Table of Contents

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 indireclty 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 bin
s 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:

```
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

```
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` .

More on Histograms

- Histogram parameterization

Histogram parameterization

Often one needs to parameterize the historgam in terms of some predefined function or expansion - e.g. parameterize the efficiency. Ostap offers a wide range of embedded parameterization

- in terms of *Bernstein polynomials*
  - simple *Bernstein sum*, aka *Bezier sum*
  - *even Bernstein sum*, such as `f(x)=f(2*x0-x)` , where `x0=0.5*(xmin+xmax)`
  - non-negative *Bernstein sum*
  - non-negative monothonic *Bernstein sum*
  - non-negative monothonic convex or concave *Bernstein sum*
  - non-negative convex or concave *Bernstein sum*
- in term of *Legendre polynomials*
- in term of *Chebyshev polynomials*
- in terms of *Fourier series*
- in terms of *Fourier cosine series*
- in terms of *Basic splines*
  - non-negative *B-spline*
  - non-negative monothonic *B-spline*
  - non-negative monothonic convex or concave *B-spline*
  - non-negative convex or concave *B-spline*

From technical side, there are three branches of methods

- methods that uses only histogram values:
  - these are safe, robust but they ignore the uncertainties
- methods that relies on `ROOT.THF1.Fit`
  - typically not very good CPU performance
  - sometimes fragile
- methods that relies on `RooFit`
  - often the best series of methods

Simple parameterization

This group of methods allows to make easy and robust histogram parameterization, ignooring histogram unncertainties

```
histo  = ...
b1 = histo.bernstein_sum     (  6 ) ## parameterize as degree-6 Bernstein sum
b2 = histo.bernsteineven_sum (  6 ) ## parameterize as degree-6 Bernstein "even"-sum
l  = histo.legendre_sum      (  6 ) ## parameterize as degree-6 Legendre sum
ch = histo.chebyshev_sum     (  6 ) ## parameterize as degree-6 Chebyshev sum
f  = histo.fourier_sum       ( 12 ) ## parameterize as order-12 Fourier sum
c  = histo.cosine_sum        ( 12 ) ## parameterize as order-12 Fourier Cosine sum
```

`ROOT.TH1.Fit` *-based parameterizations*

These methods typically have not very good CPU performance, and sometiems are fragile, but they allow more accurate treatment of parameteriztaions, in particular them takes into account the uncertainties in the historgam.

```
histo  = ...
b1  = histo.bernstein     (  6 ) ## parameterize as degree-6 Bernstein sum
b2  = histo.bernsteineven (  6 ) ## parameterize as degree-6 Bernstein "even"-sum
l   = histo.legendre      (  6 ) ## parameterize as degree-6 Legendre sum
ch  = histo.chebyshev     (  6 ) ## parameterize as degree-6 Chebyshev sum
f   = histo.fourier       ( 12 ) ## parameterize as order-12 Fourier sum
c   = histo.cosine        ( 12 ) ## parameterize as order-12 Fourier Cosine sum
m   = histo.polynomial    (  6 ) ## parameterize as simple degree-6 monomial sum
p1  = histo.positive      (  6 ) ## parameterize as degree-6 non-negative Bernstein sum
p2  = histo.positiveeven  (  6 ) ## parameterize as degree-6 non-negative even Bernstein sum
m1  = histo.monothonic    ( 6 , increasing = False ) ## parameterize as degree-6 non-negative decreasing Bernstein sum
m2  = histo.monothonic    ( 6 , increasing = True  ) ## parameterize as degree-6 non-negative increasing Bernstein sum
c1  = histo.convex        ( 6 , increasing = False , convex = True  ) ## parameterize as degree-6 non-negative decreasing conv
ex  Bernstein sum
c2  = histo.convex        ( 6 , increasing = False , convex = False ) ## parameterize as degree-6 non-negative decreasing conc
ave Bernstein sum
c3  = histo.convex        ( 6 , increasing = True  , convex = True  ) ## parameterize as degree-6 non-negative increasing conv
ex  Bernstein sum
c4  = histo.convex        ( 6 , increasing = True  , convex = False ) ## parameterize as degree-6 non-negative increasing conc
ave Bernstein sum
cc1 = histo.convexpoly    ( 6 ) #  parameterize as degree-6 non-negative convex  Bernstein sum
cc2 = histo.concavepoly   ( 6 ) #  parameterize as degree-6 non-negative concave Bernstein sum
```

