pl.pageText(50,160,"<qpmath><msub><mi>O</mi> <mn>2</mn> </msub></qpmath>")
# Subscript: Kronecker's symbol
pl.pageText(50,150,"<qpmath><msub><mi>&delta;</mi><mi>ij</mi></msub></qpmath>")
# Subscript/superscript: the Riemann's tensor
pl.pageText(50,135,"<qpmath><msubsup><mi>R</mi><mi>mrn</mi><mi>l</mi></msubsup><
```

The first element is always the base. In the case of `<msub>` and `<msup>`, the second element is the subscript/superscript. For `<msubsup>`, the second element is the subscript, the third element is the superscript.

Using underscript, overscript

There are three elements that can be used: `<munder>`, `<mover>` and `<munderover>`, as in the following Python code:

```
# Underscript
pl.pageText(50,125,"<qpmath><munder><mi>&Sigma;</mi><ms>i=0</ms></munder></qpmat
# Overscript
pl.pageText(50,105,"<qpmath><mover><mi>&Sigma;</mi><mi>&infin;</mi></mover></qpm
# All together now
text="<qpmath><munderover><mi>&Sigma;</mi><ms>i=0</ms><mi>&infin;</mi>"
text=text+"</munderover></qpmath>"
pl.pageText(50,90,text)
```

The first element is always the base. In the case of `<munder>` and `<mover>`, the second element is the underscript/overscript. For `<munderover>`, the second element is the underscript, the third element is the overscript.

Square root

The `<msqrt>` allows the user to show the square root of an expression, as in this code:

```
# Square root: notice the use of <mrow> to "glue" the sum of two terms
# Square root: notice the use of <mrow> to "glue" the sum of two terms
text="<qpmath><msqrt><mrow><msup><mi>a</mi> <mn>2</mn> </msup>"
text=text+"<mo>+</mo><msup><mi>b</mi> <mn>2</mn> </msup></mrow></msqrt></qpmath>
pl.pageText(50,75,text)
```

The <msqrt> element accept another element, the expression the operator applies to. In this example the expression consists of the sum of two terms, so they have to be glued together using the <mrow> element.

Integral

Although there's no special element for an integral, it is possible to typeset one using other MathML elements. The Lizard implementation is quite primitive, since the integral operator is not "stretchy", i.e. does not adjust its size according to the argument:

```
# An integral
text="<qpmath><mrow> <munderover><mi>&int;</mi><mrow><mo>-</mo>"
text=text+"<mi>&infin;</mi></mrow><mrow><mi> </mi><mi> </mi><mi>&infin
text=text+"</munderover><mrow><mi> </mi><msup><mi>e</mi><mrow><mo>-</mo><mi
text=text+"</msup><mi> dx</mi></mrow></mrow></qpmath>"
pl.pageText(50,60,text)
```

The well-known PAW example

As a final proof on how tricky markup could be, here's the MathML text to show a well known example taken from the PAW manual:

```
# The famous PAW example:
text="<qpmath><mrow> <msub><mi>L</mi> <mi>em</mi></msub><mo>=</mo>"
text=text+"<mi> </mi> <mi>e</mi><mi> </mi> <msubsup><mi>J</mi><mi>em</
text=text+" <mi>&mu;</mi></msubsup><msub><mi>A</mi> <mi>&mu;</mi></msub><mi>&nbs
text=text+"<mi>,</mi><mi> </mi><msubsup><mi>J</mi><mi>em</mi> <mi>&mu;</mi>
text=text+"<mo>=</mo> <mover><mi>l</mi><mo>_</mo></mover><mi> </mi><msub><m
text=text+"<mi>l</mi><mi> </mi><mi>,</mi><mi> </mi><msubsup><mi>M</mi>
text=text+"<mo>=</mo> <munderover><mi>&sum;</mi> <mrow><mi> </mi><mi>&alpha
text=text+"<mrow><mi> </mi><mi>&infin;</mi></mrow></munderover><msub><mi>A<
text=text+"<msubsup><mi>&tau;</mi><mrow><mi>&alpha;</mi><mi>j</mi></mrow> <mi>i<
pl.pageText(50,45,text)
```

The whole example

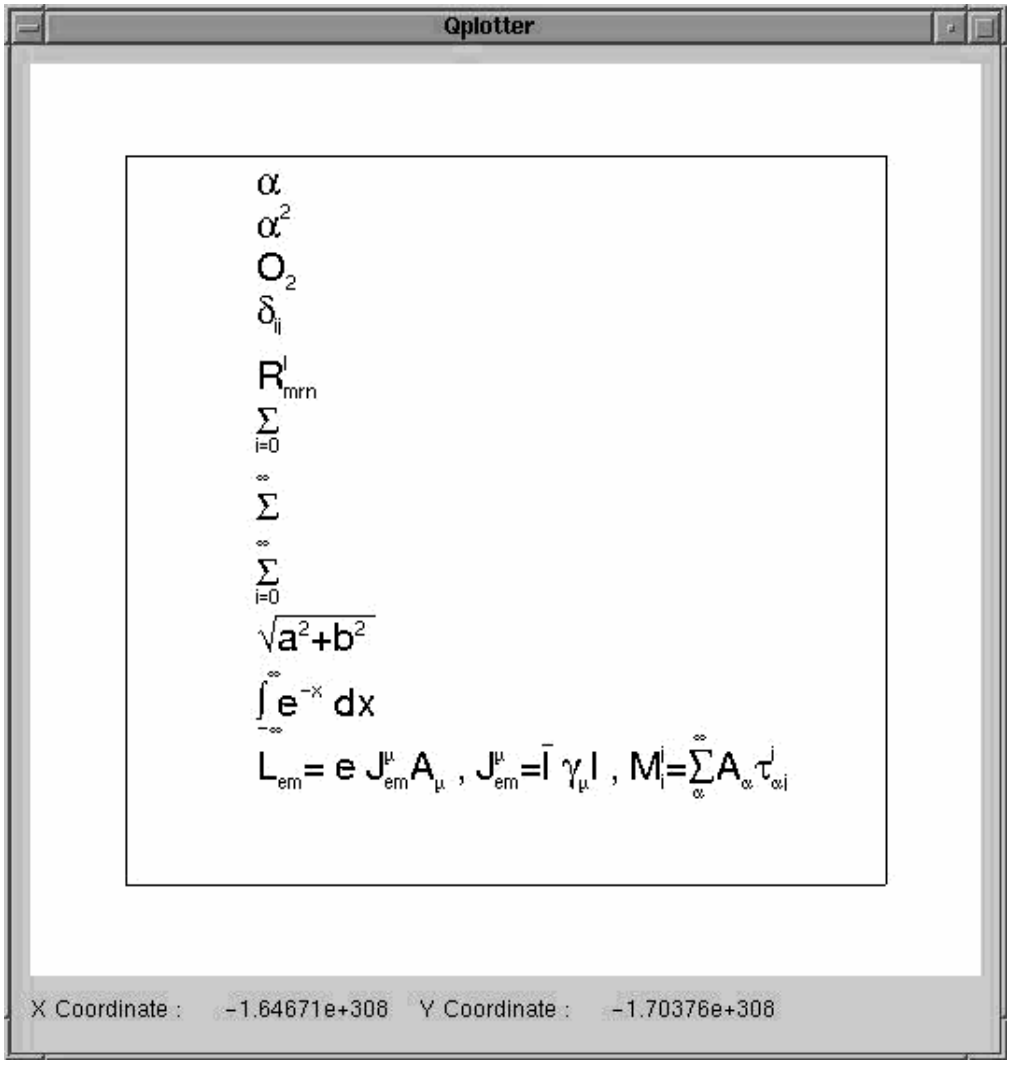
Wrapping up all these examples in a single Pyhton script would look like that:

```
pl.zone(1,1)
pl.textStyle ("fontsize","20")
# Greek letter
pl.pageText(50,180,"<qpmath><mi>&alpha;</mi></qpmath>")
# Superscript: alpha squared
pl.pageText(50,170,"<qpmath><msup><mi>&alpha;</mi><mn>2</mn></msup></qpmath>")
# Subscript: oxygen
pl.pageText(50,160,"<qpmath><msub><mi>O</mi> <mn>2</mn> </msub></qpmath>")
# Subscript: Kronecker's symbol
pl.pageText(50,150,"<qpmath><msub><mi>&delta;</mi><mi>ij</mi></msub></qpmath>")
```

```
# Subscript/superscript: the Riemann’s tensor
pl.pageText(50,135,"<qpmath><msubsup><mi>R</mi><mi>mrn</mi><mi>l</mi></msubsup><
# Underscript
pl.pageText(50,125,"<qpmath><munder><mi>&Sigma;</mi><ms>i=0</ms></munder></qpmat
# Overscript
pl.pageText(50,105,"<qpmath><mover><mi>&Sigma;</mi><mi>&infin;</mi></mover></qpm
# All together now
pl.pageText(50,90,"<qpmath><munderover><mi>&Sigma;</mi><ms>i=0</ms><mi>&infin;</
# Square root: notice the use of <mrow> to "glue" the sum of two terms
text="<qpmath><msqrt><mrow><msup><mi>a</mi> <mn>2</mn> </msup>"
text=text+"<mo>+</mo><msup><mi>b</mi> <mn>2</mn> </msup></mrow></msqrt></qpmath>
pl.pageText(50,75,text)
# An integral
text="<qpmath><mrow> <munderover><mi>&int;</mi><mrow><mo>-</mo>"
text=text+"<mi>&infin;</mi></mrow><mrow><mi> </mi><mi> </mi><mi>&infin
text=text+"</munderover><mrow><mi> </mi><msup><mi>e</mi><mrow><mo>-</mo><mi
text=text+"</msup><mi> dx</mi></mrow></mrow></qpmath>"
pl.pageText(50,60,text)
# The famous PAW example:
text="<qpmath><mrow> <msub><mi>L</mi> <mi>em</mi></msub><mo>=</mo>"
text=text+"<mi> </mi> <mi>e</mi><mi> </mi> <msubsup><mi>J</mi><mi>em</
text=text+"<msub><mi>A</mi> <mi>&mu;</mi></msub><mi> </mi><mi>,</mi><mi>&nb
text=text+"<msubsup><mi>J</mi><mi>em</mi> <mi>&mu;</mi></msubsup> <mo>=</mo>"
text=text+"<mover><mi>l</mi><mo>_</mo></mover><mi> </mi><msub><mi>&gamma;</
text=text+"<mi>l</mi><mi> </mi><mi>,</mi><mi> </mi><msubsup><mi>M</mi>
text=text+"<mo>=</mo> <munderover><mi>&sum;</mi> <mrow><mi> </mi><mi>&alpha
text=text+"<mrow><mi> </mi><mi>&infin;</mi></mrow></munderover><msub><mi>A<
text=text+"<msubsup><mi>&tau;</mi><mrow><mi>&alpha;</mi><mi>j</mi></mrow> <mi>i<
pl.pageText(50,45,text)
```

The output of such script can be seen in Figure 8.15

Figure 8.15. Examples of MathML in Lizard



Chapter 9. Using the `Analyzer` component

Table of Contents

Key concepts of using the `Analyzer` for fitting

Introduction

The `Analyzer` component allows Lizard users to execute "external" C++ code compiled as a shared library. Such a feature would allow, e.g., to run experiment-specific code (e.g. simulation or reconstruction) straight from Python. Since the coupling between Lizard and the user code is very weak, the `Analyzer` does not impose any contraint to the external code in terms e.g. of inheritance from common objects etc.

Use of shared libraries

Compiled code can be grouped in libraries. Static libraries contain code that is linked directly to the executable file. Shared libraries on the other hand keep the code outside the executable program: at loading time the Operating System loads the shared library in memory and bind its adresses to the application code.

A shared library can also be ‘loaded on-demand’, i.e. the executable explicitly loads and bind the code using the Operating System API: this is the method used by Lizard to load and execute user code. While loading a shared library is trivial, to execute the user code is a bit more tricky, since the program needs an ‘entry point’ to start from (this phase is called ‘symbol lookup’ in the jargon). Lizard uses a very simple approach to solve this program:

- the user code *must* contain a function named `doIt`.
- such function *must* have C-linkage (to avoid problems with C++ symbol’s mangling).

The following C++ code shows this in practice:

```
extern "C"
{
  void* doIt( IHistoManager* hm, INtupleManager* ntm, IVectorManager* vm )
  {
    std::cout << "Hello world!" << std::endl;
  }
}
```

If such function exists with the proper C-linkage, the `Analyzer` will just load the library, look the function up and execute it. Since there’s no limit of what the user puts in the body of the function, virtually any computation can be carries out in this way.

Building a shared library

Every compiler has its own way of building shared libraries. For instance in order to make a shared library using g++ the following Python command could be used:

```
# Ask the compiler to create a shared library
shell("g++ -fpic -shared -ldl -rdynamic myCode.cpp -o myCode.so")
```

This way of working becomes cumbersome as soon as more compiler flags are needed, so people ususally rely on more powerful tools such as gmake or full-fledged configuration managers (a.k.a.

software release tools). In the following of this chapter, examples will rely on the use of gmake, but via the Python's `shell` function, any other software building tool could be used instead.

Interaction between Lizard and the user code

Loading and executing user code from a shared library is straightforward, but it's useless unless the code can 'communicate' with Lizard. The way the two entities communicate is usually defined as a 'protocol'. Common sense would require that such protocol is complete enough not to constrain the users' capabilities, yet as simple as possible to enhance ease-of-use.

The approach taken in Lizard is to tailor the protocol around the typical use-cases physicists face in their day-to-day analysis. The basic idea is that the user code shares the set of Lizard `Managers` (see the section called " Components in Lizard" for a more precise definition). In this way it's possible to implement a sort of 'round-trip analysis' such as:

Procedure 9.0. round-trip analysis

1. Load or create the input data in Lizard, e.g.
    a. load histograms from database
    b. locate ntuples in database
    c. create vectors of unbinned data
2. Execute user code
    a. access input data via `Managers`
    b. create output data via `Managers`
3. Examine output data in Lizard

The 'protocol' is reflected in the signature of the `doIt` function which takes three arguments, respectively the `HistogramManager`, the `NtupleManager` and the `VectorManager` pointers.

```
extern "C"
{
  void* doIt( IHistoManager* hm, INtupleManager* ntm, IVectorManager* vm )
  {
    // EMPTY
  }
}
```

Making a shared library visible to programs

The most flexible way to use shared libraries is to rely on the operating system capability to look them up using a well defined environment variable. On Linux and Solaris , such variable is named `LD_LIBRARY_PATH`, thus to be able to load shared libraries in Lizard the user has update that variable *before* starting Lizard. For instance in order to add the current directory to the path to search libraries for, use the following Unix commands (respectively for csh flavor and Bourne

flavor shells):

`setenv LD_LIBRARY_PATH $PWD:${LD_LIBRARY_PATH}` or

`export LD_LIBRARY_PATH=‘pwd‘:$LD_LIBRARY_PATH`

Some `Analyzer` examples

The simplest example

This example shows how to write, compile and execute some very simple user code that computes the factorial of a number. The minimal C++ code required would be something like this:

```
#include <Interfaces/IHistoManager.h>
#include <Interfaces/INtupleManager.h>
#include <Interfaces/IVectorManager.h>

extern "C"
{
  void* doIt( IHistoManager* hm, INtupleManager* ntm, IVectorManager* vm )
  {
    int result=1,i;
    for (i=1;i<6;i++)
      result=result*i;
    std::cout << "Factorial of 6 is " << result << std::endl;
  }
}
```

To compile such code on Linux (assuming it’s saved in a file named `myAnalyzer0.cpp`, the compiler is explicitly called via the `shell` Python’s function:

```
:-) shell("g++ -I$LHCXX_REL_DIR/include -fpic -shared -ldl -rdynamic myAnalyzer0
0
```

(the `0` return value means the compilation was successful). To execute the user code it’s necessary to create an `Analyzer` instance and then call it’s `analyze` method:

```
:-) an=Analyzer()
:-) an.analyze ("myAnalyzer0.so",hm,ntm,vm)
Factorial of 6 is 120
:-)
```

Notice how the `analyze` method requires the name of the shared library and the list of the Lizard managers.

Structuring user code and compiling via gmake

To write the whole user code in a single ‘C function’ not the most appropriate way to structure it. Our suggestion is to keep the ‘entry-point’ function as simple as possible and to define a new class for the analyzer. One way to define such class which is very effective is to give it three main methods: one that carries out the bulk of the computation, one which is executed before starting the computation and one that is executed once the computation is over. This fits pretty well with the standard to-do list of a physicist doing analysis:

Procedure 9.1. Typical analysis program

1. Book histograms, create vectors or open ntuples
2. Perform the analysis
3. Clean up the output data

An example of such class could be something like that:

```
#ifndef INCLUDED_MYANALYZER1_H
#define INCLUDED_MYANALYZER1_H

class myAnalyzer
{
public:
  // Construct/destruct</para>
<para>

  myAnalyzer            ();
  ~myAnalyzer           ();
  // Methods
  bool     preExecute   ();
  bool     postExecute  ();
  void     doIt         ();
};

#endif // #ifndef INCLUDED_MYANALYZER1_H
```

This code is placed in a header file, then included in the user code. Assuming the file is named `myAnalyzer1.h`, the corresponding code in `myAnalyzer1.cpp` could be something like this:

```
#include "myAnalyzer1.h"
#include <Interfaces/IHistoManager.h>
#include <Interfaces/INtupleManager.h>
#include <Interfaces/IVectorManager.h>

extern "C"
{
  void* doIt( IHistoManager* hm, INtupleManager* ntm, IVectorManager* vm )
  {
    // Creating an analyzer instance
    myAnalyzer a ;
    // Call the pre-execution method
    a.preExecute();
    // Do computation
    a.doIt();
    // Call the post-execution method
    a.postExecute();
  }
}

myAnalyzer::myAnalyzer() {
}

myAnalyzer::~myAnalyzer() {
}

bool myAnalyzer::preExecute( ) {
```

```
}

bool myAnalyzer::postExecute( ) {
}

void myAnalyzer::doIt() {
    int result=1,i;
    for (i=1;i<6;i++)
      result=result*i;
    std::cout << "Factorial of 6 is " << result << std::endl;
}
```

Notice how the `doIt` function becomes extremely general and somehow independent from the analysis code, which would rather be placed in the `myAnalyzer` class. To compile the code this time we'll use gmake.

The gmake requires a 'makefile' containing the directives to create the shared library. If no file is explicitly specified, gmake will look for a default makefile named `GNUmakefile`. This is an example of such makefile:

```
# Linux specific!
CXX        = g++
LDSHARED   = $(CXX) -shared -ldl -rdynamic
CCSHARED   = -fpic

