Письма в ЭЧАЯ. 2005. Т. 2, № 4(127). С. 91–101

УДК 519.254

THE HISTOGRAMMING TOOL *hparse*

V. Nikulin [a,1], *G. Shabratova* [b,2]

[a] St. Petersburg Nuclear Physics Institute, Gatchina, Russia
[b] Joint Institute for Nuclear Research, Dubna

A general-purpose package aimed to simplify the histogramming in the data analysis is described. The proposed dedicated language for writing the histogramming scripts provides an effective and flexible tool for definition of a complicated histogram set. The script is more transparent and much easier to maintain than corresponding C++ code. In the TTree analysis it could be a good complement to the TTreeViewer class: the TTreeViewer is used for choice of the required histogram/cut set, while the *hparse* enables one to generate a code for systematic analysis.

Описывается программа общего назначения, предназначенная для упрощения процесса гистограммирования при анализе экспериментальных данных в рамках системы ROOT. Предлагаемый специализированный язык для написания сценариев гистограммирования предоставляет гибкий и эффективный инструмент для определения сложного набора гистограмм. Сценарий оказывается более прозрачным, чем соответствующие коды на C++; он позволяет проще модифицировать набор гистограмм. В случае анализа данных из TTree сценарий может стать хорошим дополнением к возможностям, предоставляемым классом TTreeViewer: графический интерфейс используется для выбора картинок, а *hparse* позволяет генерировать код для систематического анализа.

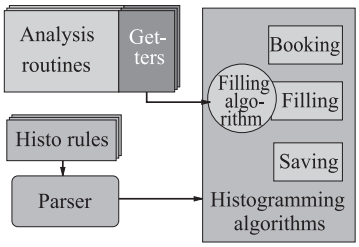
INTRODUCTION

The histogramming is rather time-consuming element of both data analysis and acquisition monitoring. Typically the booking of the histograms with meaningful names, filling it at right place with right parameters is a tedious procedure. Very often the histogramming-related actions are not localized, but distributed over the program (the *fill-as-soon-as-you-get-variable* approach). The algorithm modification becomes a nontrivial task for the user other than the author of the code.

The primary goal of the tool described here is to simplify the histogramming, to make the scenarios logically transparent. We propose to adopt the program architecture as illustrated in figure. The histogramming is independent of the analysis procedure, it is done in parallel: as soon as the event analysis is completed, the booked histogram filling according to the predefined algorithm starts (taking into account the cuts, weights, etc.). The only requirement to the analysis classes is availability of the access methods to all the information of eventual interest (by means of *Getters*). Normally, such kind of the class architecture is already implemented, and minor modifications are required.

We extensively exploit the fact that the histograms have a lot of common features: the similar detectors use similar electronic blocks, having similar data structure and similar variables of interest. The groups of histograms are filled within the same cut applied. This

[1] E-mail: nikulin@mail.cern.ch
[2] E-mail: gshabrat@sunhe.jinr.ru



Flow chart of the generic data analysis. The histogramming is separated from the analysis. The algorithm is defined in metafiles, the parser generates the C++ code for histogram booking, filling and storing

similarity allows one to define (quite limited amount of) generic variables (arguments) and cuts which could describe all the variety of possible combinations of variables.

The histogram title is generated automatically: it consists of the variable titles, optionally, the titles of the cuts applied, and weight. Thus we minimize possible errors of wrong histogram increment or attributing the wrong name.

A dedicated scripting language is proposed for definition of the histogramming scenario. The user should specify:

- The generic *arguments* — values or functions to be histogrammed, providing corresponding access methods and the titles.
- The *cuts*, also providing the appropriate access methods and their titles.
- The *filling scenario*:

— the *cut range*;

— the *histogram definition* (type, variable names, number of channels, axis info, optionally weight). The order of the lines defines the filling algorithm — the order, weight and applied cuts.

Section 1 describes the script usage, installation and trouble-shooting. The script language syntax is presented in Sec. 2. The conclusions are in the final section. The Appendix contains the examples of the histogramming scenario and the generated code.