Various types of *splines* are also provided

```
s1 = histo.bSpline ( degree=3 , knots = 2 ) ## parameterize as 3d order spline with 2 inner (uniform) knots
s2 = histo.bSpline ( degree=2 , knots = [0.1,0.4,0.8,0.9] ) ## parameterize as 3d order spline with 4 inner (non-uniform) knot
s
```

and similarly for

- non-negative spline `pSpline` ,
- non-negative monothonic spline `mSpline` ,
- non-negative monothonic convex or concave spline `cSpline` ,
- non-negative convex spline `convexSpline` ,
- non-negative concave spline `concaveSpline` .

`RooFit` *-based parameterizations*

```
r1  = histo.pdf_positive          ( 5 ) ## parameterize and non-negative degree-5 Bernstein sum
r2  = histo.pdf_positiveeven      ( 5 ) ## parameterize and non-negative degree-5 even Bernstein polynomial
r3  = histo.pdf_increasing        ( 5 ) ## parameterize and non-negative degree-5 increasing Bernstein polynomial
r4  = histo.pdf_decreasing        ( 5 ) ## parameterize and non-negative degree-5 decreasing Bernstein polynomial
r5  = histo.pdf_convex_increasing ( 5 ) ## parameterize and non-negative degree-5 convex  increasing Bernstein polynomial
r6  = histo.pdf_convex_decreasing ( 5 ) ## parameterize and non-negative degree-5 convex  decreasing Bernstein polynomial
r7  = histo.pdf_concave_increasing ( 5 ) ## parameterize and non-negative degree-5 concave increasing Bernstein polynomial
r8  = histo.pdf_concave_decreasing ( 5 ) ## parameterize and non-negative degree-5 concave decreasing Bernstein polynomial
r9  = histo.pdf_concavepoly       ( 5 ) ## parameterize and non-negative degree-5 concave Bernstein polynomial
r10 = histo.pdf_convexpoly        ( 5 ) ## parameterize and non-negative degree-5 convex  Bernstein polynomial
```

Similarly there are methods that provdies the parameterization in terms of *splines* :

- `pdf_pSpline` : non-negative *b-spline*
- `pdf_mSpline` : non-negative monothonic *b-spline*
- `pdf_cSpline` : non-negative monothonic concave or convex *b-spline*
- `pdf_convexSpline` : non-negative monothonic convex *b-spline*
- `pdf_concaveSpline` : non-negative monothonic concave *b-spline*

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**

```python
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)

```

```
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 )
```


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` :

```
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)
bare  = ROOT.RooGaussian('Gauss','Gaussian', x , mean , sigma ) ## <--- HERE
```

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 ) )
   ```

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

   ```
   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` : make the estimates for the minimal and maximal values for the PDF. For some models it is done analytically or semianalitycally, for remainig models it is done 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

Ostap provides helper class that simplify construction of fit models taking into accotun resolution functions:

```
pdf = ...
cnv_pdf = Convolution_pdf ( 'Cnv            ' ,
                            pdf = pdf        ,
                            resolution = ... )
```

As `resolution` one can specify

1. Any resolutuon model (`RooAbsPdf`)
2. simple number `s` , in this case the gaussian resolution model with sigma = `s` will be used
3. Any `RooAbsReal` objetct, it will be used as sigma for gaussian resoltuion model

There are several optional flags

- `useFFT=True` : use *Fast-Fourier-Transform* or plain numerical convolution ?
- `nbins=100000` : sampling for *Fast-Fourier-Transform*
- `buffer=0.25` : buffer size for *Fast-Fourier-Transform*, argument for `setBufferFraction` call
- `nsigmas=6` : window size for plain numeric convolution, the argument for `setConvolutionWindow` call