# User code filename: default is myAnalyzer.cpp
ifndef PROG
PROG  = myAnalyzer.cpp
endif

HDRS       = ${PROG:.cpp=.h}
OBJS       = ${PROG:.cpp=.o}
SHR_OBJS   = ${PROG:.cpp=.so}

INCLUDE   += -I${LHCXX_REL_DIR}/include

.SUFFIXES: .cpp .h
.PHONY: all

all: ${SHR_OBJS}

$(HDRS) :

${SHR_OBJS} : ${OBJS}
        $(LDSHARED) ${OBJS} -o ${SHR_OBJS}

${OBJS} : $(PROG) $(HDRS)
        @echo ++++++++++ compiling $<
        $(CXX) $(CCSHARED) -w -pipe -c $< $(INCLUDE) -o $@
```

The discussion of makefiles' structure is beyond the scope of this document: in the Lizard `examples` directory users will find an appropriate makefile to run this examples. Lizard defines a `make()` shortcut that is equivalent to **`shell("gmake")`**. The shortcut accepts a string argument which is passed over to gmake (e.g. to specify a target or the value of a variable). Now we can execute the same code using this Python script:

```
# Create analyzer
```

```
an=Analyzer()
# Compile the code
make("PROG=myAnalyzer1.cpp")
# Execute it
an.analyze ("myAnalyzer1.so",hm,ntm,vm)
# Clean-up
del an
```

Interacting with the `HistoManager` component

The first step in making user code interact with Lizard is to understand how to use the `HistoManager` component from within the C++ code.

As a first example lets' assume the task is to book and fill a histogram in the C++ code and then see how it looks like from Lizard. With respect to the code used in the previous section, the changes are:

1. Declare an histogram pointer as private member of the `myAnalizer` class (so that it can be accessed by the class methods)
2. Modify the `preExecute` method to book the histogram
3. Modify the `doIt` method to fill it

The class declaration becomes:

```
#ifndef INCLUDED_MYANALYZER2_H
#define INCLUDED_MYANALYZER2_H

class myAnalyzer
{
public:
  // Construct/destruct/copy
  myAnalyzer            ();
  virtual  ~myAnalyzer  ();
  // Methods
  // preExecute() takes the HistoManager handle as parameter
  bool     preExecute   ( IHistoManager* hm);
  bool     postExecute  ();
  void     doIt         ();
private:
  // A pointer to a histogram
  IHistogram1D* h1;
};

#endif // #ifndef INCLUDED_MYANALYZER2_H
```

while corresponding implementation is:

```
#include <stdlib.h>
#include <Interfaces/IHistoManager.h>
#include <Interfaces/IHistogram1D.h>
#include <Interfaces/INtupleManager.h>
#include <Interfaces/IVectorManager.h>
#include "myAnalyzer2.h"

extern "C"
{
```

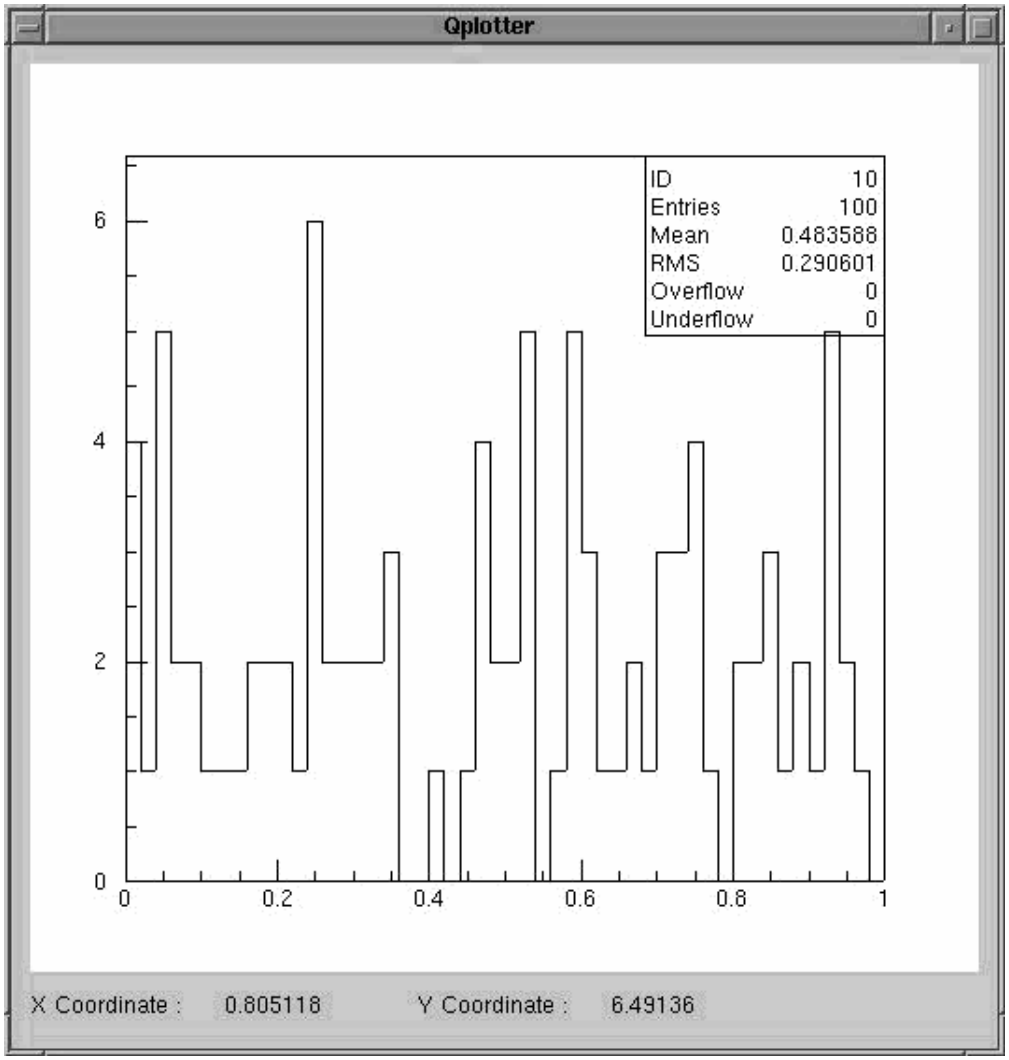
```
  void* doIt( IHistoManager* hm, INtupleManager* ntm, IVectorManager* vm )
  {
    // Creating an analyzer instance
    myAnalyzer a ;
    // Call the pre-execution method
    if (a.preExecute(hm)) {
      // Do computation
      a.doIt();
      // Call the post-execution method
      a.postExecute();
    }
  }
}