1. INSTALLATION NOTES

The Perl script `hparse.pl` parses the scenario files and generates the code for the histogram booking, filling and saving. Perl is normally installed on your machine. You should be sure that the script has execution privileges:

```
> ls -o hparse.pl
-rwxrwxr-x    1 your_login     11697 Jul 24 17:04 hparse.pl
```

that is, x should be among the string -rwxrwxr-x; otherwise type

```
> chmod a+x  hparse.pl
```

Make a link to the script in the directory, which is in your PATH like this:

```
> ln -sf $PATH_TO_HPARSE/hparse.pl $DIRECTORY_IN_PATH/hparse
```

Here `$PATH_TO_HPARSE` stands for location of the `hparse.pl`, `$DIRECTORY_IN_PATH` stands for any user directory mentioned in environment variable `$PATH` (typically `$HOME/bin` or `$ALICE_ROOT/bin`). In this case the script can be executed in any directory just typing `hparse`.

The usage information can be displayed as

```
> hparse -h                    or
> hparse -?
```

The script interprets the histogramming scenario files (suggested extension is `hsf`). In order to run it, type

```
> hparse [-p prefix] [his_scenario_file_name]
```

By default it will start to process the file `main.hsf`. Using prefix (key -p), one can create histogramming files for several projects: script will use as primary input the file with name *prefix*`main.hsf` and store the output in files [*prefix*]`hist.h`, [*prefix*]`funct.h` and [*prefix*]`Histogramming.C`. The include file [*prefix*]`hist.h` normally contains the pointers to the generated histograms. The file [*prefix*]`funct.h` is used for cuts and argument definitions, namely for the interface with the function *FillHisto()*. The file [*prefix*]`Histogramming.C` contains the generated functions `BookHisto()` (for histogram booking), `FillHisto()` for filling, `ResetHisto(Option_t opt)` for resetting (obsolete?) and `WriteHisto()` for histogram saving in the root file.

That is, invoking

```
> hparse -p anal_
```

will parse the file `anal_main.hsf` and all the files included, the results can be found in files `anal_hist.h, anal_funct.h` and `anal_Histogramming.C`.

In case the script errors are detected by parser, it exits with an error code, thus stopping the make process.

2. SCRIPT SYNTAX

2.1. General. The line started with the sign # is ignored during parsing, becoming a comment line. No in-line comments are currently supported. The argument separators are blanks (spaces or tabs); they are ignored during parsing, thus the indenting is allowed. In case the blanks are required, the double quotes (" . . . ") should be used as item separators, e.g., the statement that cut `GoldenTrack(0.5, 0.5)` should be applied:

```
if GoldenTrack(0.5, 0.5) {
```

would cause an error, the correct spelling is either

```
if "GoldenTrack(0.5, 0.5)" {
```

or

```
if GoldenTrack(0.5,0.5) {
```

(note blanks between 0.5 and 0.5).

2.2. Arguments and Cuts. The variables and cuts should be predefined. The arguments and cuts could be defined as follows:

```
cut  name [value title]
arg  name [value title]
```

That is, the lines

```
cut  myCut myPointer->GetFoo()>25 "cut name"
arg  myValue myPointer->GetBar() "argument name"
if "myCut" {
   H1F  myValue 100 0 1
}
```

will be translated into

```
#include "hist.h"
void BookHistos()
{
  h00000 = new TH1F("h00000", "argument name {myCut} ", 100, 0, 1);
}
void FillHistos()
{
  if (myPointer->GetFoo() > 25) {
    h00000->Fill(myPointer->GetBar(), 1);
  }                              // end of the cut "myCut"
}
```

In case the `title` and `value` are not present, it is assumed that the argument `name` is the name of the function which should be defined in the file `funct.h`.

The cut and variable titles are used for automatic generation of the histogram titles. Reserved letter sequences, which will be processed during the parsing:

%i To be replaced with the argument **i** of the appropriate function (**i**=0, 1, 2. . .).

`$arg`**i** To be replaced with the argument **i** of the include file at current level; (**i**=0, 1 or 2).

