ROOT user experience

ROOT Users Workshop

Ruggero Turra

Università degli Studi di Milano & INFN

March 12, 2013

Introduction

My background

- I work in the ATLAS collaboration since 2010
- I am involved in data analysis ($H \to \gamma\gamma$) and in performance (energy calibration)
- I started to use ROOT in the first year of university, during the "Physics laboratory 1.0.1"
- I work 80% in python using various libraries: ROOT, numpy/scipy, matplotlib, . . . , Proof, TMVA, RooFit, RooStats

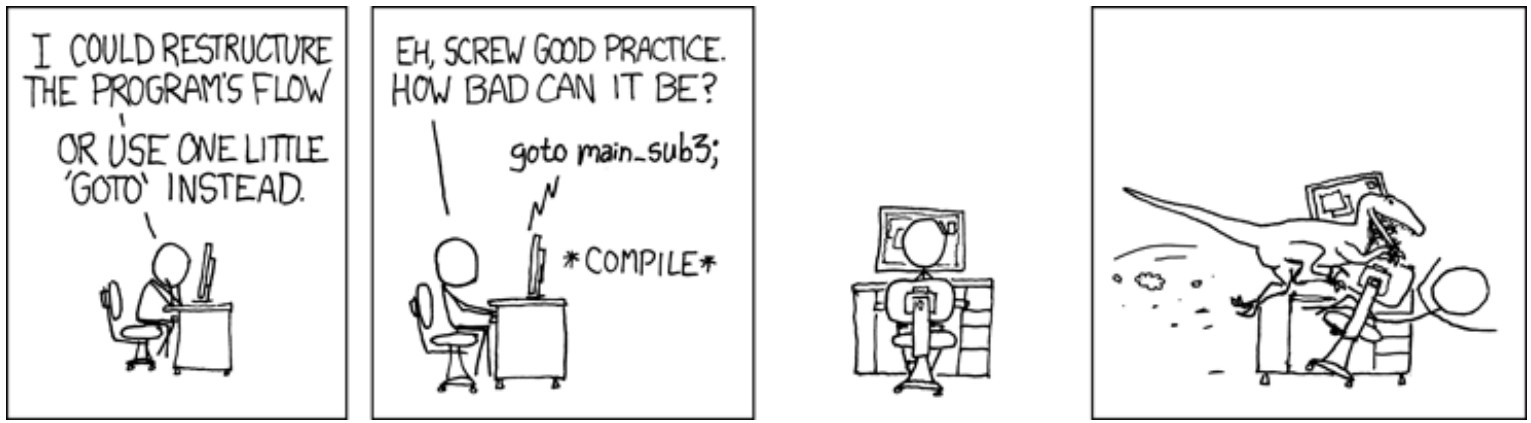
- ROOT is widely used and supported
- it provides you many useful tools (TString, TPRegexp, . . .), no need to include other libraries, very easy to share code
- very fast
- flexible I/O
- nice integration with python

```
1  import ROOT
2  ROOT.gROOT.ProcessLine(".x \$ROOTCOREDIR/scripts/load_packages.C+");
3  ROOT.my_cpp_function()
```

Some considerations about ROOT users

A lot of users...

- prefer a quick and dirty approach



- don't compile code
- have only a basic C++ knowledge
- don't know any alternative package for data analysis (PAW)
- don't know about ownership in ROOT (white canvases, double delete)
- have troubles with pointers (memory leaks)
- very small feedback, they prefer workarounds

Some considerations about ROOT users/2

Personal utilities

- some of them have their own `MyRootUtils`, very simple tasks (convert histograms to graphs, . . .)

```
from PyH4l import ROOTDefs, PyH4lTools

fPowHeg = ROOT.TFile("Hgg_mc10b_VBFH120_PowHeg.root")
fHerwig = ROOT.TFile("Hgg_mc10b_VBFH120_Herwig.root")

tPowHeg = fPowHeg.tree
tHerwig = fHerwig.tree

x = ROOTDefs.Draw([tPowHeg, tHerwig],
      "pT", "20./(20 + 10*(pT > 200) + 30*(pT > 300) + 50*(pT>400))",
      binning=range(0,200,10) + range(200, 300, 20) + [300, 350, 400, 500], ↩
           norm=1,
      legend_labels=["PowHeg", "Herwig"], logy=1, min=1e-5, max=0.2, ↩
           markersize=0.8,
      legend_options="PL", ytitle = "dN/dP_{T} / 10 GeV", xtitle="P_{T}^{#↩
           gamma#gamma} [GeV]")

hDiv = ROOTDefs.H1Divide(x.plots[0], x.plots[1], xtitle="P_{T}^{#gamma#gamma} [↩
     GeV]", ytitle="PowHeg / Herwig")
hDiv.Draw()
```

Some consideration about ROOT and C++

ROOT and C++ don't help them very much:

- C++ is extremely difficult
- C++ is not quick, but it can be very dirty
- ROOT is not forcing the user to write safe code (avoiding pointers for example)
- some internal ROOT mechanisms are too implicit and magic[1]

[1] from Wikipedia: "magic is an informal term for abstraction - it is used to describe code that handles complex tasks while hiding that complexity to present a simple interface. The term is somewhat tongue-in-cheek and carries bad connotations, implying that the true behavior of the code is not immediately apparent"

ROOT Object Ownership

average user

```
1  void f(TCanvas *canvas)
2  {
3      canvas->cd();
4      TH1F* h = new TH1F("h", "h", 10, 0, 10);
5      h->Fill(2);
6      h->Fill(3);
7      h->Draw();
8  }
9
10 int main()
11 {
12     TCanvas* canvas = new TCanvas();
13     f(canvas);
14 }
```