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 )
```

`1D` -models

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

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

Generic backrgound models

Polynomial models

Here the list of the most useful polynomial models:

- `PolyPos_pdf` : positive (non-negative) polynomial
- `PolyEven_pdf` : positibe (non-negative) *symmetric* polynomial: `p(x)= p(2*x0-x)` , where `x0=0.5*(xmin+xmax)`
- `Monotonic_pdf` : positive (non-negative) polynomial with fixed sign of the first derivative: posynomial either non-decreasing or non-increasing
- `Convex_pdf` : positive (non-negative) polynomial with fixed signs of the first (non-decreasing or non-increasing) and second (convex or concave) derivatives
- `ConvexOnly_pdf` : positive (non-negative) polynomial with fixed sign of the second (convex or concave) derivative

Phasespace-based models

Here the list of the most useful phasespace-based models:

- `PS2_pdf` : 2-body phase space (no parameters)
- `PSLeft_pdf` : Low edge of N-body phase space
- `PSRight_pdf` : High edge of L-body phase space from N-body decays
- `PSNL_pdf` : approximation for L-body phase space from N-body decays
- `PS23L_pdf` : 2-body phase space from 3-body decays with orbital momenta

Polynomial-based models

- `Bkg_pdf` : The exponential function, modulated by the positive polynomial. In practice it is the most useful function to describe the combinatorial background
- `PSPol_pdf` : L-body phase space from N-body decays modulated by a positive polynomial
- `Sigmoid_pdf` : sigmoid function (`atanh`) modulated by the positive polynomial
- `TwoExpoPoly_pdf` : difference of two exponents, modulated by the positive polynomial

Spline-based models

The models, based on *B-splines* :

- `PSpline_pdf` : positive (non-negative) spline

- `MSpline_pdf` : positive (non-negative) monothonic (non-decreasing or non-increasing) spline
- `CSpline_pdf` : positive (non-negative) monothonic (non-decreasing or non-inclreasing) convex or concave spline
- `CPSpline_pdf` : positive (non-negative) convex or concave spline

Generic signal models

The signal-like models (peaks):