If `ArgTitle` is substituted literally, without the conversion to the character string, the double-quoted (" . . . ") string is processed automatically. The title without double quotes is interpreted as `char *` variable. This would enable overriding the default histogram title.

The cut could be applied as: `if cutName {`. The title of the appropriate cut will be appended to the automatically generated titles of all histograms defined inside the braces {}.

2.3. C++ Statements. One can add flexibility and optimization (beware: not transparency! :) — use with moderation) including the pieces of C++ code in the histogramming algorithms. The following options are foreseen:

‘ stands for verbatim insert of the rest of the line in the function FillHisto()

’ stands for verbatim insert of the rest of the line in the function BookHisto()

@ stands for verbatim insert of the rest of the line in the file func.h

2.4. Histogram Definitions. The one- and two-dimension procedures are supported.

```
H1X argName nChannels x0 xF [weight]
H2X argName1 argName2 nChannels1 x0 xF nChannels2 y0 yF [weight]
```

Here `X` stands for `I, F, U, D, S` (all possibilities defined by root `TH1` and `TH2`). Weight is an optional argument.

2.5. Include Files. Directive to process lines in a file RuleFileileName

```
include        RuleFileileName arg1 arg2 arg3
```

Up to 3 arguments are currently supported; they are referenced as `$argN (N=0,1,2)` in the included file. It is easy to increase this number if needed.

2.6. Control Statements. The `do`-loops could be used for the definition of the repetitive histograms. The following syntax is implemented:

```
do doVar= ind_beg ind_fin [step]
   ........
enddo
```

By default the value of the optional argument `step` is 1. The text between the `do` and the `enddo` statements repeats `(ind_fin-ind_beg)/step` times, each occurrence of the `doVar` is *literally* replaced by current value of the index (changing from `ind_beg` to `ind_fin-1`). That is why it is strongly recommended to use unusual character sequences for the `doVar`; e.g.,

```
do i_= 1 3
   SomethingUseful(i_)
enddo
```

will be translated into

```
SomethingUseful(1)
SomethingUseful(2)
```

while

```
do i= 1 3
   SomethingUseful(i)
enddo
```

will generate the following code:

```
   Someth1ngUseful(1)
   Someth2ngUseful(2)
```

Currently the folded do-loops are not supported; however, if needed, there is a simple work-around to place the inner do-loops in the included files; for example,

```
do i_= 1 5
  do j_ =02
    SomeLines(i_,j_)
  enddo
enddo
```

will not work as naively expected, while the same effect could be achieved by

```
do i_= 1 5
  include FileWithDoLoop i_
enddo
```

The contents of the *FileWithDoLoop* is

```
 do j_ =0 2
    SomeLines($arg0,j_)
 enddo
```

The `end` statement terminates parsing of the current file.

2.7. Comments. The keyword starting with `Comm` defines the comment string to be included in the titles of the histograms between the cut list and weight. Examples:

```
# At the beginning of the include file processing
# the comment string is automatically reset.
# No comments below
   ...
Comment "my comment"
# the string "my comment" will be inserted in histo titles
  ...
Comm "$arg1"
# the value of the current include file argument 1 will be used as
```

```
# comment in the histos below
  ...
Comm ""
# resetting comment string
  ...
```

CONCLUSIONS

The described facility is general-purpose; it could be used in conjunction with any ROOT-based analysis code which provides access to its intermediate results. More specifically, as *advantages* one can mention:

• The script is compact, modular, well structured, and more readable/understandable than the regular C++ code, especially in case of complicated histogramming scenario.

• The correspondence of the histogram names given at booking time and filling algorithm is normally automatic.

• Multilevel script coding by means of the file including and the argument passing features allows the code modularity, re-usability and ease of the configuration.

• The generated code is one-level, straightforward and well optimized.

• The resulting code is more easily maintainable and changeable.

However, the alpha testers pointed out the *drawbacks* of the approach:

• One should learn new (though simple) language prior to use it.

• The error-finding procedure is not straightforward: in order to fix the logical errors one should analyze the C++ file and only then correct the appropriate script.