ROOT Object Ownership

try to avoid memory leak is (blank canvas)

```
1  void f(TCanvas *canvas)
2  {
3      canvas->cd();
4      TH1F* h = new TH1F("h", "h", 10, 0, 10);
5      h->Fill(2);
6      h->Fill(3);
7      h->Draw();
8      delete h;
9  }
10
11 int main()
12 {
13     TCanvas* canvas = new TCanvas();
14     f(canvas);
15     // wait here
16     delete canvas;
17 }
```

ROOT Object Ownership

try to avoid pointers

```
1  void f(TCanvas & canvas)
2  {
3      canvas->cd();
4      TH1F h("h", "h", 10, 0, 10);
5      h.Fill(2);
6      h.Fill(3);
7      h.Draw();
8  }
9
10 void a()
11 {
12     TCanvas* canvas = new TCanvas();
13     f(*canvas);
14 }
```

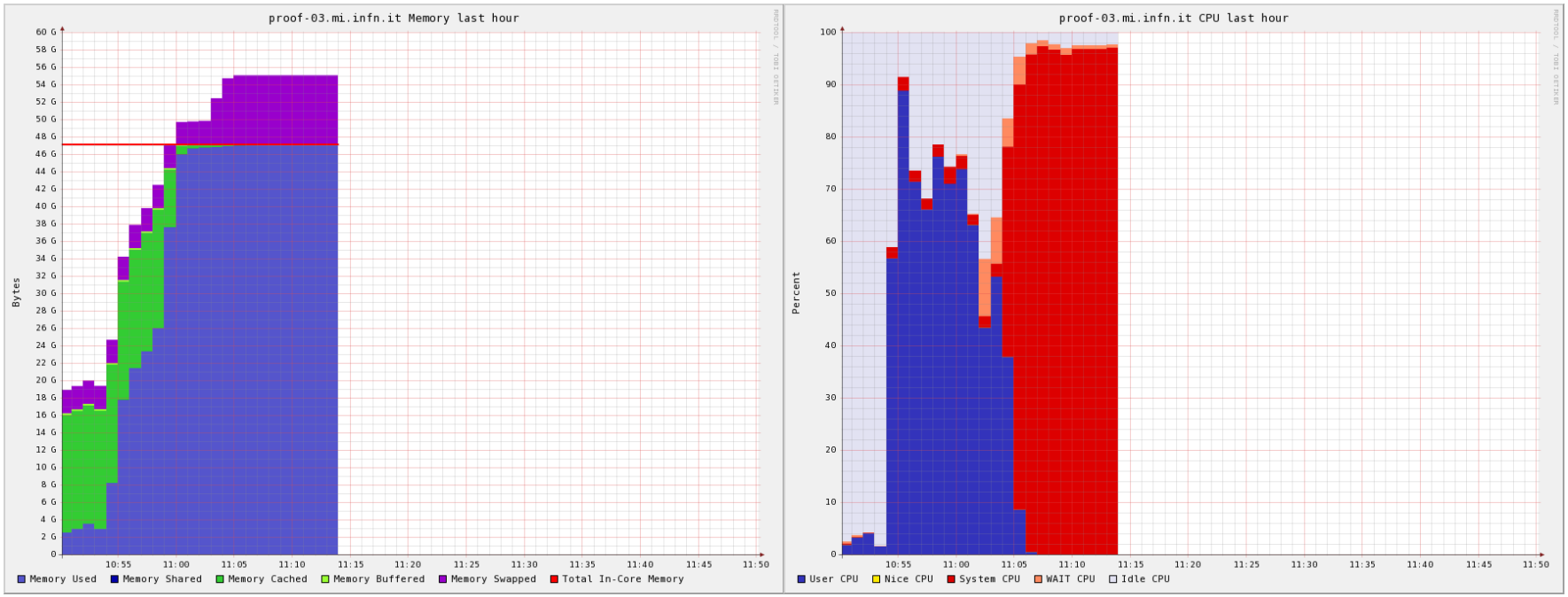
ROOT Object Ownership

try to use python

```
1  import ROOT
2
3  def f(canvas):
4      canvas.cd()
5      h = ROOT.TH1F("h", "h", 10, 0, 10)
6      h.Fill(2)
7      h.Fill(3)
8      h.Draw()
9
10 if __name__ == "__main__":
11     canvas = ROOT.TCanvas()
12     f(canvas)
13     raw_input()
```

Consequences

Very frequent: this morning!



- in addition al lot of ROOT method return pointer: it not always clear who has the responsibility for the memory (`TProfile* TH2::ProfileX`)

Other memory model: matplotlib example

matplotlib example

```
1  import matplotlib.pyplot as plt
2
3  def f(pad):
4      n, bins, patches = pad.hist([2,3], bins=10, range=(0,10))     (1)
5
6  if __name__ == "__main__":
7      fig = plt.figure()
8      ax = fig.add_subplot(111)     (2)
9      f(ax)
10     plt.show()     (3)
```

- explict ownership: (2) `fig` owns `ax`, (1) `ax(pad)` owns the histogram
- (3) explict show

