Table of Contents

The Ostap tutorials 1.1
Getting started 2.1
- Values with uncertainties 2.1.1
- Simple operations with histograms 2.1.2
- Simple operations with trees 2.1.3
- Persistency 2.1.4

Using RooFit 3.1
- Useful decorations 3.1.1
- PDFs and the basic models 3.1.2
- Compound fit models 3.1.3

Contributing 4.1
Examples of formatting 4.2
Download PDF 5.1

The Ostap tutorials

Ostap is a set of extensions/decorators and utilities over the basic `PyROOT` functionality (python wrapper for `ROOT` framework). These utilities greatly simplify the interactive manipulations with `ROOT` classes through python. The main ingredients of `Ostap` are

- preconfigured ipython script `ostap`, that can be invoked from the command line.
- *decoration* of the basic `ROOT` objects, like histograms, graphs etc.
  - operations and operators
  - iteration, element access, etc
  - extended functionality
- *decoration* of many basic `ROOT.RooFit` objects
- set of new useful fit models, components and operations
- other useful analysis utilities

Getting started

The main ingredients of `Ostap` are

- preconfigured ipython script `ostap`, that can be invoked from the command line.

  ```
  ostap
  ```

  ### Challenge

  Invoke the script with `-h` option to get the whole list of all command line options and keys

  Optionally one can specify the list of python files to be executed before appearance of the interactive command prompt:

  ```
  ostap a.py b.py c.py d.py
  ```

  The list of optional arguments can include also root-files, in this case the files will be opened and their handlers will be available via local list `root_files`

  ```
  ostap a.py b.py c.py d.py file1.root file2.root e.py file3.root
  ```

  Also `ROOT` macros can be specified on the command line

  ```
  ostap a.py b.py c.py d.py file1.root q1.C file2.root q2.C e.py file3.root q4.C
  ```

The script automatically opens `TCanvas` window (unless `--no-canvas` option is specified) with (a little bit modified) LHCb style. It also loads necessary decorators for `ROOT` classes. At last it executes the python scripts and opens root-files, specified as command line arguments.

Values with uncertanties: `ValueWithError`

One of the central object in `ostap` is C++ class `Gaudi::Math::ValutWithError`, accessible in python via shortcut `VE`. This class stands fot r a combination of the value with uncertainties:

```
from Ostap.Core import VE
a = VE( 10 , 10 ) ## the value & squared uncertainty - 'variance'
b = VE( 20 , 20 ) ## the value & squared uncertainty - 'variance'
print "a=%s" % a
print "b=%s" % b
print 'Value    of a is %s'  % a.value()
print 'Effor    of b is %s'  % b.error()
print 'Variance of b is %s'  % b.cov2 ()
```

A lot of math operations are predefined for `VE`-objects.

Challenge

Make a try with all binary operations (`+,-,*,/,**`) for the pair of `VE` objects and combinations of `VE`-objects with numbers, e.g.

```
a + b
a + 1
1 - b
2 ** a
a +=1
b += a
```

Compare the difference for following expresssions:

```
a/a      ## <--- HERE
a/VE(a)  ## <--- HERE
a-a      ## <--- HERE
a-VE(a)  ## <--- HERE
```

Note that for trivial cases the correlations are propertly taken into account

Additionally many math-functions are provided, carefully takes care on uncertainties

```
from LHCbMath.math_ve import *
sin(a)+cos(b)/tanh(b)
atan2(a,b)/log(a)
```

Simple operations with histograms

Historgam content

`Ostap.PyRoUts` module provides two ways to access the histogram content

- by bin index, using operator `[]` : for 1D historgam index is a simple integer number, for 2D and 3D-histograms the bin index is a 2 or 3-element tuple
- using *functional* interface with operator `()` .

  ```
  histo = ...
  print histo[2]    ## print the value/error associated with the 2nd bin
  print histo(2.21) ## print the value/error at x=2.21
  ```

  Note that the result in both cases is of type `VE` , *value+/-uncertainty*, and the interpolation is involved in the second case. The interpolation can be controlled using `interpolation` argument

  ```
  print histo ( 2.1 , interpolation = 0 ) ## no interpolation
  print histo ( 2.1 , interpolation = 1 ) ## linear interpolation
  print histo ( 2.1 , interpolation = 2 ) ## parabolic interpolation
  print histo ( 2.1 , interpolation = 3 ) ## cubic interpolation
  ```

  Similarly for 2D and 3D cases, `interpolation` parameter is 2 or 3-element tuple, e.g. `(2,2)` `(3,2,2)` , `(3,0,0)` , ...

Set bin content

```
histo[1] = VE(10,10)
histo[2] = VE(20,20)
```

Loops over the histogram content:

```
for i in histo :
    print 'Bin# %s, the content%s' % ( i, histo[i] )
for entry in histo.iteritems() :
    print 'item  ', entry
```

The *reversed* iterations are also supported

```
for i in reversed(histo) :
    print 'Bin# %s, the content%s' % ( i, histo[i] )
```

Histogram slicing

The slicing of 1D-historgam can be done easily using native `slice` in python

```
h1 = h[3:8]
```

For 2D and 3D-casss the slicing is less trivial, but still simple

```
histo2D = ...
h1 = histo2D.sliceX ( 1 )
h2 = histo2D.sliceY ( [1,3,5] )
h3 = histo2D.sliceY ( 3 )
h4 = histo2D.sliceY ( [3,4,5] )
```

Operators and operations

A lot of operators and operations are defined for histograms.

```
histo += 1
histo /= 10
histo  = 1      + histo      ## operations with constants
histo  = histo + math.cos    ## operations with functions
histo /= lambda x : 1 + x    ## lambdas are also functions
```

Also binary operations are defined

```
h1 = ...
h2 = ...
h3 = h1 + h2
h4 = h1 / h2
h5 = h1 * h2
h6 = h1 - h2
```

For the binary operations the action is defiened accordinh to the rule

- the type of the result is defined by the first operand (type, and binning)
- for each bin `i` the result is estimated as `a oper b`, where:
  - `oper` stands for corresponding operator (`+,-,*,/,**)`
  - `a = h1[i]` is a value of the first operand at bin `i`
  - `b = h2(x)` , where `x` is a bin-center of bin `i`

More operations

There are many other useful opetations:

- `abs` : apply `abs` function bin-by-bin
- `asym` : equivalent to `(h1-h2)/(h1+h2)` with correct treatment of correlated uncertainties
- `frac` : equivalent to `(h1)/(h1+h2)` with correct treatment of correlated uncertainties
- `average` : make an average of two historgam
- `chi2` : bin-by-bin chi2-tension between two historgams
- ... and many more

Transformations

```
h1 = histo.transform ( lambda x,y : y    ) ## identical transformation (copy)
h2 = histo.transform ( lambda x,y : y**3 ) ## get the third power of the histogram content
h3 = histo.transform ( lambda x,y : y/x  ) ## less trivial functional transformation
```

Math functions

The standard math-functions can be applied to the histoigram (bin-by-bin):

```
from LHCbMath.math_ve import *
h1 = sin ( histo )
h2 = exp ( histo )
h3 = exp ( abs ( histo ) )
...
```

Sampling

There is an easy way to sample the histograms according to their content, e.g. for toy-experiments:

```
h1 = histo.sample() ## make a random histogram with content sampled according to bin+-error in original histo
h2 = histo.sample( accept = lambda s : s > 0 ) ##sample but require that sampled values are positive
```

Smearing/convolution with gaussian

It is very easy to smear 1D histogram according to gaussian resolution

```
h1 = histo.smear ( 0.015 ) ## apply "smearing" with sigma = 0.015
h2 = histo.smear ( sigma = lambda x :  0.1*x ) ## smear using 'running' sigma of 10% resolution
```

Rebin

```
original = ... ## the original historgam to be rebinned
template = ... ## historgams that deifned new binning  scheme
rebin1   = original.rebinNumbers  ( template ) ## compare it!
rebin2   = original.rebinFunction ( template ) ## compare it!
`
```

Note that there are *two* methods for rebin `rebinNumbers` and `rebinFunction` - they depends on the treatment of the histogram.

Challenge

Choose some initial histogram with non-uniform biuning, choose *template* historam with non-uniform binning and compare two methods: `rebinNumbers` and `rebinFunction`.

Integrals

There are several*integral*-like methods for (1D)histograms

- `accumulate` : useful for *numbers*-like histograms, only bin-content inn used for summation (unless the bin is effectively split in case of low/high summation edge does not coinside with bin edges)

  ```
  s = histo.accumulate ()
  s = histo.accumulate ( cut = lambda s :  0.4<=s[1].value()<0.5 )
  s = histo.accumulate ( low  = 1    , high = 14 ) ## accumulate over    1<= ibin <14
  s = histo.accumulate ( xmin = 0.14 , xmax = 14 ) ## accumulate over xmin<= x    <xmax
  ```

- `integrate` : useful for *function*-like histograms, perform integration taking into account bin-width.

  ```
  s = histo.integrate ()
  s = histo.integrate ( cut  = lambda s :  0.4<=s[1].value()<0.5 )
  s = histo.integrate ( lowx = 1    , highx = 14   ) ## integrate over    1<= xbin <14
  s = histo.integrate ( xmin = 0.14 , xmax  = 21.1 ) ## integrate over xmin<= x    <xmax
  ```

- `integral` it transform the histogram into `ROOT.TF1` object and invokes `ROOT.TF1.Integral`

Running sums

and the efficiencies of cuts_

```
h1 = histo.sumv ()        ## increasing order: sum(first,x)
h2 = histo.sumv ( False ) ## decreasing order: sum(x,last )
```

Efficiency of the cut

Such functionality immediately allows to calculate efficiency historgrams using `effic` method:

```
h1 = histo.effic ()        ## efficiency of var<x cut
h2 = histo.effic ( False ) ## efficiency of var>x cut
```

Conversion to `ROOT.TF(1,2,3)`

Scaling

In additon to *trivial* scaling operations `h *= 3` and `h /= 10` there are seevral dedicated methdo for scaling

- `scale` it scales the historgam content to a given sum of *in-range* bins

  ```
  print histo.accumulate()
  histo.scale(10)
  print histo.accumulate()
  ```

  - `rescale_bins` : it allows the treatment of non-uniform histograms as density distributions. Essentially each bin `i` is rescaled according to the rule `h[i] *= a / S` , where `a` is specified factor and `S` is *bin-area*. such type of rescaling is important for historgams with non-uniform binning

Density

There is method `density` that converts the histgram into *density* histogram. The density histogram (being interpreted as *function*) has unit integral. It is different from the simple rescaling for historgams with non-uniform bins.

```
d = histo.density()
```

Statistics

There are many *statistic* functions

- `mean`
- `rms`
- `kurtosis`
- `skewness`
- `moment`
- `centralMoment`
- `nEff` : number of equivalent entries
- `stat` : statistical information about bin-to-bin content: mean, rms, minmax, ... in form of `Gaudi::Math::StatEntity` class

Figure-of-Merit evaluation and cut optimisation

If *figure-of-merit* is natural and equals to *sigma(S)/S* (note that it is equal to *sqrt(S+B)/S*):

```
signal = ...   ## distribition for signal
fom1   = signal.FoM2 () ## FoM for var<x cut
fom2   = signal.FoM2 ( False ) ## FoM for var>x cut
```

Note that no explicit knowledge of background is needed here - it enters indirecltly via the uncertainties in signal determination.

If *figure-of-merit* is defined as *S/sqrt(S+alpha*B)*

```
signal = ...
background = ...
alpha = ...
fom1  = signal.FoM1 ( background , alpha ) ## FoM for var<x cut
fom2  = signal.FoM1  ( background , alpha , False ) ## FoM for var>x cut
```

Solve equations

One can also *solve equations* `h(x) = v`

```
value = 3
solutions = histo.solve ( value )
for x in solutions : print x
```

Conversion to `ROOT.TF(1,2,3)

The conversion of histogram to `ROOT.TF1` objects is straighforward

```
f = histo.tf1()
```

Optionally one can specify `interpolate` flag to define the interpolation rules.