```
'Gauss_pdf'              , ## simple     Gauss
'CrystalBall_pdf'        , ## Crystal-ball function
'CrystalBallRS_pdf'      , ## right-side Crystal-ball function
'CB2_pdf'                , ## double-sided Crystal Ball function
'Needham_pdf'            , ## Needham function for J/psi or Y fits
'Apolonios_pdf'          , ## Apolonios function
'Apolonios2_pdf'         , ## Apolonios function
'BifurcatedGauss_pdf'    , ## bifurcated Gauss
'DoubleGauss_pdf'        , ## double Gauss
'GenGaussV1_pdf'         , ## generalized normal v1
'GenGaussV2_pdf'         , ## generalized normal v2
'SkewGauss_pdf'          , ## skewed gaussian (temporarily removed)
'Bukin_pdf'              , ## generic Bukin PDF: skewed gaussian with exponential tails
'StudentT_pdf'           , ## Student-T function
'BifurcatedStudentT_pdf' , ## bifurcated Student-T function
'SinhAsinh_pdf'          , ## "Sinh-arcsinh distributions". Biometrika 96 (4): 761
'JohnsonSU_pdf'          , ## JonhsonSU-distribution
'Atlas_pdf'              , ## modified gaussian with exponenital tails
'Slash_pdf'              , ## symmetric peakk wot very heavy tails
'RaisingCosine_pdf'      , ## Raising  Cosine distribution
'QGaussian_pdf'          , ## Q-gaussian distribution
'AsymmetricLaplace_pdf'  , ## asymmetric laplace
'Sech_pdf'               , ## hyperboilic secant  (inverse-cosh)
'Logistic_pdf'           , ## Logistic aka "sech-squared"
#
## pdfs for "wide" peaks, to be used with care - phase space corrections are large!
#
'BreitWigner_pdf'      , ## (relativistic) 2-body Breit-Wigner
'Flatte_pdf'           , ## Flatte-function  (pipi)
'Flatte2_pdf'          , ## Flatte-function  (KK)
'LASS_pdf'             , ## kappa-pole
'Bugg_pdf'             , ## sigma-pole
'Swanson_pdf'          , ## Swanson's S-wave cusp
##
'Voigt_pdf'            , ## Voigt-profile
'PseudoVoigt_pdf'      , ## PseudoVoigt-profile
'BW23L_pdf'            , ## BW23L
```

`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.

Generic signal models

The signal-like models (peaks):

Narrow signals :

- `Gauss_pdf` : simple *Gauss*
- `CrystalBall_pdf` : *Crystal Ball* function
- `CrystalBallRS_pdf` : right-side *Crystal Ball* function
- `CB2_pdf` : double-sided *Crystal Ball* function
- `Needham_pdf` : *Needham* function for J/psi or Upsilon fits
- `Apolonios_pdf` : *Apolonios* function
- `Apolonios2_pdf` : *Apolonios* function
- `BifurcatedGauss_pdf` : bifurcated *Gaussian*
- `DoubleGauss_pdf` : double *Gaussian*
- `GenGaussV1_pdf` : *generalized Gaussian* v1
- `GenGaussV2_pdf` : *generalized Gaussian* v2
- `SkewGauss_pdf` : _skewed Gaussian
- `Bukin_pdf` : generic *Bukin* PDF: skewed gaussian with exponential tails
- `StudentT_pdf` : *Student` T-function*
- `BifurcatedStudentT_pdf` : bifurcated *Student` T-function*
- `SinhAsinh_pdf` : *Sinh-arcsinh* distribution
- `JohnsonSU_pdf` : *Jonhson-SU* distribution
- `Atlas_pdf` , modified *Gaussian* with exponenital tails
- `Slash_pdf` , symmetric peak with very heavy tails
- `RaisingCosine_pdf` , *Raising Cosine* distribution
- `QGaussian_pdf` , *Q-Gaussian* distribution
- `AsymmetricLaplace_pdf` , asymmetric *Laplace*
- `Sech_pdf` , hyperboilic secant (inverse-cosh)
- `Logistic_pdf` , Logistic aka "sech-squared"

"Wide" peaks

These PDF are useful to describe *wide* peaks with the natural width. (Keep in miid that phase space corrections and resoltuion effect could be large)

- `BreitWigner_pdf` : (relativistic) 2-body *Breit-Wigner*
- `Flatte_pdf` : *Flatte* function (pipi)
- `Flatte2_pdf` : *Flatte* function (KK)
- `LASS_pdf` : kappa-pole
- `Bugg_pdf` : sigma-pole
- `Swanson_pdf` : Swanson`s S-wave cusp
- `Voigt_pdf` : Voigt-profile
- `PseudoVoigt_pdf` : PseudoVoigt-profile
- `BW23L_pdf` : BW23L

Generic backrgound models

Here is incomplete list of *background-like* models - the models that often could be used to describe the background distribution

Polynomial models

Here the list of the most useful polynomial models:

- `PolyPos_pdf` : positive (non-negative) polynomial