Other memory model: gtk+

gtk+ example

```
1  void f(GtkWidget* window) {
2      GtkWidget *button = gtk_button_new_with_label ("Hello World");
3      gtk_container_add(GTK_CONTAINER (window), button);     (1)
4      gtk_widget_show(button);     (2)
5  }
6
7  int main() {
8      GtkWidget *window = gtk_window_new(GTK_WINDOW_TOPLEVEL);     (3)
9      f(window);
10     gtk_widget_show(window);     (4)
11     gtk_main();
12 }
```

- (1) explict ownership
- (2) (4) explict show
- (3) reference counter

Other memory model: gtkmm

gtkmm example

```
1  struct HelloWorldWindow : public Gtk::Window
2  {
3    HelloWorldWindow() : m_button("Hello World") {
4      add(m_button);      (1)
5      m_button.show();    (2)
6    }
7    Gtk::Button m_button;   (3)
8  };
9
10 int main(int argc, char** argv) {
11   Glib::RefPtr<Gtk::Application> app = Gtk::Application::create(argc, argv);   (4)
12   HelloWorldWindow helloworld;   (5)
13   return app->run(helloworld);
14 }
```

- (1) explict ownership
- (2) explict show
- (3) (5) no pointers at all
- (4) reference-counting smartpointer

Another common mistake

gtkmm example

```
1  std::vector<TObject*> get_histograms(std::string filename) {
2     std::vector<TObject*> result;
3     TFile f(filename.c_str());
4     result.append(f.Get("histo1"));
5     result.append(f.Get("histo2"));
6     return result;
7  };
8
9  int main() {
10    std::vector<TObject*> histograms = get_histograms("my_file.root");
11    draw_with_mystyle(histograms);
12 }
```

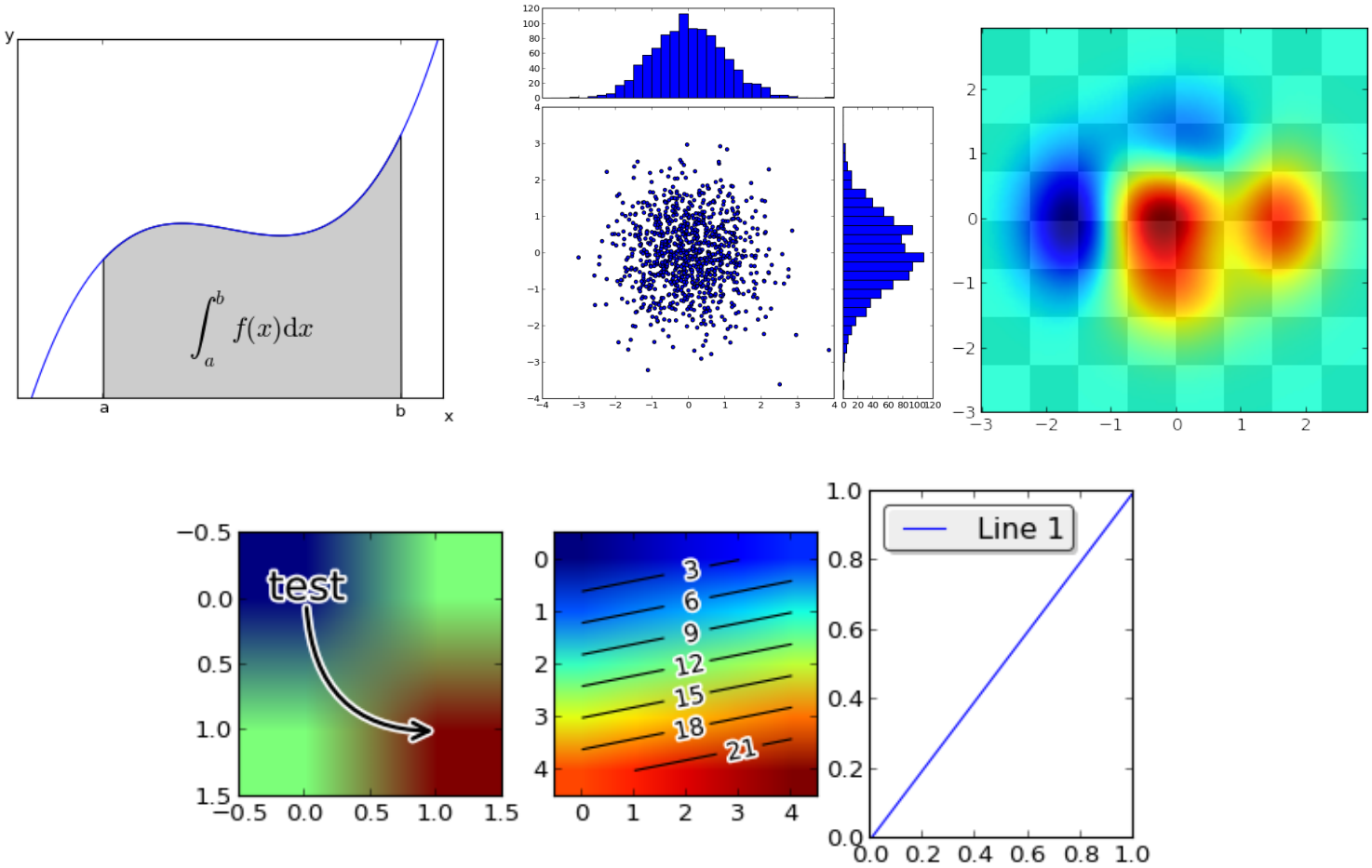
Why I am using python

- productivity
- C++ doesn't have a large std library (as java)
- because I can do:

```
1 "histo_" + str(10)
2 logging.error("file not found")
3 os.listdir(".")
4 files = glob("data8TeV*_p1234")
5 re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Ruggero Turra")
6 p = multiprocessing.Pool(5); p.map(f, data)
```