The obtained `TF1` object is defined with three parameters

1. normalization
2. bias
3. scale

It can be used e.g. for visualize interpolated historgam as function or e.g. in `ROOT.TH1.Fit` method for fitting of other historgams

Efficiencies

There are several special cases to get the efficiency-historgams

```
accepted = ... ## historgam with accepted sample
rejected = ... ## historgam with rejected sample
total    = ... ## historgam with total sample

eff1 = accepted/total          ## value is correct, uncertainties are *NOT* correct
eff2 = 1/(1+rejected/accepted) ## everything is correct (binomial)
eff3 = accepted %  total       ## everything is correct (binomial)
eff4 = accepted // total       ## correct binomial, if both histograms are "natural"
```

Binomial efficiencies

In additon to the methods described above, few more sophisticated treatments of *binomial effiiciencies* are provided

```
accepted = ...
total    = ...

eff1 = accepted.        zechEff ( total ) ## valid for all histograms, including sPlot-weighted
eff2 = accepted.       binomEff ( total ) ## only for natural histograms
eff3 = accepted.      wilsonEff ( total ) ## only for natural histograms
eff4 = accepted.agrestiCoullEff ( total ) ## only for natural histograms
```

For *natural* historgams only one can use even more sophisticated methods, that evaluates the interval. Each method returns *graph*, and the graphs can be visuzalised for comparison:

```
accepted = ...
rejected = ...

eff1 = accepted.eff_wald                    ( rejected )
eff2 = accepted.eff_wilson_score            ( rejected )
eff3 = accepted.eff_wilson_score_continuity ( rejected )
eff4 = accepted.eff_arcsin                  ( rejected )
eff5 = accepted.eff_agresti_coull           ( rejected )
eff6 = accepted.eff_jeffreys                ( rejected )
eff7 = accepted.eff_clopper_pearson         ( rejected )
```

All of this functions have an optional argument `interval` that defines the confidence interval, the default value is `interval=0.682689492137086` that corresponds to 1 sigma.

Optimal binning?

It is not a rare case when one needs to find the binbing of the histogram that ensures almost equal bin populations. This task could be solved using `eqaul_bins` method

```
very_fine_binned_histo = ... ## get the fine binned histograms
edges1 = fine_binned.equal_edges ( 10 ) ## try to fing binning with 10 almost equally populated bins
edges2 = fine_binned.equal_edges ( 10 , wmax = 5 ) ## try to fing binning with 10 almost equally populated bins, but avoid bins wider than "wmax"
```

Persistency

`Ostap.ZipShelve`

Ostap offers very nice&efficient way to store the objects in persistent dbase. This persistency is build around `shelve` module and differs in two way

1. the conntent of payload is compressed, using `zlib` module making the data base very compact
   - (optionally) the whole database can ve further `gzip` 'ed using `gzip` module, if the extension `.gz` is provided. It makes data banse even more compact.
2. in addition to the native `dict` interface from `shelve` , more extensiveinterface with more methods is supported.

Create database and write objects to it:

```
a = ...
import Ostap.ZipShelve as DBASE
with DBASE.open ( 'my_dbase.db' ) as db : ## create DBASE
  db.ls()
  db['a'] =  a
  db['histo'] = ROOT.TH1D('h1','',10,0,1)
```

Reading objects from database

```
with DBASE.open ( 'my_dbase.db' , 'read') as db : ## read DBASE
  db.ls()
  b  = db['a']
  h2 = db['histo']
```

One can store in database all *pickable* objects, that means all python objects, all (serializeable) `ROOT` objects. All `C++` objects with `LCG/Reflex/Cint` -dictionaries are also could be stored database. In practice, everything is storable, including complex combination of python&C++ objects, like dictionary of historgams and python classed, inherited from `C++` -base classes.

Plain `ROOT.TFile`

Ostap adds some decorations even for the plain `ROOT.TFile` , making its interface more *pythonic*:

```
rfile = ...
obj   = rfile['A/B/C/myobj']     ## READ  object form the file/directory
rfile['A/B/C/myobj2'] = object2  ## WRITE object to the file/directory
obj  = rfile.A.B.C.myobj              ## another way to access to the object
obj  = rfile.get ( 'A/B/C/q' , None ) ## one more way to get object
for obj in rfile : print obj          ## loop over all objects in file
for key,obj in rfile.iteritems() : print key, obj            ## another loop
for key,obj in rfile.iteritems( ROOT.TH1 ) : print key, obj   ## advanced loop, get only histograms
for k in rfile.keys()     : print k   ## get all keys and loop over them
for k in rfile.iterkeys() : print k   ## loop over all keys in the file
del rfile['A/B']                      ## delete the object from the file
rfile.rm ( 'A/B' )                    ## delete the object from the file
if 'A/MyHisto' in rfile          : print 'OK!' ## check presence of the key
if rfile.has_key ( 'A/MyHisto' ) : print 'OK!' ## check presence of the key
with ROOT.TFile('aa.root') as rfile : rfile.ls() ## context manager protocol
```

`RootOnlyShelve`

The module `Ostap.RootShelve` offers the thin wrapper over `ROOT.TFile` that implement `shelve` -interface. As a resutl one gets a ligth database build a top of underlying `ROOT.TFile` , where `ROOT` -objects could be stored:

```python
from Ostap.RootShelve import RooOnlyShelf
db = RooOnlyShelf('mydb.root','c')
h1 = ...
db ['histogram'] = h1
db.ls()
```

`RootShelve`

The module `Ostap.RootShelve` offers also more sophisticated wrapper over `ROOT.TFile` that also implements `shelve`-interface and able to store ROOT and any other *pickable* objects

```python
from Ostap.RootShelve import RooShelf
db = RooShelf('mydb.root','c')
h1 = ...
db ['histogram'] = h1
db ['histogramlist'] = h1,h2,h3
db.ls()
```

In details ...

For non-`ROOT` objects, database actually stores them as `ROOT::TString` objects carrying their pickle representation with on-flight removal/substitutions of some magic symbol sequences, since `ROOT::TString` is not a real `BLOB`.

Decorations

Ostap *decorates* many `ROOT.RooFit` classes, adding more convinient methods to them.