myAnalyzer::myAnalyzer() : h1 (0) {
}

myAnalyzer::~myAnalyzer() {
}

// book the histogram before using it
bool myAnalyzer::preExecute( IHistoManager* hm ) {
  h1 = hm->create1D( "10", "random", 50, 0., 1. );
  return h1 != 0;
}

bool myAnalyzer::postExecute( ) {
}

// Fill the histogram with random values from [0,1] uniform distribution
void myAnalyzer::doIt() {
  int i;
  for (i=0;i<100;i++) {
    double val = (static_cast<double>(rand()))/RAND_MAX;
    h1->fill(val);
  }
}
```

Now a small Python script to execute the code and show the resulting histogram

```
# Create an analyzer
an=Analyzer()
# Compile the user code
rc = make("PROG=myAnalyzer2.cpp")
if (rc == 0) :
 # Execute user code
 an.analyze ("myAnalyzer2.so",hm,ntm,vm)
 # Retrieve histogram handle using the ID
 h10=hm.retrieveHisto1D(10)
 # Plot histogram using shortcut
 v1=hplot(h10)

# Delete analyzer
del an
```

Notice how the histogram is retrieved in Lizard using the `retrieveHisto1D()` method of the `HistoManager`. The resulting output will be something like Figure 9.14

Figure 9.14. Histogram created in C++ and visualized in Lizard



Another way to obtain the same result would be to create the histogram in Lizard and then retrieve it in the `preExecute` method. The Python script must be modified accordingly:

```
# Create an analyzer
an=Analyzer()
# Compile the user code
rc = make("PROG=myAnalyzer2.cpp")
if (rc == 0) :
# Create the histogram so that can be retrieved later in C++
 h10=hm.create1D(10,"Random",50,0.,1.)
 # Execute user code
 an.analyze ("myAnalyzer2.so",hm,ntm,vm)
 # Plot histogram using shortcut
 v1=hplot(h10)

# Delete analyzer
del an
```

The impact on the C++ code is just the modification of the `preExecute` method so to retrieve an existing histogram rather than create it:

```
...
```

```
// book the histogram before using it
bool myAnalyzer::preExecute( IHistoManager* hm ) {
  h1 = hm->retrieveHisto1D( "10");
  return h1 != 0;
}
...
```

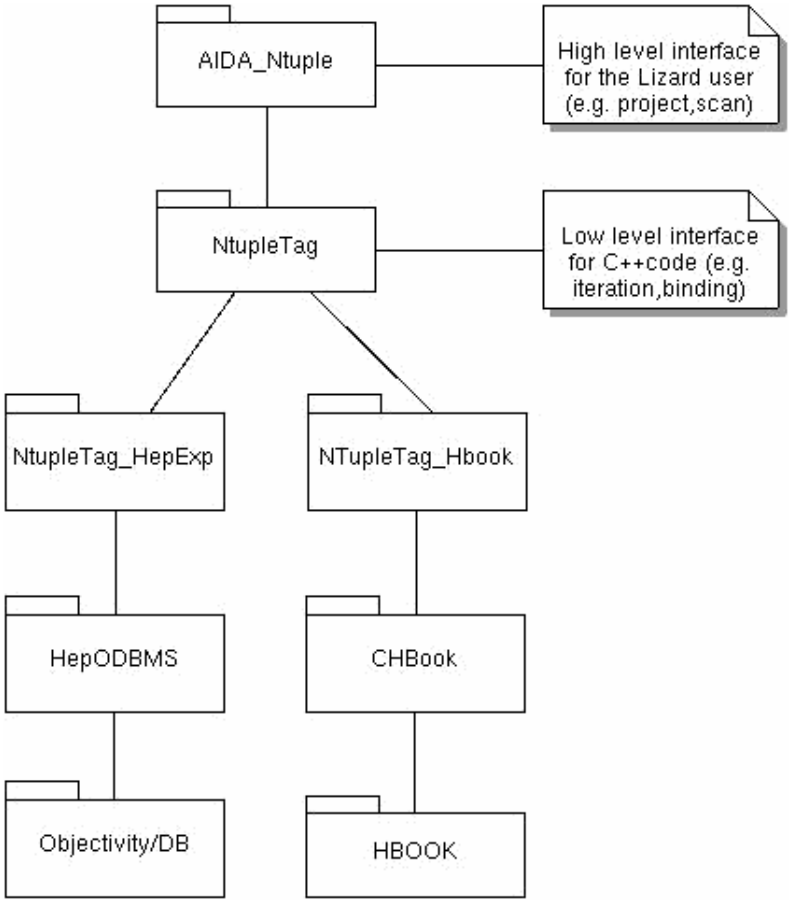
Ntuple analysis using the `Analyzer`

Introduction

As described in Chapter 6, Lizard provides methods to scan,plot and project ntuple attributes using C++ cuts and expressions. If more complex ntuple analysis must be carried out, all Interactive Analysis tools provide some kind of ‘gateway’ to execute user code written in the programming language of choice. For instance PAW allowed to execute Fortran programs (via COMIS) or to trigger the execution of a Fortran subroutine for each ntuple entry, as in `NT/LOOP`. The Lizard approach is to use the `Analyzer` for the same purpose: execute user code that goes through the ntuple, performs actions on the data and share the outcome (e.g. histograms) with the tool. The main difference with respect to the above-mentioned `NT/LOOP` solution is that the user code manages the so called ‘ntuple loop’, i.e. the iteration over the ntuple. This gives the user more flexibility at a modest cost, since the looping code is very simple and can be automatically generated on his behalf. The current version of Lizard does not support such automatic generation yet, but the additional code is so simple that cut-and-paste from examples is largely enough.

The ntuple component(s) of Lizard are structured as a ‘layered system’ so to avoid direct coupling between high level functionalities (as those described in Chapter 6) and low-level storage back-end (such as Objectivity/DB). The relation among those layers are depicted in Figure 9.15.

Figure 9.15. Relations among ntuple components



Key concepts of Lizard ntuple analysis in C++

The 'low-level' programming interface to ntuples is not implemented in terms of Abstract Interfaces (yet). This means the `Analyzer` code is less portable, although the Lizard implementation is quite general and supports multiple backends for storage (so far Objectivity/DB and HBOOK).

The four basic concepts of any ntuple analysis code in Lizard are:

- Ntuples are looked up via a *factory*, e.g.