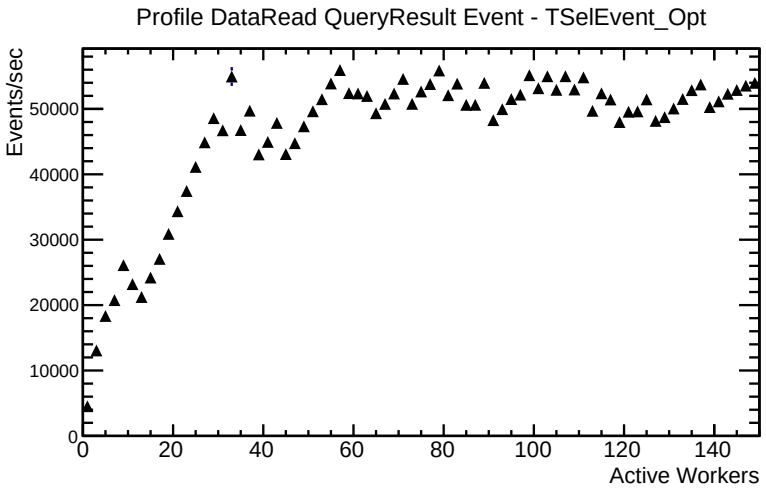
- no pointers
- no compilation, try and fix approach
- everybody can learn python in one week
- of course you can't do everything in python (e.g. cutflow, skimming, ...)
- time = time to develop + time to run

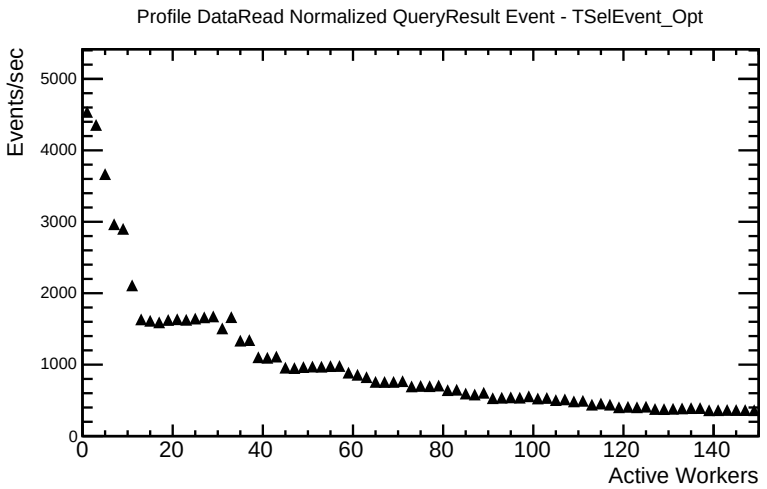
Why I am using matplotlib

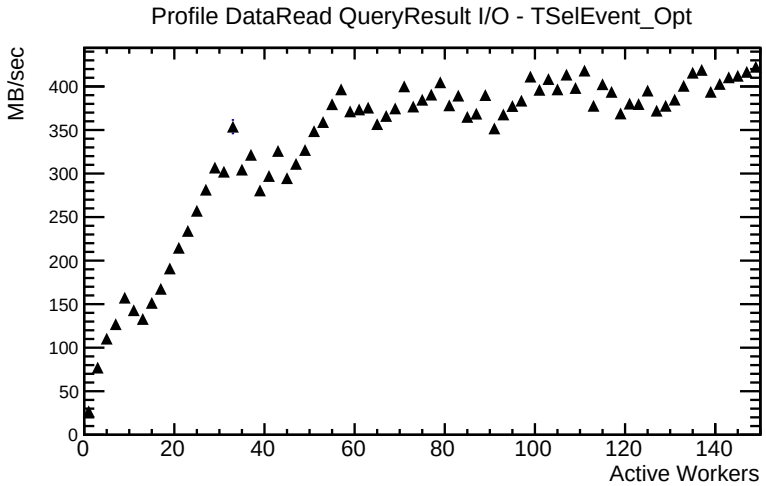


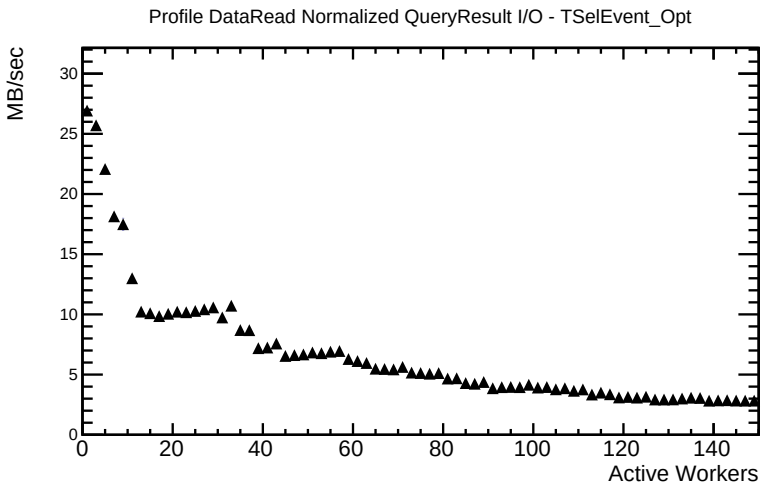
- it is a bit slow
- very high customizability