`RooArgList` and `RooArgSet`

All these classes have got set of additional python-like methods for iteration, extension, addition, elemtn access checking the content etc... Also several methods to provide more coherent interfaces (e.g. `add` vs `Add`) are added.

```
# Ostap.PyRoUts             INFO    Zillions of decorations for ROOT/RooFit objects
Lengths are 2 2
'a' : ( 0 +- 0 )
'b' : ( -10 +- 0 )
'b' : ( -10 +- 0 )
'c' : ( 1 +- 0 )
 a in l ? True True
 b in l ? True True
 c in l ? False False
 a in l ? False False
 b in l ? True True
 c in l ? True True
'a' : ( 0 +- 0 )  'b' : ( -10 +- 0 )
'b' : ( -10 +- 0 )  'c' : ( 1 +- 0 )
l1+l1 :    ['a:0.0', 'b:-10.0', 'a:0.0', 'b:-10.0']
l1+l2 :    ['a:0.0', 'b:-10.0', 'b:-10.0', 'c:1.0']
l2+l2 :    ('b:-10.0', 'c:1.0')
l2+l1 :    ('b:-10.0', 'c:1.0', 'a:0.0')
l1+c :    ['a:0.0', 'b:-10.0', 'c:1.0']
l2+c :    ('b:-10.0', 'c:1.0')
l1+d :    ['a:0.0', 'b:-10.0', 'd:-1.0']
l2+d :    ('b:-10.0', 'c:1.0', 'd:-1.0')
c+l1 :    ['a:0.0', 'b:-10.0', 'c:1.0']
c+l2 :    ('b:-10.0', 'c:1.0')
d+l1 :    ['a:0.0', 'b:-10.0', 'd:-1.0']
d+l2 :    ('b:-10.0', 'c:1.0', 'd:-1.0')
```

output.txt hosted with ❤ by **GitHub** **view raw**

```
import ROOT
import Ostap.PyRoUts

a  = ROOT.RooRealVar ('a','a',-10,10)
b  = ROOT.RooRealVar ('b','b',-10)
c  = ROOT.RooConstVar('c','c',  1)
d  = ROOT.RooConstVar('d','d', -1)

```

```python
l1 = ROOT.RooArgList    ( a , b )
l2 = ROOT.RooArgSet     ( b , c )

print 'Lengths are %s %s ' % ( len ( l1 ) , len( l2 ) )

for i in l1 : print i
for i in l2 : print i

for l in  ( l1 , l2 ) :
    print ' a in l ? %s %s '  % ( a in l , 'a' in l )
    print ' b in l ? %s %s '  % ( b in l , 'b' in l )
    print ' c in l ? %s %s '  % ( c in l , 'c' in l )


print l1[0]   , l1[1]
print l2['b'] , l2['c']

print 'l1+l1 :    %s'  % ( l1 + l1 )
print 'l1+l2 :    %s'  % ( l1 + l2 )
print 'l2+l2 :    %s'  % ( l2 + l2 )
print 'l2+l1 :    %s'  % ( l2 + l1 )

print 'l1+c :    %s'  % ( l1 + c )
print 'l2+c :    %s'  % ( l2 + c )
print 'l1+d :    %s'  % ( l1 + d )
print 'l2+d :    %s'  % ( l2 + d )

print 'c+l1 :    %s'  % ( c + l1 )
print 'c+l2 :    %s'  % ( c + l2 )
print 'd+l1 :    %s'  % ( d + l1 )
print 'd+l2 :    %s'  % ( d + l2 )
```

roofit_lists.py hosted with ❤ by **GitHub** **view raw**

RooAbsData and RooDataSet

These methods also have got the extended interface with many useful methods and operators, like e.g. concatenation of datasets `a+b` and merging them `a*c` .

`RooDataSet` class also has go many methods, that are similar to those of `ROOT.TTree` , in particular `project` and `draw` :

```python
dataset = ...
dataset.draw('mass','pt>1')
histo   = ...
dataset.project ( histo , 'mass', 'pt>1' )
```

Many other methonds like `statVar` , `sumVar` , `statCov` , `vminmax` are also the same as for `ROOT.TTree` , see above.

```
s1    = dataset.statVar ('eff')
s2    = dataset.sumVar  ('eff')
r     = dataset.statCov ('eff','pt')
mn,mx = dataset.vminmax ('eff')
```

RooFitResult

The class `RooFitResult` get many decorations that allow to access fit results

```
result = ...
par1 = result.params()  ## get all floating parameters
par2 = result.params( float_only = False ) ## all parameters
a,v  = result.param ( 'a' )      ## par by name
a,v  = result.param (  a  )      ## par by RooFit object itself
p    = result.a                 ## par as attribute
for par in result :   print par                ## iteration
for name,par in result.iteritems() : print par ## iteration
print result.cov  ( 'a' , 'b' )   ## get the covariance submatrix
print result.corr ( 'a' , 'b'  )  ## get the correlation coefficient
```

Also the simple math with fiting parameters is supported

```
result = ...
s = result.sum       ('S','B' ) ## S+B
d = result.divide    ('S','B' ) ## S/B
s = result.subtract  ('B','B1') ## B-B1
m = result.multiply  ('A','B' ) ## A*B
f = result.fraction  ('S','B' ) ## S/(S+B)
```

RooRealVar & friends

Few simple operations are added to simplify the calculations with `RooRealVar` objects:

```
x = ROOT.RooRealVar( ... )
x + 10
x - 10
x * 10
x / 10
10 + x
10 - x
10 * x
10 / x
x += 2
x -= 2
x *= 2
x /= 2
x ** 3
```

PDFs and the basic models

Ostap provides set of useful wrapper and helper class that drastically simplify the construction and manipulations with `RooAbsPdf` -objects.

E.g. consider the simplest case - creation of the Gaussian PDF using the standard way the standard way:

```
x     = ROOT.RooRealVar ('x'    ,'x'   ,2,3)
mean  = ROOT.RooRealVar ('mean' ,'mean' ,3.100,3.080,3.120)
sigma = ROOT.RooRealVar ('sigma','sigma',0.015,0.010,0.025)
pdf   = ROOT.RooGaussian('Gauss','Gaussian', x , mean , sigma )
```

In Ostap it can be done in a bit simpler way

```
gauss = Gauss_pdf  ( 'Gauss' ,
                        xvar  = ( 2 , 3 ) ,
                      mean  = ( 3.100 , 3.080 , 3.120 ) ,
                      sigma = ( 0.015 , 0.010 , 0.025 ) )
gauss.draw() ## and one can immediately visualize the model
```

How to define parameter?