- `PolyEven_pdf` : positibe (non-negative) *symmetric* polynomial: `p(x)= p(2*x0-x)` , where `x0=0.5*(xmin+xmax)`
- `Monotonic_pdf` : positive (non-negative) polynomial with fixed sign of the first derivative: posynomial either non-decreasing or non-increasing
- `Convex_pdf` : positive (non-negative) polynomial with fixed signs of the first (non-decreasing or non-increasing) and second (convex or concave) derivatives
- `ConvexOnly_pdf` : positive (non-negative) polynomial with fixed sign of the second (convex or concave) derivative

Phasespace-based models

Here the list of the most useful phasespace-based models:

- `PS2_pdf` : 2-body phase space (no parameters)
- `PSLeft_pdf` : Low edge of N-body phase space
- `PSRight_pdf` : High edge of L-body phase space from N-body decays
- `PSNL_pdf` : approximation for L-body phase space from N-body decays
- `PS23L_pdf` : 2-body phase space from 3-body decays with orbital momenta

Polynomial-based models

- `Bkg_pdf` : The exponential function, modulated by the positive polynomial. In practice it is the most useful function to describe the combinatorial background
- `PSPol_pdf` : L-body phase space from N-body decays modulated by a positive polynomial
- `Sigmoid_pdf` : sigmoid function (`atanh`) modulated by the positive polynomial
- `TwoExpoPoly_pdf` : difference of two exponents, modulated by the positive polynomial

Spline-based models

The models, based on *B-splines* :

- `PSpline_pdf` : positive (non-negative) spline
- `MSpline_pdf` : positive (non-negative) monothonic (non-decreasing or non-increasing) spline
- `CSpline_pdf` : positive (non-negative) monothonic (non-decreasing or non-inclreasing) convex or concave spline
- `CPSpline_pdf` : positive (non-negative) convex or concave spline

Other useful models

- `GammaDist_pdf` : Gamma-distributuon in shape/scale parameterization
- `GenGammaDist_pdf` : Generalized Gamma-distribution
- `Amoroso_pdf` : another view of generalized Gamma distribution
- `LogGammaDist_pdf` : Gamma-distributuon in shape/scale parameterization
- `Log10GammaDist_pdf` : Gamma-distributuon in shape/scale parameterization
- `LogGamma_pdf`
- `BetaPrime_pdf` : Beta-prime distribution
- `Landau_pdf` : Landau distribution
- `Argus_pdf` : ARGUS distribution
- `TwoExpos_pdf` : difference of two exponents
- `Gumbel_pdf` : Gumbel distributions
- `Weibull_pdf` : Weibull distributions

Useful to describe pt-spectra:

- `Tsallis_pdf` : Tsallis PDF
- `QGSM_pdf` : QGSM PDF

Useful 2D-background models

2D-models useful to describe non-factorazable (`f(x,y)!=f(x)*f(y)`) background:

- `PolyPos2D_pdf` : positive (non-negative) polynomial in 2D
- `PolyPos2Dsym_pdf` : positive (non-negative) symmetric polynomial in 2D
- `PSPol2D_pdf` : product of phase spaces functions, modulated with 2D polynomial
- `PSPol2Dsym_pdf` : symmetric product of phase spaces, modulated with 2D polynomial
- `ExpoPSPol2D_pdf` : sxponential times phase space times positive 2D-polynomial
- `ExpoPol2D_pdf` : product of exponents times positive 2D-polynomial
- `ExpoPol2Dsym_pdf` : symmetric version of above
- `Spline2D_pdf` : 2D-generic positive (non-negative) spline
- `Spline2Dsym_pdf` : 2D symmetric positive (non-negative) spline

Useful 3D-background models

3D-models useful to describe non-factorazable (`f(x,y,z)!=f(x)*f(y)*f(z)`) background:

- `PolyPos3D_pdf` : positive (non-negative) polynomial in 3D
- `PolyPos3Dsym_pdf` : positive (on-negative) symmetric polynomial in 3D
- `PolyPos3Dmix_pdf` : positive partly symmetric (x<-->y) polynomial in 3D

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

Actually, the separation into *signal*, *background* and *other* components is a bit arbitrary. However it is helpful for

- to define the meaningful names for the fit parameters
- to separate different components for visualisation, since different styles (lines, colors, etc) are used for differentr categories

Access to the model components

The individual 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 parametters 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
```