Why I am using proof









My workflow

First dump the data: branch → memory (array)

dump data (simplified)

```
1  import ROOT
2  import numpy as np
3
4  def GetValuesFromTreeWithProof(tree, variable, cut = ""):
5    variable = "Entry\$:" + variable    (1)
6    if not hasattr(ROOT, "gProof"):
7      ROOT.TProof.Open("")
8    if not isinstance(tree, ROOT.TChain):    (2)
9      chain = ROOT.TChain(tree.GetName())
10     chain.Add(tree.GetDirectory().GetName())
11     tree = chain
12   tree.SetProof(True)
13
14   N = tree.Draw(variable, cut)    (3)
15
16   output = ROOT.gProof.GetOutputList()[2]
17   i, y = output.GetX(), output.GetY()
18   i.SetSize(N)
19   y.SetSize(N)
20
21   i = np.array(i, dtype=int)
22   y = np.array(y)
23
24   sorting_index = np.argsort(i)    (4)
25   y = y[sorting_index]
26
27   return y
```

- (1) (4) sort the output with entry number
- (2) why `TTree` doesn't have `SetProof`?
- (3) use `Draw` method to dump the values

My workflow

select categorize compute and draw

```
1   selection = "el_is_tight"
2   el_EoverEtrue = GetValuesFromTreeWithProof(mychain, "el_E/el_truth_E",
3                                              selection)    (1)
4   el_eta = GetValuesFromTreeWithProof(mychain, "el_eta", selection)
5
6   is_barrel = np.abs(el_eta) < 1.425
7   is_endcap = np.abs(el_eta) > 1.55
8   el_EoverEtrue_barrel = el_EoverEtrue[is_barrel]
9   el_EoverEtrue_endcap = el_EoverEtrue[is_endcap]
10
11  resolution_barrel = np.std(el_EoverEtrue_barrel)    (2)
12  resolution_endcap = np.std(el_EoverEtrue_endcap)
13
14  # produce histograms, correlations, ...
```

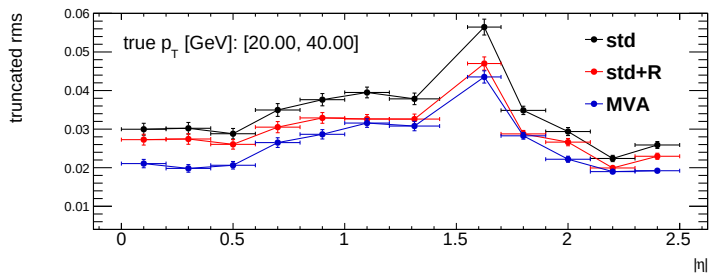
- (1) dump values in memory (can be problematic with large datasets)
- (2) easy to parallelize

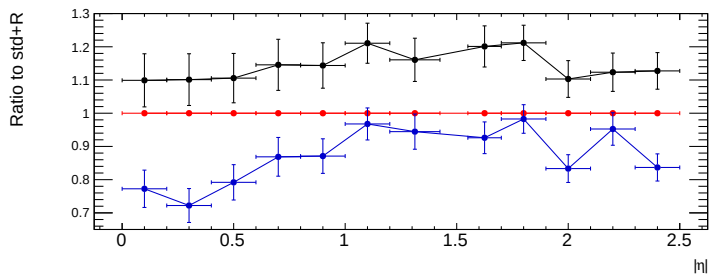
What I am actually doing

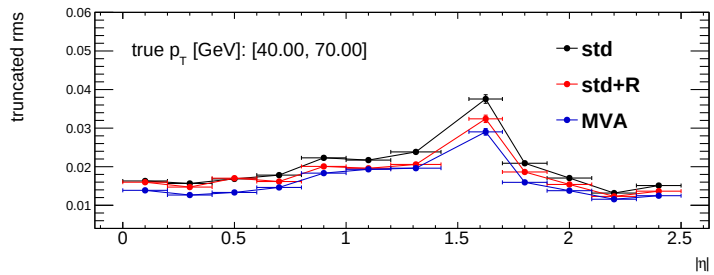
some point in common with WebOOT by Peter

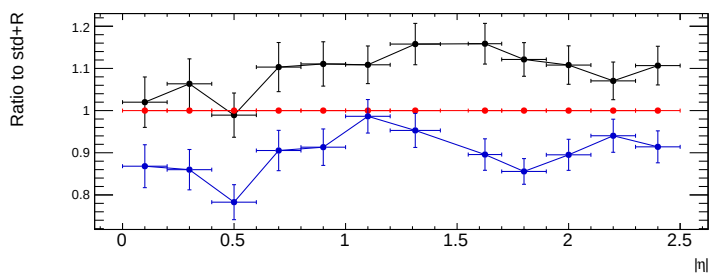
```
1  import BinAndDraw as bad
2
3  quantities = ("ph_E/ph_truth_E",)
4  estimators = bad.estimators(bad.truncated_rms(outliers=0.95), bad.↩
       gaussian_width(), bad.FWHM())), display_as=bad.CANVAS)
5
6  dataset_std = bad.dataset_from_glob("std_calibration/*.root", label="std")
7  dataset_stdR = bad.dataset_from_glob("stdR_calibration/*.root", label="std+R")
8  dataset_MVA = bad.dataset_from_glob("MVA_calibration/*.root", label="MVA")
9  datasets = (dataset_std, dataset_stdR, dataset_MVA)
10
11 var_dataset = bad.vartype.different_dataset(datasets, display_as=bad.COLOR)
12 var_eta = bad.vartype.branch("abs(ph_eta)", bins=np.linspace(0,2.5,10), ↩
       display_as=bad.XAXIS, label="|#eta|")
13 var_pt = bad.vartype.branch("ph_truth_pt/1E3", bins=(20,40,70,100,500), ↩
       display_as=bad.CANVAS, label="true p_{T} [GeV]")
14
15 loops = bad.loop(var_dataset * var_eta * var_pt, var_dataset * var_pt) * bad.↩
       loop(estimators)
16
17 bad.run(quantities, loops,
18         global_selection="ph_is_tight", output_dir = "output",
19         plot_ratio = ("x/y", dataset_stdR),
20         use_proof=True, workers=4)
```