There are may ways to define parameter

1. One can use the existing `RooAbsReal` object, e.g. `RooRealVar` or `RooConstVar` :

```
mean  = ROOT.RooRealVar ('mean' ,'mean' ,3.100,3.080,3.120)
gauss = Gauss_pdf  ( 'Gauss' ,
                  xvar  = ( 2 , 3 ) ,
                mean  = mean , ## <--- HERE
                sigma = ( 0.015 , 0.010 , 0.025 ) )
```

   i. One can use 2 or 3-element tuple `(minval,maxval)` or `(value, minval,maxval)` or plain single number. In this case the variable of the type `RooRealVar` will be automatically created using this specification

```
gauss = Gauss_pdf  ( 'Gauss' ,
               xvar  = ( 2 , 3 ) ,                 ## <-- HERE
             mean  = ( 3.100 , 3.080 , 3.120 ) , ## <-- HERE
             sigma =   0.015 )                   ## <-- HERE
```

For all models, all known parameter are accessible (and documented) as python *property*

```
gauss = ...
help(gauss.xvar)
print gauss.sigma
help(gauss.mean)
```

There are many predefined models, accesible via `Ostap.FitModels` module:

```
import Ostap.FitModels as Models
help(Models)
```

Base class `PDF`

All Ostap-based fit models and PDFs (directy or indirectly) inherit from python base class `PDF`, that provides great additional functionality, in particular the methods `fitTo` and `draw` that simplfy the fitting procedure itself and visualzation of the results:

The method `fitTo`

```
gauss   = Gauss_pdf ( ... )
dataset = ....
result , frame = gauss.fitTo ( dataset , silent = True , reFit = 2 )
print 'FitResults: %s' % result
```

All the native `RooFit` *commands* can be specified as optional arguments, as well as many commands specific for Ostap, e.g. `reFit=2` above means *in case of fit failure, try to refit it (up to 2 times)*, and the meaning of `silent=True` is obvious.

The method `draw`

```
gauss   = Gauss_pdf ( ... )
dataset = ....
result , frame = gauss.fitTo ( dataset , silent = True , reFit = 2 )
print 'FitResults: %s' % result
frame = gauss.draw ( dataset , nbins = 100 )
```

Fitting and *vizualisation* can be combined:

```
gauss   = Gauss_pdf ( ... )
dataset = ....
result , frame = gauss.fitTo ( dataset , draw  = True , nbins = 100 ) ## draw it after the fit
```

Access to the underlying `RooAbsPdf` object

The access to the underlying bare `RooAbsPdf` -object can be done (if needed) via the propety `pdf`

```
gauss = Gauss_pdf ( ... )
root_pdf  = gauss.pdf
```

Other methods

`PDF` class is equipped with many other useful methods:

- `fitHisto` : The method `fitTo` can be *blindly* applied not only to `RooDataSet` -objects, but also to the histograms:

  ```
  histo = ...
  r, f   = gauss.fitTo ( histo , draw = True )
  ```

  However the dedicated method `fitHisto` sometimes could be more usefu

  ```
  histo = ...
  gauss.fitHisto ( histo , draw = True )
  ```

- `draw_nll` : vizualize NLL-scans and LL-profiles

  ```
  r , f = gauss.fitTo ( dataset , draw = False )
  nll     , f1 = gauss.draw_nll ( 'sigma' ,  dataset )                  ## NLL
  profile , f2 = gauss.draw_nll ( 'sigma' ,  dataset , profile = True ) ## PROFILE
  ```

  - `generate` : tiny but useful wrapper for `RooAbsPdf::generate`
  - `minmax` : mane the estimates for minimal and maximal values for the PDF. For some models it is done analytically or semianalitycally, for remainnig models it is doen using random shoots.

```
mn,mx = gauss.minmax( 500000 )
```

- `__call__` : it allows to use `PDF` as simple *function*

```
gauss = ...
print gauss( 3.090 ), gauss( 3.100 ), gauss( 3.110 )
```

- Several *statistical* functions. For some models analytical orsemianalitycal calculations are used, for remnig models numerical estimations are performed using `scipy`
- `rms` : *rms* for the distribution
- `fwhm` : *full width at half maximum*
- `fwhm` : *full width at half maximum*
- `moment` : the *moment* of the distribution
- `central_moment` : the *central moment* of the distribution
- `skewness` : *skewness* for the distribution
- `kurtosis` : *kurtosis* for the distribution
- `mode` : the *mode* for the distribution
- `median` : *median* value for the distribution
- `get_mean` : *mean* value for the distribution
- `cl_symm` : *symmetric confidence interval*
- `cl_asymm` : *asymmetric confidence interval*
- `quantile` : *quantile* value for the distribution
- `integral` : *integral* for the distribution
- `derivative` : *derivative* of the PDF at the given point

Convolution

...

Generic Wrapper *Generic1D_pdf*

The bare `RooAbsPdf` could be easily converted to Ostap-form using the generic wrapper `Generic1D_pdf` :

```
bare = ROOT.RooGaussian('Gauss','Gaussian', x , mean , sigma )
gauss = Generic1D_pdf ( pdf = bare , xvar = x )
gauss.draw() ## one can immediately use the full power of Ostap-PDF
```

In a similar way there are generic wrappers for `2D` and `3D` -models:

```
bare2D = ...
bare3D = ...
ostap_2d =  Generic2D_pdf ( pdf = bare2D , xvar = x ,  xvar = y )
ostap_3d =  Generic2D_pdf ( pdf = bare3D , xvar = x ,  xvar = y , zvar = z )
```

`2D` and `3D` -cases

For `2D` and `3D` cases there are base classes `PDF2` and `PDF3` that in turn inhetic from `PDF` and gets all the nice functionality. Of course several new method specific for `2D` and `3D` -cases are added and th ebehaviosu of some `1D` -specific methods is fixed.