Extended multi-component fit model

```
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 ] ,
    )
```

One can define some initial setting for fit-parameters:

```
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
```

The fit itself is trivial

```
r, f = model_ext1.fitTo ( dataset , draw = False , silent = True )
r, f = model_ext1.fitTo ( dataset , draw = False , silent = True )
```

And accessing fit results is also simple:

```
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
```

Extended fit model 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   , ## <-- HERE
    combine_backgrounds = True   , ## <-- HERE
    combine_others     = True   , ## <-- HERE
```

Setting the initial values of fit parameters is trivial:

```
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
```

The fit itself and access to fit parameters is the same as above.

Non-extended multi-component fit model with non-recursive fit-fractions

```
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 , ## <--- HERE
    recursive           = False   ## <--- HERE
    )
```

Setting the initial values of fit-fractions:

```
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
```

The fit itself and access to fit parameters is the same as above.

Non-extended multi-component fit model with recursive fit-fractions

```
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 , ## <-- HERE
    recursive           = True  , ## <-- HERE
    )
```

Setting initial fit parameters:

```
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
```

The fit itself and access to fit parameters is the same as above.

Non-extended fit model with compound components and non-recursive fit-fractions

```
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   , ## <--- HERE
    combine_backgrounds = True   , ## <--- HERE
    combine_others      = True   , ## <--- HERE
    ##
    extended            = False  , ## <--- HERE
    recursive           = False    ## <--- HERE
    )
```

Setting the initial values:

```
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
```

The fit itself and access to fit parameters is the same as above.

Non-extended fit model with compound components and recursive fit-fractions

```
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   , ## <--- HERE
    combine_backgrounds = True   , ## <--- HERE
    combine_others      = True   , ## <--- HERE
    ##
    extended            = False  , ## <--- HERE
    recursive           = True     ## <--- HERE
```

Setting the initial values:

```
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
```

The fit itself and access to fit parameters is the same as above.

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
# ============================================================================
```

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

sPlot

Using sPlot is rather trivial in Ostap:

```
dataset = ...
model   = Fit1D ( signal = ... , backgrund = ... )
model.fitTo ( dataset )
print datatset
model.sPlot ( dataset )  ## <--- HERE
print datatset           ## <--- note appearence of new variables
```

Using *Weighted fits*

Often one needs to fit *weighted dataset*, etg. *backrgonud-subtracted* or *efficiency_corrected*. It is purely trivial in Ostap:

```
dataset = ...
dsw     = dataset.makeWeighted( 'S_sw/eff' ) ##
model   = ...
model.fitTo ( dsw , .... , sumw2 = True , ... ) ## <--- HERE
```

Using *Constraints* in the fit

Often oen can add *soft Gaussian constraint* for soem fit parameters, e.g. one can constraing the rsignal resolution:

```
sigma_MC       = VE( 0.015 , 0.001**2 ) ##
sigma_cnt      = model.sigma.constrainTo ( sigma_MC , 'sigma_constraint')
my_constraints = ROOT.RooFit.ExternalConstraints ( ROOT.RooArgSet ( sigma_cnt  ) )
dataset         = ...
model.fitTo ( dataset , ... , constraints = my_constraints , ....  )
```

Clearly several constraints can be combined togather

```
sigma_cnt  = model.sigma.constrainTo ( sigma_MC           , 'sigma_constraint')
peak_cnt   = model.mean .constrainTo ( VE(3.096,0.001**2) , 'mass_constraint' )
my_constraints = ROOT.RooFit.ExternalConstraints ( ROOT.RooArgSet ( sigma_cnt , peak_cnt ) )
```

For the next version of ostap, one will be able to avoid the explicit creation of `ROOT.RooFit.ExternalConstraint` and `ROOT.RooArgSet`

```
sigma_cnt  = model.sigma.constrainTo ( sigma_MC           , 'sigma_constraint')
peak_cnt   = model.mean .constrainTo ( VE(3.096,0.001**2) , 'mass_constraint' )
model.fitTo ( dataset , ... , constraints = ( sigma_cnt , peak_cnt ) , ....  )
```

Tools

There are several *tools* embedded in Ostap to implement common analysis operations

- using `TMVA`
- training `TMVA` using *chopping* approach
- Reweighting
- easier access and manipulations with `PidCalib`

Using `TMVA`

Ostap hosts couple of classes, that simplifies the training and using of `TMVA` .