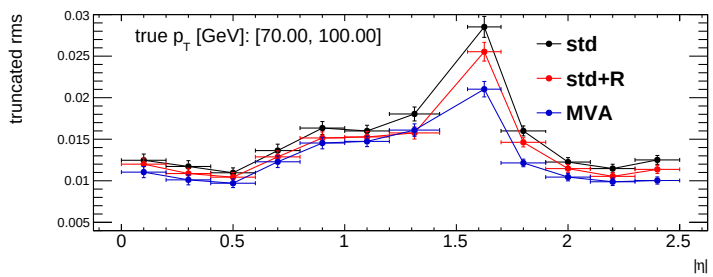
result of `(var_dataset * var_eta * var_pt) * truncated_rms`

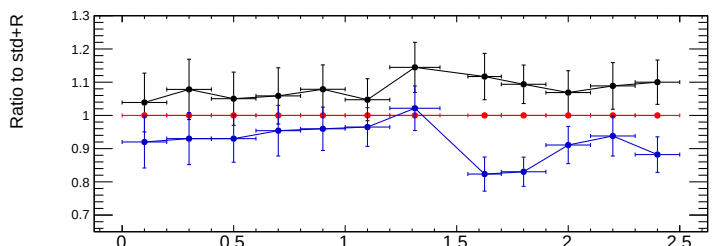


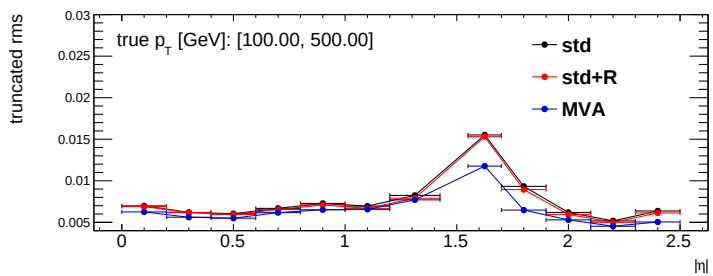


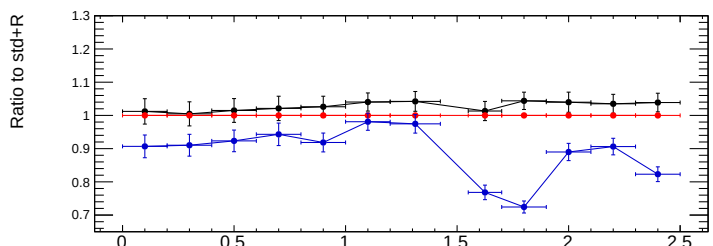












What I would like to see in ROOT/1

1. support alternative API, more explicit API, no magic, hidden mechanisms. (see rootpy talk by Noel)

What I would like to see in ROOT/1

1. support alternative API, more explicit API, no magic, hidden mechanisms. (see rootpy talk by Noel)
2. Better memory model (ownership)

What I would like to see in ROOT/1

1. support alternative API, more explicit API, no magic, hidden mechanisms. (see rootpy talk by Noel)
2. Better memory model (ownership)
3. less pointers, force the user to write safe code (no pointers, smart pointers, . . .)

What I would like to see in ROOT/1

1. support alternative API, more explicit API, no magic, hidden mechanisms. (see rootpy talk by Noel)
2. Better memory model (ownership)
3. less pointers, force the user to write safe code (no pointers, smart pointers, . . .)
4. extend `TTree::Draw` and similar interactive-oriented methods

What I would like to see in ROOT/1

1. support alternative API, more explicit API, no magic, hidden mechanisms. (see rootpy talk by Noel)
2. Better memory model (ownership)
3. less pointers, force the user to write safe code (no pointers, smart pointers, . . .)
4. extend `TTree::Draw` and similar interactive-oriented methods
5. extend `TTree::Friend` idea (proof)

What I would like to see in ROOT/1

1. support alternative API, more explicit API, no magic, hidden mechanisms. (see rootpy talk by Noel)
2. Better memory model (ownership)
3. less pointers, force the user to write safe code (no pointers, smart pointers, . . .)
4. extend `TTree::Draw` and similar interactive-oriented methods
5. extend `TTree::Friend` idea (proof)
6. support (full) c++ in interactive session (no more `#ifdef __CINT__`)

What I would like to see in ROOT/1

1. support alternative API, more explicit API, no magic, hidden mechanisms. (see rootpy talk by Noel)
2. Better memory model (ownership)
3. less pointers, force the user to write safe code (no pointers, smart pointers, ...)
4. extend `TTree::Draw` and similar interactive-oriented methods
5. extend `TTree::Friend` idea (proof)
6. support (full) c++ in interactive session (no more `#ifdef __CINT__`)
7. do we need to reinvent the wheel? (TMath, TPRegexp, THtml, GUI, ...?)