• Lack of the run-time parameters support. That is, one can book/process only predefined number of histograms. The workaround (using included C++ code) is not transparent enough.

Plans:

• To implement the statement '`for i_ A B C .. {..}`' that executes the operators in parenthesis with `i_` equal to each of the list.

• The parser needs improved error detecting facility.

• It may (??) be useful to implement the Folded do-loops.

• The GUI for automatization of the script might be useful.

Known Bugs. The following unpleasant features should be mentioned:

• The do-loop variable should not contain the `$` sign.

• The argument and cut names are global within the project. In some sense this is an advantage; on the other hand, the double definition would cause a compilation error in generated code.

• The function arguments are not checked during the parsing.

Acknowledgements. The authors would like to thank L. Luquin, J.-P. Cussenau and Ch. Finck of the Nantes SUBATECH team for their hospitality and useful discussions. The work was supported by INTAS, grant 00-00538.

A. EXAMPLES

A.1. Scripts.

A.1.1. main.hsf. Demonstrates the modularity, usage of the cuts and do loops:

```
@// Analysis classes definitions:
@#include "MyStuff.h"

# Definition of a histogramming argument as macro
 arg evLeng gDate->Event()->Length() "Event length (words)"

# And 1-D histogram using this argument
 H1I evLeng 100 0 5000

# Define raw histos for 1 C-RAMS input 0 (i_ stands for loop index)
do i_=0 1
  include RawCrams.hsf i_
enddo


# Definition of a cut as function
@// example of the functional cut:
 cut  GoldenTrack
# Function body optionally stored in the file func.h
@ int GoldenTrack(Float_t chiX, Float_t chiY){
@   if (gDateMuonTracker->GetChi2X() < chiX  &&
@       gDateMuonTracker->GetChi2Y() < chiY) return 1;
@   else return 0;
@ }

# One can use pointers to the objects to pass the arguments
# from the analysis to the histogramming routines:
@ MyClass *myPointer;
cut GoodCluster "myPointer->GetCluster($arg0) > 0"

# Present chamber 0 and 1 info for good tracks only
if  "GoldenTrack(0.5, 0.5)" {
 do i@j = 0 2
  include Chamber.hsf i@j
 enddo
}
end
```

A.1.2. RawCrams.hsf

```
# include in FillHisto(): get pointers to the frequently used objects
```

```
‘ AliDareEvent      *ev = gDate->Event();               // Current event
# Here include file argument 1 ($arg0) is the C-RAMS input number
‘ AliDateCramsInput *cr = gDate->Event()->Crams($arg0); // Current C-Rams
     In

arg leng ev->CramsDataL($arg0) #title
arg ampl cr->GetAml(i)         #title
arg chan cr->GetChan(i)        #title

# BookHisto(): generate the title
’ // This is an example of the self-made histogram title
’  char title[80];
’  sprintf(title,"C-RAMS %d Event length",
      gDate->Header->GetCramsSerN($arg0));
   H1I leng 200 0 2000

’  Short_t nClocks=gDate->Header->GetCramsNofClocks($arg0);
’  sprintf(title,"CRAMS %d Channel hitted",
      gDate->Header->GetCramsSerN($arg0));
‘  for (int i=0; i<gDate->Event->CramsDataL($arg0); i++){
    H1I chan nClocks 0 nClocks
’   sprintf(title,"C-RAMS %d Ampl at Channel hitted",
’         gDate->Header->GetCramsSerN($arg0));
    H1I chan nClocks 0 nClocks ampl
‘ } // End of FillRawCrams

end
```

A.2. Chamber.hsf. A rather simplified version of the Chamber presentation histos:

```
 arg resY (gDateMuonChamber->Chamber($arg0))->GetResY()
    "Y-resolution of CPC-$arg0"
 arg resX (gDateMuonChamber->Chamber($arg0))->GetResX()
    "X-resolution of CPC-$arg0"

 H1F resY    200 0 2.

 if  "GoodCluster" {
  do i_j = 1 2
     H1F resX    200 0 2.
 enddo
}

end
```

A.3. Parsing Results. We do not present here the resulting header file `hist.h` and functions `ResetHisto()` and `WriteHisto()` as they are trivial.