Compound fit models

`1D` -case

Ostap offers a very easy way to build the compound fit models from the individual components. E.g.the case of the trivial fit model that consists of one signal and one background components:

```
signal      = ...
background  = ...
model    = Fit1D ( signal = signal , background = backround ) ## <-- HERE!
dataset  = ...
result , frame = model.fitTo ( dataset , draw = True ) ## fit and vizualize
```

The fit model can contains several *signal* and *backround* components, and also *other* components :

```
model    = Fit1D ( signal = signal ,
                   background       = backround ,
                   othersignals     = [ ... ]   ,
                   otherbackgrounds = [ ... ]   ,
                   others           = [ ... ]   )
```

In this case several *signal*, *backgrounds* and/or *others* components can be combined into single *signal*, *backround* and/or *others* components:

```
model    = Fit1D ( combine_signals     = True ,
                   combine_backgrounds = True ,
                   combine_others      = True , ... )
```

In practice it is very convinients approach is several signal/background/other componens are specified.

On default *extended* `RooAddPdf' fit model is created, however , one can force *non-extended* model:

```
model    = Fit1D ( extended = False , ... )
```

In this case one can also instruct the class `Fit1D` to create *recursive* (default) or *non-recursive* fit fractions:

```
model    = Fit1D ( extended = False , recursive = False , ... )
```

All components (*signal/background/others*) can be specified as Ostap-based models. Also one can provide them in a form of bare `RooAbsPdf` , but for this case one needs to provide also `xvar` -variable

```
mass  = ROOT.RooRealVar('mass','mass',2,3)
gauss = ROOT.RooGaussian( 'Gauss', 'Gauss', mass , ... )
model = Fit1D( signal = gauss , xvar = mass , ... )
```

For *background* components there is also an alternative way to specify it:

- `None` : `RooPolynomial` of zero degree (uniform distribution) will be created and used as *background* component
    - Attention: `background=None` does *not* imply the absence of background component
- negative integer `n` : Ostap model `PolyPos_pdf` will be created and used as *background* component. This model corresponds to the *positive* polynomial of degree `-n` . The polinomial is constrained to be non-negative for the whole considered interval of `xvar` . This constraint allows rather robust and stable fits, especialy for the low-statistics case.
- non-negative integer `n` : Ostap model `Bkg_pdf` , that is a product of the exponential function and the *positive* polynomial of degree `n` will be created and used as *background* component. Note:
    - The `background=0` case corresponds to simple exponential backtround

    - Since the polynomial is constrained to be *non-negative* this PDF is very stable and robust, especually for the low-statistic case,
- as `RooAbsReal` object, in this case it is interpreted as the exponental slope

Access to the model componens

The indiviaul components can be accessed using python *properties*

```
gaudd = Gauss_pdf ( ...
model = Fit1D ( signal = gauss , ... )
print model.signal.sigma  ##   get sigma of Gauss
print model.signal.mean   ##   get mean  of Gauss
```

Fit parameters

The paramettes of the created `RooAddPdf` can be accessed via python properties, e.g. for *extended* fits:

```
gaudd = Gauss_pdf ( ...
model = Fit1D ( signal = gauss , ... )
print 'signal yield(s):' , model.S
print 'background(s):'   , model.B
print 'others: '         , model.C
model.S = 100   ## set value of signal component to be 100 events
model.B.fix(50) ## fix the yield of the background component at 50  events
model.draw()
```

Depending on the number of corresponsing componens and flags `combine_signals` , `combine_backgrounds` , `combine_others` these properties can be *scalar* values or arrays/tuples.

For `combine_signal=True` , `combine_backgrounds =True` , `combine_others=True` cases oen also gets properrties `fS` , `fB` and `fC` that corresponds to the fractions of individual *signal/backgroud/others* components for the compound signal/ *signal/backgroud/others*.

For *non-extended* fits, the main parameters are *fractions*:

```
gaudd = Gauss_pdf ( ...
model = Fit1D ( signal = gauss , ... , extended = False )
...
print 'fractions:' , model.F
```

All the ways to deal with `Fit1D` objects are illustrated here:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# =============================================================================
## @file TestComponents.py
#
#  tests for various multicomponents models
#
#  @author Vanya BELYAEV Ivan.Belyaeve@itep.ru
#  @date 2014-05-11
# =============================================================================
"""Tests for various multicomponent models
"""
# =============================================================================
__version__ = "$Revision:"
__author__  = "Vanya BELYAEV Ivan.Belyaev@itep.ru"
__date__    = "2014-05-10"
```

```
__all__     = ()  ## nothing to be imported
# =============================================================================
import ROOT, random
from   Ostap.PyRoUts import *
from   Ostap.Utils   import rooSilent
# =============================================================================
# logging
# =============================================================================
from AnalysisPython.Logger import getLogger
if '__main__' == __name__  or '__builtin__' == __name__ :
    logger = getLogger ( 'Ostap.TestComponents' )
else :
    logger = getLogger ( __name__ )
# =============================================================================
logger.info ( 'Test for multi-component models from Analysis/Ostap')
# =============================================================================
## make simple test mass
mass    = ROOT.RooRealVar ( 'test_mass' , 'Some test mass' , 0 , 10 )

## book very simple data set
varset  = ROOT.RooArgSet  ( mass )
dataset = ROOT.RooDataSet ( dsID() , 'Test Data set-0' , varset )

mmin, mmax = mass.minmax()

### fill it
m1 = VE(3,0.300**2)
m2 = VE(5,0.200**2)
m3 = VE(7,0.100**2)

for i in xrange(0,5000) :
    for m in (m1,m2,m3) :
        mass.value = m.gauss ()
        dataset.add ( varset    )

for i in xrange(0,5000) :
    mass.value = random.uniform ( *mass.minmax() )
    dataset.add ( varset   )

logger.info ('Dataset: %s' % dataset )
import Ostap.FitModels as Models

signal_1 = Models.Gauss_pdf ( 'G1'  , xvar = mass , mean = m1.value() , sigma = m1.error() )
signal_2 = Models.Gauss_pdf ( 'G2'  , xvar = mass , mean = m2.value() , sigma = m2.error() )
signal_3 = Models.Gauss_pdf ( 'G3'  , xvar = mass , mean = m3.value() , sigma = m3.error() )

```

```
wide_1   = Models.Gauss_pdf ( 'GW1' , xvar = mass , mean = 1.0 , sigma = 2 )
wide_2   = Models.Gauss_pdf ( 'GW2' , xvar = mass , mean = 9.0 , sigma = 3 )

narrow_1 = Models.Gauss_pdf ( 'GN1' , xvar = mass , mean = 4.0 , sigma = 1 )
narrow_2 = Models.Gauss_pdf ( 'GN2' , xvar = mass , mean = 6.0 , sigma = 1 )