What I would like to see in ROOT/1

1. support alternative API, more explicit API, no magic, hidden mechanisms. (see rootpy talk by Noel)
2. Better memory model (ownership)
3. less pointers, force the user to write safe code (no pointers, smart pointers, . . .)
4. extend `TTree::Draw` and similar interactive-oriented methods
5. extend `TTree::Friend` idea (proof)
6. support (full) c++ in interactive session (no more `#ifdef __CINT__`)
7. do we need to reinvent the wheel? (TMath, TPRegexp, THtml, GUI, . . . ?)
8. use c++11 libraries (~~math~~, regex, . . .)

What I would like to see in ROOT/1

1. support alternative API, more explicit API, no magic, hidden mechanisms. (see rootpy talk by Noel)
2. Better memory model (ownership)
3. less pointers, force the user to write safe code (no pointers, smart pointers, ...)
4. extend `TTree::Draw` and similar interactive-oriented methods
5. extend `TTree::Friend` idea (proof)
6. support (full) c++ in interactive session (no more `#ifdef __CINT__`)
7. do we need to reinvent the wheel? (TMath, TPRegexp, THtml, GUI, ...?)
8. use c++11 libraries (~~math~~, regex, ...)
9. support other input/output formats (xml, HDF5, ...), see "Mathematica with ROOT" by Prof. Sebastian White

What I would like to see in ROOT/1

1. support alternative API, more explicit API, no magic, hidden mechanisms. (see rootpy talk by Noel)
2. Better memory model (ownership)
3. less pointers, force the user to write safe code (no pointers, smart pointers, . . .)
4. extend `TTree::Draw` and similar interactive-oriented methods
5. extend `TTree::Friend` idea (proof)
6. support (full) c++ in interactive session (no more `#ifdef __CINT__`)
7. do we need to reinvent the wheel? (TMath, TPRegexp, THtml, GUI, . . . ?)
8. use c++11 libraries (~~math~~, regex, . . .)
9. support other input/output formats (xml, HDF5, . . .), see "Mathematica with ROOT" by Prof. Sebastian White
10. python as interactive session? notebook as Mathematica/Maple/. . . (ipython notebook?)

IPy IPython Dashboard | IPy spectrogram

127.0.0.1:8888/a5222740-848b-4ac1-b212-d732c9f8f78b

IP[y]: Notebook spectrogram Last saved: Mar 07 11:14 PM

File Edit View Insert Cell Kernel Help

Markdown

Simple spectral analysis

An illustration of the Discrete Fourier Transform

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} kn} \qquad k = 0, \ldots, N-1$$

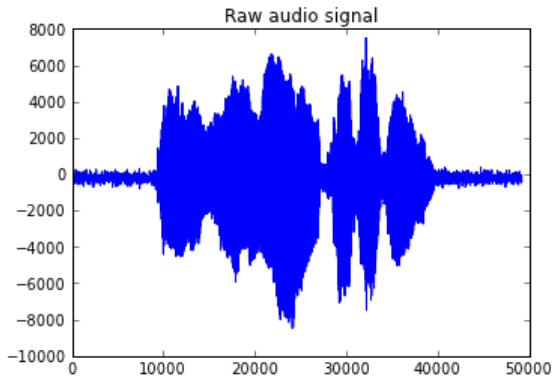
using windowing, to reveal the frequency content of a sound signal.

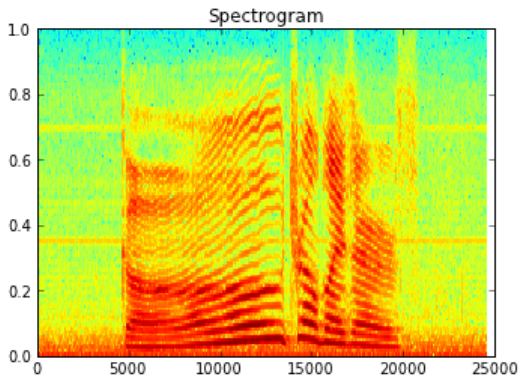
We begin by loading a datafile using SciPy's audio file support:

In [1]:
```
from scipy.io import wavfile
rate, x = wavfile.read('test_mono.wav')
```

And we can easily view its spectral structure using matplotlib's builtin `specgram` routine:

In [2]:
```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))
ax1.plot(x); ax1.set_title('Raw audio signal')
ax2.specgram(x); ax2.set_title('Spectrogram');
```





What I would like to see in ROOT/2

1. use c++03(98) features: template, exceptions, STD library

What I would like to see in ROOT/2

1. use c++03(98) features: template, exceptions, STD library
2. provide copy-constructors: `return my_canvas;`

What I would like to see in ROOT/2

1. use c++03(98) features: template, exceptions, STD library
2. provide copy-constructors: `return my_canvas;`
3. separate data from presentation (TH1F, ...)

What I would like to see in ROOT/2

1. use c++03(98) features: template, exceptions, STD library
2. provide copy-constructors: `return my_canvas;`
3. separate data from presentation (TH1F, ...)
4. improve the visual result (e.g. support alpha channel, automatic legend placement)

What I would like to see in ROOT/2

1. use c++03(98) features: template, exceptions, STD library
2. provide copy-constructors: `return my_canvas;`
3. separate data from presentation (TH1F, ...)
4. improve the visual result (e.g. support alpha channel, automatic legend placement)
5. better class hierarchy (e.g. why `TGraph` inherits from `TAttLine`? why `TH1::GetZaxis()`?)