Training `TMVA`

```
tSignal  = ... ## signal     TTree/TChain
tBkg     = ... ## background TTree/TChain
## book TMVA trainer
from Ostap.PyTMVA import Trainer
trainer = Trainer (
   name    = 'TestTMVA' ,
   methods = [
   # type                   name   configuration
   ( ROOT.TMVA.Types.kMLP        , 'MLP'        , 'H:!V:EstimatorType=CE:VarTransform=N:NCycles=200:HiddenLayers=N+3:TestRate=
5:!UseRegulator' ) ,
   ( ROOT.TMVA.Types.kBDT        , 'BDTG'       , 'H:!V:NTrees=100:MinNodeSize=2.5%:BoostType=Grad:Shrinkage=0.10:UseBaggedBoo
st:BaggedSampleFraction=0.5:nCuts=20:MaxDepth=2' ) ,
   ( ROOT.TMVA.Types.kCuts       , 'Cuts'       , 'H:!V:FitMethod=MC:EffSel:SampleSize=200000:VarProp=FSmart' ) ,
   ( ROOT.TMVA.Types.kFisher     , 'Fisher'     , 'H:!V:Fisher:VarTransform=None:CreateMVAPdfs:PDFInterpolMVAPdf=Spline2:Nbins
MVAPdf=50:NsmoothMVAPdf=10' ),
   ( ROOT.TMVA.Types.kLikelihood , 'Likelihood' , 'H:!V:TransformOutput:PDFInterpol=Spline2:NSmoothSig[0]=20:NSmoothBkg[0]=20:
NSmoothBkg[1]=10:NSmooth=1:NAvEvtPerBin=50' )
   ] ,
   variables  = [ 'var1' , 'var2' ,  'var3' ] , ## Variables to be used for training
   signal     = tSignal                      , ## ``Signal'' sample
   background = tBkg                         , ## ``Background'' sample
   verbose    = False )
```

Optionally one can specify also `signal_cuts` and `background_cuts` .

Traing `TMVA` itself is trivial, one needs to invoke the method `train` :

```
weights_files = trainer.train ()
```

It returs the list/tuple of *weight-* `XML` *-files*, the output of `TMVA` trainer. Optionally one can retrieve also the list of _ `C++-class` -files, using the proeprty `class_files` or everything together in a form of `tar` -file using the property `tar_file` :

```
weight_files = trainer.weight_files ## XML weights
class_files  = trainer.class_files  ## C++ classes
tar_file     = trainer.tar_file     ## everything together
```

Using `TMVA`

To use trained `TMVA` one exploits `TMVA` *reader*:

```
from Ostap.PyTMVA import Reader
reader = Reader(
   'MyMLP' ,
   variables     = [ ('var1' , lambda s : s.var1 )   ,
                     ('var2' , lambda s : s.var2 )   ,
                     ('var3' , lambda s : s.var3 ) ] ,
  weights_files = tar_file  )
```

What is `lambda s : s.var1` here?

The the element of the pair is, obviously, the variable name. The second argument is *accessor function*. It will be applied for *1-argument call* of the *method*. E.g. in this example, one can apply it to `TTree` / `TChain` / `RooAbsData` / `RooArgSet` and the variable `var1` from this `TTree` / `TChain` / `RooAbsData` / `RooArgSet` will be used as `'var1'` for the `TMVA`. Accessor functions coudl be trivial, as on this case, but they also can be less trivial:

```
variables     = [ ('var1' , lambda s : s.rapidity     )   , ## use another name
                  ('var2' , lambda s : s.pt/1000      )   , ## make some rescaling
                  ('var3' , lambda s : atan2(s.y,s.x) ) ] , ## make more complicated calculations
```

If one wants to use other objects for *1-argument call*, other set of accessor functions need to be supplied. E.g. if data are expected to be supplied as a `tuple` / `list` / `std::vector<...>`, one can use

```
variables     = [ ('var1' , lambda s : s[0] )   , ## use another name
                  ('var2' , lambda s : s[1] )   , ## make some rescaling
                  ('var3' , lambda s : a[2] ) ] , ## make more complicated calculations
```

One can also use just the plain list of variable names:

```
variables     = [ 'var1' , 'var2' , 'var3' ]
```

This list will be automatically transformed into

```
variables     = [ ('var1' , lambda s : getattr( s , 'var1') ) ,
                  ('var2' , lambda s : getattr( s , 'var2') ) ,
                  ('var3' , lambda s : getattr( s , 'var3') ) ]
```

As `weight_files` arguments one can use either the list of *weights-files* from *the trainer*, or, *much easier*, use the single *'tar'-file* from *the trainer*. The methods, available from the weight files can be checked as

```
print reader.methods
`
```

And the usage of the reader is rather trivial, e.g. one can explicitly request the responce for certain set of arguments:

```
v1,  v2,  v3 = ....
mlp  = reader['MLP']                    ## get  one method
print 'MLP value is %s'  % mlp ( v1 , v2 ,  v3 )
```

In practice, one practially always uses it with `TTree` / `TChain` / `RooAbsData` / `RooArgSet`, in this case one use *1-argument call*, assuming then proper *accessor functions* are supplied:

```
tree = ... ##  the tree
mlp  = reader['MLP']                    ## get  one method
for i in tree :                         ## loop over the entries
    print 'MLP value is %s'  % mlp ( i ) ## get the  value
```

Using *chopping* for `TMVA` training/using

Chopping is a technique to use the limited set of data for TMVA training. In this approach data are *chopped* into several categories and for each category `i` TMV is trained using the remaing `N-1` categories, and the trained `TMVA` is applied to the events from category `i` .