```
// Finding ntuple
iNtuple = factory->findOneC( "TagCollection1" );
```

- Iteration is done via the `Ntuple` object, e.g.

```
for( iNtuple->begin(); !iNtuple->isEnd(); iNtuple->next() ) {
// User code
}
```

- Ntuple attributes are retrieved via `Quantity` objects. Quantities behave as standard C++ typed variables, e.g.

```
// Quantity to bind pT attribute in ntuple
```

```
Quantity<double> pt;
```

- Quantites are explicitly bound to ntuple variables, e.g.

```
iNtuple->bind( "pt", pt )
```

From then on, whenever the program acces the `Quantity` instance, its value reflects the value of the attribute in the current ntuple row, so it can be used as a cut expression or to fill an histogram:

```
if (pT > 5.0)
  h1->fill(pt);
```

An example of Lizard ntuple analysis in C++

The user code

The following example is based on the same ntuples used in Chapter 6. The code locates the ntuple named `TagCollection1`, binds two attributes (out of four) and project one of those attributes using the other one for the cut. The general structure of the code is very close to what was used in the section called " Interacting with the `HistoManager` component ": a header file with the class declaration and a source file containing the class implementation and the `Analyzer` "entry-point".

First the header file. The interesting part is certainly the declaration of `private` member variables:

```
#ifndef INCLUDED_MYANALYZER3_H
#define INCLUDED_MYANALYZER3_H

#include <NtupleTag/LizardNTupleFactory_HepExp.h>
#include "NtupleTag/LizardQuantity.h"

USE_LIZARD_NAMESPACE

// Forward declarations
class IHistoManager;
class INtupleManager;
class IVectorManager;
class IHistogram1D;

class myAnalyzer
{
public: // Interface
  // Construct/destruct
  myAnalyzer            ();
  virtual  ~myAnalyzer  ();
  // Methods
  bool     preExecute   ( IHistoManager* hm );
  void     doIt         ( INtupleManager * ntm );
private:
  bool      bind        ( NTuple* aNtp );
private:
  // Histogram pointer
  IHistogram1D *h1;
  // Analyzed nTuple
```

```
  NTuple* iNtuple;
  // Attributes to bind
  Quantity<double> pt;
  Quantity<double> phi;
};
#endif // #ifndef INCLUDED_MYANALYZER3_H
```

There we find a histogram pointer (initialized with an histogram in the `preExecute()` method), a pointer to an `NTuple` instance and two `Quantity` objects that will contain the ntuple's attributes `phi` and `pt`. Quantities are template classes having a basic C++ type as argument:

```
  // Quantity bound to a double attribute
  Quantity<double> pt;
  // Quantity bound to a float attribute
  Quantity<float>  px;
  // Quantity bound to a int attribute
  Quantity<int>  channel;
```

Lizard will take care of checking the consistency between the Quantity type and the type stored in the ntuple. From the user point of view Quantities behave exactly as the type in the argument, so this is perfectly legal:

```
// Print squared momentum
std::cout<<pt*pt<<std::endl;
```

Although the source code is listed in its entirety, the focus is mainly on two methods: `doIt` and `bind`:

```
#include "myAnalyzer3.h"
#include <Interfaces/IHistoManager.h>
#include <Interfaces/IHistogram1D.h>
#include <Interfaces/IHistogram2D.h>
#include <NtupleTag/LizardTransactionController.h>

void myAnalyzer::doIt( INtupleManager * ntm ) {
  // Creating a factory for HepExplorable
  //  NTupleFactory *factory = createNTupleFactory();
  NTupleFactory *factory =  new NTupleFactory_HepExp;
  // Starting Transaction
  factory->getTransCont().startRead();
  // Finding ntuple
  iNtuple = factory->findOneC( "TagCollection1" );
  if( iNtuple != 0) {
    // Check binding is successful
    if( bind( iNtuple )) {
      for( iNtuple->begin(); !iNtuple->isEnd(); iNtuple->next() ) {
        // A cut
        if (phi > 0)
          h1->fill(pt);
        std::cout<<pt*pt<<std::endl;
      }
    } else
      std::cout << "Bind error" << std::endl;
  }
  else
    std::cout << "Unable to open TagCollection1" << std::endl;
```

```
  // Committing Transaction
  factory->getTransCont().commit();
  delete factory;
}

bool myAnalyzer::bind( NTuple* aNtp ) {
  return aNtp->bind( "pt", pt ) && aNtp->bind( "phi", phi );
}

myAnalyzer::myAnalyzer() : iNtuple(0)  {
}

myAnalyzer::~myAnalyzer() {
  delete iNtuple;
}

bool myAnalyzer::preExecute( IHistoManager* hm ) {
  // Book histogram
  h1 = hm->create1D( "10", "pt", 50, 0., 50 );
  return h1 != 0;
}

extern "C"
{
  void* doIt( IHistoManager* hm, INtupleManager* ntm, IVectorManager* vm )
  {
    // Creating myAnalyzer
    myAnalyzer a ;
    // If pre execute successful, loop and fill
    if (a.preExecute( hm )) {
      a.doIt( ntm );
      a.postExecute();
    }
  }
}
```

Binding quantities to attributes

The `bind` method tries to bind the two quantities to their counterparts among the ntuple attributes. If all bindings are successful, the method returns a true value. The `bind` method on the ntuple accepts two argument, the name of the attribute (as a string) and a `Quantity`. While the name of the attribute must match the names stored in the ntuple, the name of the Quantity is entirely arbitrary (although using the same *word* certainly improves code readability).

```
// First argument is name in the ntuple, second is the related Quantity
iNtuple->bind( "pt", pt );
```

Iterating over an ntuple in the `doIt()` method

The `doIt` first looks up the ntuple via a factory:

```
  //  NTupleFactory *factory = createNTupleFactory();
  NTupleFactory *factory =  new NTupleFactory_HepExp;
  // Starting Transaction
  factory->getTransCont().startRead();
  // Finding ntuple
  iNtuple = factory->findOneC( "TagCollection1" );
```

If this step is successful, the code tries to bind the attributes to the class Quantities by invoking the private method `bind()` (see the section called "Binding quantities to attributes" for details).