What I would like to see in ROOT/2

1. use c++03(98) features: template, exceptions, STD library
2. provide copy-constructors: `return my_canvas;`
3. separate data from presentation (TH1F, ...)
4. improve the visual result (e.g. support alpha channel, automatic legend placement)
5. better class hierarchy (e.g. why `TGraph` inherits from `TAttLine`? why `TH1::GetZaxis()`?)
6. proof: support sharing of resources: why declare one histogram for every workers instead of sharing it?

What I would like to see in ROOT/2

1. use c++03(98) features: template, exceptions, STD library
2. provide copy-constructors: `return my_canvas;`
3. separate data from presentation (TH1F, ...)
4. improve the visual result (e.g. support alpha channel, automatic legend placement)
5. better class hierarchy (e.g. why `TGraph` inherits from `TAttLine`? why `TH1::GetZaxis()`?)
6. proof: support sharing of resources: why declare one histogram for every workers instead of sharing it?
7. better `SetBranchStatus`, read on demand: `TTreeReader`!

What I would like to see in ROOT/2

1. use c++03(98) features: template, exceptions, STD library
2. provide copy-constructors: `return my_canvas;`
3. separate data from presentation (TH1F, ...)
4. improve the visual result (e.g. support alpha channel, automatic legend placement)
5. better class hierarchy (e.g. why `TGraph` inherits from `TAttLine`? why `TH1::GetZaxis()`?)
6. proof: support sharing of resources: why declare one histogram for every workers instead of sharing it?
7. better `SetBranchStatus`, read on demand: `TTreeReader`!
8. generalize TSelector (run on batch queues, grid, ...), see EventLoop (see Attila talk), sframe

What I would like to see in ROOT/2

1. use c++03(98) features: template, exceptions, STD library
2. provide copy-constructors: `return my_canvas;`
3. separate data from presentation (TH1F, ...)
4. improve the visual result (e.g. support alpha channel, automatic legend placement)
5. better class hierarchy (e.g. why `TGraph` inherits from `TAttLine`? why `TH1::GetZaxis()`?)
6. proof: support sharing of resources: why declare one histogram for every workers instead of sharing it?
7. better `SetBranchStatus`, read on demand: `TTreeReader`!
8. generalize TSelector (run on batch queues, grid, ...), see EventLoop (see Attila talk), sframe
9. restructure old code: ROOT has one `goto` (ROOT has one `goto` every 900 lines, V8 JavaScript Engine every 84k, minuit every 29 lines, boost every 1000 lines.

What I would like to see in ROOT/2

1. use c++03(98) features: template, exceptions, STD library
2. provide copy-constructors: `return my_canvas;`
3. separate data from presentation (TH1F, ...)
4. improve the visual result (e.g. support alpha channel, automatic legend placement)
5. better class hierarchy (e.g. why `TGraph` inherits from `TAttLine`? why `TH1::GetZaxis()`?)
6. proof: support sharing of resources: why declare one histogram for every workers instead of sharing it?
7. better `SetBranchStatus`, read on demand: `TTreeReader`!
8. generalize TSelector (run on batch queues, grid, ...), see EventLoop (see Attila talk), sframe
9. restructure old code: ROOT has one `goto` (ROOT has one `goto` every 900 lines, V8 JavaScript Engine every 84k, minuit every 29 lines, boost every 1000 lines.
10. static polymorphism (`TH1<double>`) ?

What I would like to see in ROOT/2

1. use c++03(98) features: template, exceptions, STD library
2. provide copy-constructors: `return my_canvas;`
3. separate data from presentation (TH1F, ...)
4. improve the visual result (e.g. support alpha channel, automatic legend placement)
5. better class hierarchy (e.g. why `TGraph` inherits from `TAttLine`? why `TH1::GetZaxis()`?)
6. proof: support sharing of resources: why declare one histogram for every workers instead of sharing it?
7. better `SetBranchStatus`, read on demand: `TTreeReader`!
8. generalize TSelector (run on batch queues, grid, ...), see EventLoop (see Attila talk), sframe
9. restructure old code: ROOT has one `goto` (ROOT has one `goto` every 900 lines, V8 JavaScript Engine every 84k, minuit every 29 lines, boost every 1000 lines.
10. static polymorphism (`TH1<double>`) ?
11. better communication of new features (e.g. `TEfficiency`): blog, ..., release note is not enough

ATLAS data model

- RAW data $\rightarrow$ ESD / AOD $\rightarrow$ ROOT ntuple (D3PD)
- data object model inside ESD/AOD (particles are objects: electron.pt, electron.energy, electron.track, ...)
- D3PD is a TTree of (vector of) floats, no object model (el_pt, el_E, el_track_index, track_pt, ...)
- when dumping information from AOD to D3PD some new quantities are computed
- many D3PD flavours, now we are trying to merge someones

(actual) ATLAS analysis model

- very few analyses are made from AODs inside the core software (Athena), most of them uses D3PDs (ntuples) with plain ROOT
- most users uses MakeClass
- someone uses Proof (MakeSelector)
- few people use D3PDReader (from Physics Analysis Tools)
- D3PDReader restores the object model from D3PD (el_pt, el_E, el_track_index, track_pt, ... $\rightarrow$ el.pt, el.E, el.track, ...)
- with D3PDReader many feautures are provided:
  - every variables is a template proxy: it is readed only when it is needed (very efficient), no need of TTree.SetBranchStatus
  - EventLoop interfacing with proof, grid, batch queues, ...

See Attila presentation