## ==============================================================================
logger.info ( 'Test the extended fit with many components' )
model_ext1 = Models.Fit1D (
    name              = 'EXT1'     ,
    signal            =   signal_1 ,
    othersignals      = [ signal_2 , signal_3 ] ,
    background        =   wide_1   ,
    otherbackgrounds = [ wide_2 ] ,
    others            = [ narrow_1 , narrow_2 ] ,
    )
model_ext1.S[0].value = 5000
model_ext1.S[1].value = 5000
model_ext1.S[2].value = 5000

model_ext1.B[0].value = 1700
model_ext1.B[1].value = 2300

model_ext1.C[0].value =  500
model_ext1.C[1].value =  400

r, f = model_ext1.fitTo ( dataset , draw = False , silent = True )
r, f = model_ext1.fitTo ( dataset , draw = False , silent = True )
logger.info ( 'Model %s Fit results:\n#%s ' % ( model_ext1.name , r ) )
print 'Signals               [S]:' , model_ext1.S
print 'Backgrounds           [B]:' , model_ext1.B
print 'Components            [C]:' , model_ext1.C
print 'Fractions             [F]:' , model_ext1.F
print 'Signal fractions     [fS]:' , model_ext1.fS
print 'Background fractions [fB]:' , model_ext1.fB
print 'Component fractions  [fC]:' , model_ext1.fC
print 'Yields           [yields]:' , model_ext1.yields
print 'Fractions     [fractions]:' , model_ext1.fractions


## ==============================================================================
logger.info ( 'Test the extended fit with compound components' )
model_ext2 = Models.Fit1D (
    name                 = 'EXT2'     ,
    signal               =   signal_1 ,
    othersignals         = [ signal_2 , signal_3 ] ,
```

```
    background          =   wide_1   ,
    otherbackgrounds    = [ wide_2 ] ,
    others              = [ narrow_1 , narrow_2 ] ,
    #
    combine_signals     = True   ,
    combine_backgrounds = True   ,
    combine_others      = True   ,
    )
model_ext2.S  = 5000
model_ext2.B  = 4200
model_ext2.C  =  700

model_ext2.fS[0].value = 0.33
model_ext2.fS[1].value = 0.50

model_ext2.fB[0].value = 0.40
model_ext2.fC[0].value = 0.60

r, f = model_ext2.fitTo ( dataset , draw = False , silent = True )
r, f = model_ext2.fitTo ( dataset , draw = False , silent = True )
logger.info ( 'Model %s Fit results:\n#%s ' % ( model_ext2.name , r ) )
print 'Signals              [S]:' , model_ext2.S
print 'Backgrounds          [B]:' , model_ext2.B
print 'Components           [C]:' , model_ext2.C
print 'Fractions            [F]:' , model_ext2.F
print 'Signal fractions    [fS]:' , model_ext2.fS
print 'Background fractions [fB]:' , model_ext2.fB
print 'Component fractions  [fC]:' , model_ext2.fC
print 'Yields          [yields]:' , model_ext2.yields
print 'Fractions    [fractions]:' , model_ext2.fractions

## ============================================================================
logger.info ( 'Test non-extended fit with all components, non-recursive' )
model_ne1 = Models.Fit1D (
    name                = 'NE1'                   ,
    signal              =   signal_1              ,
    othersignals        = [ signal_2 , signal_3 ] ,
    background          =   wide_1   ,
    otherbackgrounds    = [ wide_2 ] ,
    others              = [ narrow_1 , narrow_2 ] ,
    ##
    extended            = False ,
    recursive           = False
    )

model_ne1.F[0].value = 0.25
```

```
model_ne1.F[1].value = 0.25
model_ne1.F[2].value = 0.25
model_ne1.F[3].value = 0.08
model_ne1.F[4].value = 0.12
model_ne1.F[5].value = 0.05

r, f = model_ne1.fitTo ( dataset , draw = False , silent = True )
r, f = model_ne1.fitTo ( dataset , draw = False , silent = True )
logger.info ( 'Model %s Fit results:\n#%s ' % ( model_ne1.name , r ) )
print 'Signals               [S]:' , model_ne1.S
print 'Backgrounds           [B]:' , model_ne1.B
print 'Components            [C]:' , model_ne1.C
print 'Fractions             [F]:' , model_ne1.F
print 'Signal fractions     [fS]:' , model_ne1.fS
print 'Background fractions [fB]:' , model_ne1.fB
print 'Component fractions  [fC]:' , model_ne1.fC
print 'Yields           [yields]:' , model_ne1.yields
print 'Fractions     [fractions]:' , model_ne1.fractions

## ============================================================================
logger.info ( 'Test non-extended fit with all components, recursive' )
model_ne2 = Models.Fit1D (
    name                 = 'NE2'                   ,
    signal               =   signal_1              ,
    othersignals         = [ signal_2 , signal_3 ] ,
    background           =   wide_1    ,
    otherbackgrounds     = [ wide_2 ] ,
    others               = [ narrow_1 , narrow_2 ] ,
    ##
    extended             = False ,
    recursive            = True  ,
    )

model_ne2.F[0].value = 0.25
model_ne2.F[1].value = 0.33
model_ne2.F[2].value = 0.50
model_ne2.F[3].value = 0.37
model_ne2.F[4].value = 0.74
model_ne2.F[5].value = 0.50

r, f = model_ne2.fitTo ( dataset , draw = False , silent = True )
r, f = model_ne2.fitTo ( dataset , draw = False , silent = True )
logger.info ( 'Model %s Fit results:\n#%s ' % ( model_ne2.name , r ) )
print 'Signals               [S]:' , model_ne2.S
print 'Backgrounds           [B]:' , model_ne2.B
print 'Components            [C]:' , model_ne2.C
```

```
print 'Fractions             [F]:' , model_ne2.F
print 'Signal fractions     [fS]:' , model_ne2.fS
print 'Background fractions [fB]:' , model_ne2.fB
print 'Component fractions  [fC]:' , model_ne2.fC
print 'Yields           [yields]:' , model_ne2.yields
print 'Fractions     [fractions]:' , model_ne2.fractions