A.3.1. BookHistos()

```
// Automatically generated by hparse v 0.005
// at 21:30:34 on February, 10 2002
//
#include "hist.h"
void BookHistos()
{
  h00000 = new TH1I("h00000", "Event length (words) ", 100, 0, 5000);
  // This is an example of the self-made histogram title
  char title[80];
  sprintf(title,"C-RAMS %d Event length",gDate->Header->GetCramsSerN(0));
  h00001 = new TH1I("h00001", title, 200, 0, 2000);
  Short_t nClocks = gDate->Header->GetCramsNofClocks(0);
  sprintf(title, "CRAMS %d Channel hitted",
  gDate->Header->GetCramsSerN(0));
  h00002 = new TH1I("h00002", title, nClocks, 0, nClocks);
  sprintf(title, "C-RAMS %d Ampl at Channel hitted",
  gDate->Header->GetCramsSerN(0));
  h00003 = new TH1I("h00003", title, nClocks, 0, nClocks);
  h00004 =
      new TH1F("h00004", "Y-resolution of CPC-0
                {GoldenTrack(0.5, 0.5)} ", 200, 0, 2.);
  h00005 =
      new TH1F("h00005",
               "X-resolution of CPC-0 {GoldenTrack(0.5, 0.5) &
                GoodCluster} ", 200, 0, 2.);
  h00006 =
      new TH1F("h00006", "Y-resolution of CPC-1
               {GoldenTrack(0.5, 0.5)} ", 200, 0, 2.);
  h00007 =
      new TH1F("h00007",
               "X-resolution of CPC-1 {GoldenTrack(0.5, 0.5) &
               GoodCluster} ", 200, 0, 2.);
}
```

A.3.2. FillHistos()

```
// Automatically generated by hparse v 0.005
//  at 21:30:34 on  February, 10 2002
//
#include "hist.h"
void FillHistos()
{
  h00000->Fill(gDate->Event()->Length(), 1);
  AliDareEvent *ev = gDate->Event();    // Current event
  AliDateCramsInput *cr = gDate->Event()->Crams(0); // Current C-Rams In
```

```
  h00001->Fill(ev->CramsDataL(0), 1);
  for (int i = 0; i < gDate->Event->CramsDataL(0); i++) {
    h00002->Fill(cr->GetChan(i), 1);
    h00003->Fill(cr->GetChan(i), cr->GetAml(i));
  }                             // End of FillRawCrams
  if (GoldenTrack(0.5, 0.5)) {
    h00004->Fill((gDateMuonChamber->Chamber(0))->GetResY(), 1);
    if (myPointer->GetCluster($arg0) > 0) {
      h00005->Fill((gDateMuonChamber->Chamber(0))->GetResX(), 1);
    }                           // end of the cut "GoodCluster"
    h00006->Fill((gDateMuonChamber->Chamber(1))->GetResY(), 1);
    if (myPointer->GetCluster($arg0) > 0) {
      h00007->Fill((gDateMuonChamber->Chamber(1))->GetResX(), 1);
    }                           // end of the cut "GoodCluster"
  }                             // end of the cut "GoldenTrack(0.5, 0.5)"
}
```

A.3.3. funct.h

```
// Automatically generated by hparse v 0.005
// at 21:30:34 on February, 10 2002
#ifndef ALIDATEFUNCT_H
#define ALIDATEFUNCT_H
#ifndef __CINT__
#include "TROOT.h"
#include "TH1.h"
#include "TH2.h"
#endif
// Analysis classes definitions:
#include "MyStuff.h"
// example of the functional cut:
int GoldenTrack(Float_t chiX, Float_t chiY)
{
  if (gDateMuonTracker->GetChi2X() < chiX &&
      gDateMuonTracker->GetChi2Y() < chiY)
    return 1;
  else
    return 0;
}

MyClass *myPointer;
#endif
```

Received on May 19, 2004.