```
// Check binding is successful
if( bind( iNtuple )) {
    ...
} else
  std::cout << "Bind error" << std::endl;
```

On successful binding, the code iterates over the ntuple using a `for` statement:

```
for( iNtuple->begin(); !iNtuple->isEnd(); iNtuple->next() ) {
  ...
}
```

Every cycle in the `for` loop brings in memory an ntuple row, i.e. the bound `Quantities` are updated (notice how the other attributes are just ignored, saving memory transfers). Using this quantities is now possible to apply a cut and fill the histogram previously booked:

```
// Cut on non-zero phi values
if (phi > 0)
  // Fill the pT histogram
  h1->fill(pt);
```

Transaction management

The code in the previous example contains calls to transaction management methods (`getTransCont().startRead()` and `getTransCont().commit()` respectively). Transaction management is common when working with databases (as in our example using Objectivity/DB). Indeed most databases wouldn't do any operation if no transaction is open. Since other (non-database based) back-ends may not have such concept, Lizard will provide empty transaction managers so that the code can easily be ported without changes.

Writing ntuples using the `Analyzer`

Key concepts

Although writing an ntuple is less frequent than reading it back, Lizard `Analyzer` can be used for this purpose as well. The concepts involved in ntuple creation can be summarized as follows:

- Ntuples are created via a *factory*, e.g.

```
// Create a HepExplorable-type nTuple factory
NTupleFactory_HepExp *factory = new NTupleFactory_HepExp;
// Create the nTuple via the factory and open for writing
NTuple* ntuple = factory->createC( aNTupleName );
```

- Ntuple attributes are created via `Quantity` objects. Quantities behave as standard C++ typed

variables, e.g.

```
// Quantity to bind pT attribute in ntuple
Quantity<double> pt;
// Declare the ntuple attribute and bind it to the quantity
ntuple->addAndBind( "phi", phi )
```

- Once the attributes for a row are properly initialized, a new row is inserted by calling the `NTuple::addRow()` method.

```
for( eventNo = 0; eventNo < 1000; eventNo++ ) {
  pt = 0;
  // Values of attributes are prepared; store them to the nTuple
  ntuple->addRow();
}
```

The user code

The example will create an ntuple having the same structure of those we saw so far (four attributes `eventNo` ,`pt` ,`phi` ,`Energy`), the data being just random samples drawn using CLHEP random generators. The header file is very simple, since no private member variables are allocated at all:

```
#ifndef INCLUDED_MYANALYZER4_H
#define INCLUDED_MYANALYZER4_H

//#include <NtupleTag/LizardNTupleFactory.h>
#include <NtupleTag/LizardNTupleFactory_HepExp.h>
#include "NtupleTag/LizardQuantity.h"

USE_LIZARD_NAMESPACE

// Forward declarations
class IHistoManager;

class myAnalyzer
{
public: // Interface
  // Construct/destruct
  myAnalyzer            ();
  virtual  ~myAnalyzer  ();
  // Methods
  void     doIt         ( INtupleManager * ntm );
};

#endif // #ifndef INCLUDED_MYANALYZER4_H
```

Apart from the implementation of the "entry-point", the C++ code is all grouped in the `doIt()` method:

```
#include <math.h>
#include <CLHEP/Random/Randomize.h>
#include <CLHEP/Units/PhysicalConstants.h>
#include "myAnalyzer3.h"
#include <Interfaces/IHistoManager.h>
```

```
#include <Interfaces/IHistogram1D.h>
#include <Interfaces/IHistogram2D.h>
#include <NtupleTag/LizardTransactionController.h>

extern "C"
{
  void* doIt( IHistoManager* hm, INtupleManager* ntm, IVectorManager* vm )
  {
    // Creating myAnalyzer
    myAnalyzer a ;
    a.doIt( ntm );
  }
}

myAnalyzer::myAnalyzer()  {
}

myAnalyzer::~myAnalyzer() {
}

void myAnalyzer::doIt( INtupleManager * ntm ) {
  string aNTupleName = "BrandNewTuple";
  // Create a HepExplorable-type nTuple factory
  NTupleFactory_HepExp *factory = new NTupleFactory_HepExp;
  // Start a write transaction
  factory->getTransCont().startUpdate();
  // Remove old nTuple if present
  if( factory->removeOne( aNTupleName ) )
    std::cout << "Old ntuple \"" << aNTupleName << "\" removed." << std::endl;
  // Create the nTuple via the factory and open for writing
  NTuple* ntuple = factory->createC( aNTupleName );
  // Check if successful
  if( ntuple != 0 ) {
    // Declare quantities which reflects attributes
    Quantity<long>    eventNo;
    Quantity<double>  pt;
    Quantity<double>  phi;
    Quantity<double>  Energy;
    // Add and bind new attributes
    if( ( ntuple->addAndBind( "eventNo", eventNo ) &&
          ntuple->addAndBind( "pt", pt ) &&
          ntuple->addAndBind( "phi", phi ) &&
          ntuple->addAndBind( "Energy", Energy ) ) ) {
      // Write 1000 rows
      for( eventNo = 0; eventNo < 1000; eventNo++ ) {
        pt = RandExponential::shoot( 1. / 0.17 );
        // Invert gaussian pseudo-rapidity
        double theta = 2. * atan( exp( -RandGauss::shoot( 3.0, 1.2 ) ) );
        // Flat phi distribution
        phi = RandFlat::shoot( twopi );
        // All particles make a cluster, no smearing
        Energy = sqrt( ( pt / sin( theta ) ) *
                       ( pt / sin( theta ) ) + ( .13956 ) * ( .13956 ) );
        // Values of attributes are prepared; store them to the nTuple
        ntuple->addRow();
      }
      std::cout << eventNo << " rows are generated successfully for ";
      std::cout << aNTupleName << std::endl;
    } else
      std::cout << "Error: unable to add attribute." << std::endl;
    delete ntuple;
  } else {
    std::cout << "Error: nTuple \"" << aNTupleName;
    std::cout << "\" cannot be created" << std::endl;
  }
```

```
  // commit changes
  factory->getTransCont().commit();
  delete factory;
}
```

After retriebving the *factory* the code starts an update transaction and tries to locate a ntuple with the same name. If so, it removes the old ntuple before creating the new one (this means the default behaviour is not to overwrite, to avoid losing data by mistake). The code then creates a new ntuple and declares local quantities that will store the values to put in the ntuple:

```
  // Create the nTuple via the factory and open for writing
  NTuple* ntuple = factory->createC( aNTupleName );
  // Declare quantities which reflects attributes
  Quantity<long>    eventNo;
  Quantity<double>  pt;
  Quantity<double>  phi;
  Quantity<double>  Energy;
```

The next step is to declare this new attributes in the ntuple and to bind them to the allocated quantities: this is done using the `NTuple::addAndBind()` method:

```
if ( ( ntuple->addAndBind( "eventNo", eventNo ) &&
   ntuple->addAndBind( "pt", pt ) &&
   ntuple->addAndBind( "phi", phi ) &&
   ntuple->addAndBind( "Energy", Energy ) ) )
```

Notice how the calls to `addAndBind()` are chained so that if one fails the test evaluates to `false`. AT this point an empty ntuple with all its attributes has been created, so it’s possible to fill it with rows of data. All quantities are assigned the proper value and then the call to `addRow()` make the current set of values a row in the ntuple.

```
      // Write 1000 rows
      for( eventNo = 0; eventNo < 1000; eventNo++ ) {
        pt = RandExponential::shoot( 1. / 0.17 );
        // Invert gaussian pseudo-rapidity
        double theta = 2. * atan( exp( -RandGauss::shoot( 3.0, 1.2 ) ) );
        // Flat phi distribution
        phi = RandFlat::shoot( twopi );
        // All particles make a cluster, no smearing
        Energy = sqrt( ( pt / sin( theta ) ) *
                       ( pt / sin( theta ) ) + ( .13956 ) * ( .13956 ) );
        // Values of attributes are prepared; store them to the nTuple
        ntuple->addRow();
      }
```

This is a small Python script which could be used to execute the code and check the new ntuple has been really created:

```
# Create an analyzer
an=Analyzer()
# Compile the user code
rc = make("PROG=myAnalyzer4.cpp")
if (rc == 0) :
```

```
 # Execute user code
 an.analyze ("myAnalyzer4.so",hm,ntm,vm)

# Delete analyzer
del an
# List ntuples
ntm.listNtuples ()
```

The output of the last command should show the new ntuple beside the old ones:

```
:-) ntm.listNtuples ()

Explorables present:

        TagCollection1
        TagCollection2
        TagCollection3
        TagCollection4
        BrandNewTuple
```

Fitting using the `Analyzer`

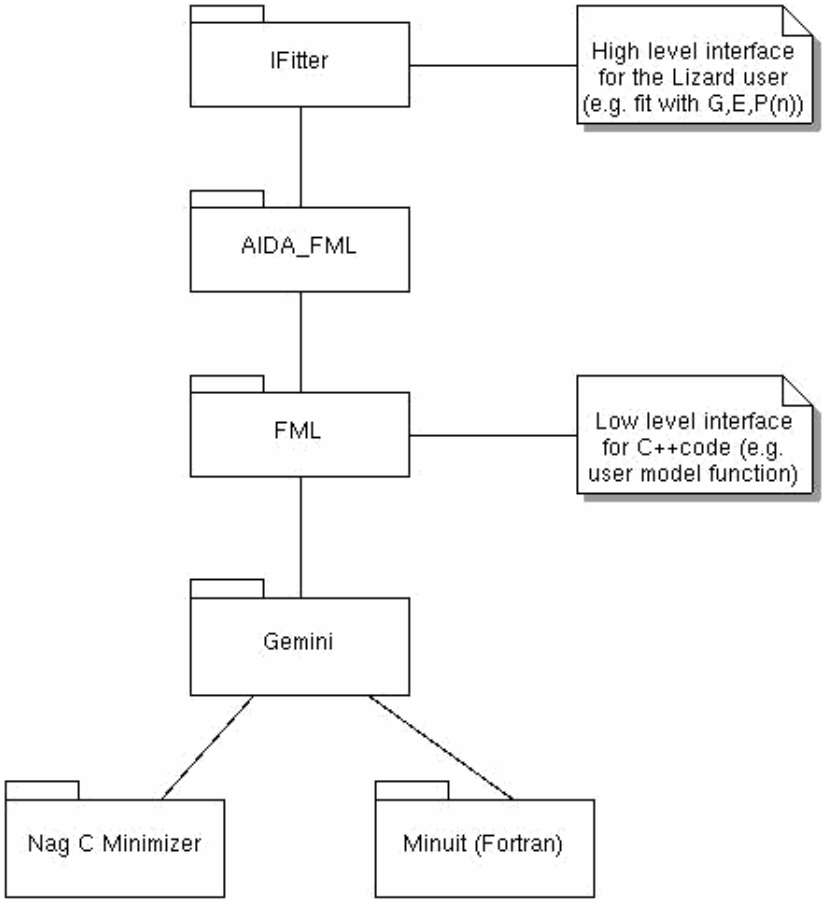
Introduction

The `Fitter` components allows users to compute best fit of data samples against distribution such as *Gaussian*, *Polynomial(N)*, *Exponential* (or additive combinations of those elementary models). Although this covers a large fraction of day-to-day analysis, Lizard allows users to fit data to an arbitrary function, via the `Analyzer` component. The rest of the section will provide more detailed information on how to do this.

Key concepts of using the `Analyzer` for fitting

The ‘low-level’ programming interface to fitting is (partially) defined in terms of Abstract Interfaces (although not yet AIDA ones). One implementation of such interfaces is the `FML` (Fitting and Minimization Library) package. FML in turns relies on Gemini, thus making possible to switch the minimization engine among different implementations (currently NAG C and Minuit). The relations among those packages are depicted in Figure 9.16.

Figure 9.16. Relations among fitting components



Bibliography and Useful Links

Books

[Gamma95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Copyright © 1995. 0-201-63361-2. Addison-Wesley Publishing Company. *Design Patterns: Elements of Reusable Object-Oriented Software*.

[Dalheimer99] Matthias Kalle Dalheimer. Copyright © 1999. 1-56592-588-2. O'Reilly. *Programming with Qt*. Writing Portable GUI applications on UNIX and Win32.

Web pages

[*Qt Home*] *Qt Home Page* .

[*Lizard Home*] *Lizard Home Page* .

[*W3CMathHome*] *W3C Math Home Page* .