## ================================================================================
logger.info ( 'Test non-extended fit with compound components, non-recursive' )
model_ne3 = Models.Fit1D (
    name                = 'NE2'                   ,
    signal              =   signal_1              ,
    othersignals        = [ signal_2 , signal_3 ] ,
    background          =   wide_1    ,
    otherbackgrounds    = [ wide_2 ] ,
    others              = [ narrow_1 , narrow_2 ] ,
    ##
    combine_signals     = True    ,
    combine_backgrounds = True    ,
    combine_others      = True    ,
    ##
    extended            = False   ,
    recursive           = False
    )

model_ne3.F[0].value  = 0.75
model_ne3.F[1].value  = 0.30

model_ne3.fS[0].value = 0.33
model_ne3.fS[1].value = 0.50

model_ne3.fB[0].value = 0.41
model_ne3.fC[0].value = 0.58

r, f = model_ne3.fitTo ( dataset , draw = False , silent = True )
r, f = model_ne3.fitTo ( dataset , draw = False , silent = True )
logger.info ( 'Model %s Fit results:\n#%s ' % ( model_ne3.name , r ) )
print 'Signals               [S]:' , model_ne3.S
print 'Backgrounds           [B]:' , model_ne3.B
print 'Components            [C]:' , model_ne3.C
print 'Fractions             [F]:' , model_ne3.F
print 'Signal fractions     [fS]:' , model_ne3.fS
print 'Background fractions [fB]:' , model_ne3.fB
print 'Component fractions  [fC]:' , model_ne3.fC
print 'Yields           [yields]:' , model_ne3.yields
```

```
print 'Fractions     [fractions]:' , model_ne3.fractions


## =============================================================================
logger.info ( 'Test non-extended fit with compound components, recursive' )
model_ne4 = Models.Fit1D (
    name                = 'NE4'                  ,
    signal              =   signal_1             ,
    othersignals        = [ signal_2 , signal_3 ] ,
    background          =   wide_1    ,
    otherbackgrounds    = [ wide_2 ] ,
    others              = [ narrow_1 , narrow_2 ] ,
    ##
    combine_signals     = True    ,
    combine_backgrounds = True    ,
    combine_others      = True    ,
    ##
    extended            = False   ,
    recursive           = True
    )

model_ne4.F[0].value  = 0.75
model_ne4.F[1].value  = 0.80

model_ne4.fS[0].value = 0.33
model_ne4.fS[1].value = 0.50

model_ne4.fB[0].value = 0.41
model_ne4.fC[0].value = 0.50

r, f = model_ne4.fitTo ( dataset , draw = False , silent = True )
r, f = model_ne4.fitTo ( dataset , draw = False , silent = True )
logger.info ( 'Model %s Fit results:\n#%s ' % ( model_ne4.name , r ) )
print 'Signals               [S]:' , model_ne4.S
print 'Backgrounds           [B]:' , model_ne4.B
print 'Components            [C]:' , model_ne4.C
print 'Fractions             [F]:' , model_ne4.F
print 'Signal fractions     [fS]:' , model_ne4.fS
print 'Background fractions [fB]:' , model_ne4.fB
print 'Component fractions  [fC]:' , model_ne4.fC
print 'Yields           [yields]:' , model_ne4.yields
print 'Fractions     [fractions]:' , model_ne4.fractions


# ==============================================================================
# The END
```

```
293    # ==========================================================================
```

components.py hosted with ❤ by GitHub view raw

The corresponding output can be inspected here

`2D` -case

Generic `2D` -case

Symmetric `2D` -case

`3D` -case

Generic `3D` -case

Symmetric `3D` -case

Mixed-symmetry `3D` -case

Contributing

ostap-tutorials is an open source project, and we welcome contributions of all kinds:

- New lessons;
- Fixes to existing material;
- Bug reports; and
- Reviews of proposed changes.

By contributing, you are agreeing that we may redistribute your work under these licenses. You also agree to abide by our contributor code of conduct.

Getting Started

1. We use the fork and pull model to manage changes. More information about forking a repository and making a Pull Request.
2. To build the lessons please install the dependencies.
3. For our lessons, you should branch from and submit pull requests against the `master` branch.
4. When editing lesson pages, you need only commit changes to the Markdown source files.
5. If you're looking for things to work on, please see the list of issues for this repository. Comments on issues and reviews of pull requests are equally welcome.

Dependencies

To build the lessons locally, install the following:

1. Gitbook

Install the Gitbook plugins:

```
$ gitbook install
```

Then (from the ostap-tutorials directory) build the pages and start a web server to host them:

```
$ gitbook serve
```

You can see your local version by using a web-browser to navigate to `http://localhost:4000` or wherever it says it's serving the book.

The title

Learning Objectives

- The starterkit lessons all start with objectives about the lesson
- Objective 2 with some *formatted* **text** ~~like~~ `this`

Basic formatting

You can make **bold**, *italic* and ~~strikethrough~~ text. Add relative links like this one and absolute links in a couple of different ways.

Have bulleted lists:

- Point 1
- Point 2
    - Sub point
        - Sub point
    - Sub point
- Point 2

Use numbered lists:

1. First
2. Second
    i. Second first
        i. Second first first
    ii. Second second
3. Third

LaTeX

You can use inline LaTeX maths such as talking about the decay $$D^{* +} \rightarrow \left(D^0 \rightarrow K^{-} \pi^{+} \right)$$.

Code highlighting

And have small lines of code inline like saying `print("Hello world")` or have multiple lines with syntax highlighting for python:

```
import sys

def stderr_print(string):
    sys.stderr.write(string)

stderr_print("Hello world")
```

bash:

```
lb-run Bender/latest $SHELL
dst_dump -f -n 100 my_file.dst 2>&1 | tee log.log
```

and more!

Callouts

Prequisites

- Prequisite 1
- Prequisite 2

Objectives

- Objective 1
- Objective 2

Challenge

Set a challenge here, and the solution will remain hidden until it's clicked

- How to print?

Solution

The answer is:

```
print("Hello world")
```

Extra details that are hidden by default

Some extra details

Keypoints

- Summary point 1

- Summary point 2

Quotes

> This was said by someone

Tables

Simple tables are possible

First Header	Second Header
Content from cell 1	Content from cell 2
Content in the first column	Content in the second column

Images

Section types

This is a section

Subsections

And a subsection

Subsubsections

And a subsubsection