Training `TMVA` -chopper

The *trainig-with-chopping* is fairly trivial. First one need to define the number of distinct categories and the function to classify the events into training categories. E.g. for `TMVA` training

```
tSignal  = ... ## signal     TTree/TChain
tBkg     = ... ## background TTree/TChain
## book TMVA trainer
from Ostap.TMVAChopper import Trainer
trainer = Trainer (
  N        = N                , ## ATTENTION! N is number of categories
  category = "137*evt+813*run" , ## ATTENTION! It is a classification function
  chop_signal       = False    , ## chop the signal     ?  (default)
  chop_background   = True     , ## chop the background ?  (default)
  ...
```

All other arguments of `Trainer` are the same as for regular `TMVA` trainer. Arguments `chop_signal` and `chop_background` defined what sample (or both) to be *chopped*. The argument caterory described *the integer-valued function*, used for classification of events. Actually *trainer* construct classification function as `category%N` .

How to choose chopping parameters?

For efficient usage of events number of categroeis shoudl be rather large `N>>2` . For the given number of categories `N` , the fraction of events used for `TMVA` training is `(N-1)/N` . Therefore with large `N` events are used more efficiently. From other side, for large `N` the traing time is proportional to `N` , while the traing results shodul be more or less independent on `N` . It makes senseless usage of `N>100` . Therefore one gets `2<<N<100` . In practice it is convinient to choose `10<N<20` .

The ideal classification function *must* be independent on the properties of signal and or background. It should be `pseudorandom` and provide almost uniform population of categories. It is very easy to achive using the following expression `(Na*a+Nb*b+Nc*c+...+Nz*z+)%N` , where `a` , `b` , ... , `z` are some integer-valued variables from the input `TTree` / `TChain` (event number, run-number, GPS time in nanoseconds, number of tracks in event. number of hits in SPD, etc...), and `Na` , `Nb` , ..., `Nz` are prime numbers, that are large enough (`Na>>N` , `Nb>>N` , ... , `Nz>>N`). With such construction, choosing `N` to be also a prime number, one almost guaranteed that events are *randomly* distributed into `N` -categories.

The category population can be checked using setof control historgams:

```
bc =  trainer.background_categories
sc =  trainer.signal_categories
bc[0].Draw() ## show popultion of background categroies
bc[1].Draw() ## the same with  different binning

sc[0].Draw() ## show popultion of signal categroies
sc[1].Draw() ## the same with  different binning
```

Using `TMVA` -chopper

Again one needs to define the classification function for input data. Clealry this function should match the one used in training

```
category = lambda s :  int ( s.evt*137 + 813*s.run ) % N ## the  classification function
from Ostap.TMVAChopper import Reader   ## ATTENTION
reader = Reader(
    N             = N         , ##  number of   categories
    categoryfunc  = category  , ## category
    ...
```

All other arguments of `Reader` are the same as for regular `TMVA` reader. The created *reader* is used exactly in the same way as for *no-chopping*-case:

```
tree = ... ##  the tree
mlp  = reader['MLP']                    ## get  one method
for i in tree :                         ## loop over the entries
    print 'MLP value is %s'  % mlp ( i ) ## get the  value
```

For tests and debug

For test and debug purposes one can use it also as a function:

```
v1,  v2,  v3 = ....
mlp  = reader['MLP']    ## get  one method
for i in range ( N ) :  ## loop over   categories
    print 'MLP value for  categroy %s is %s'  % ( i , mlp ( i , v1 , v2 ,  v3 ) )
```

And even get the difference between responces for different categories. Clearly the spread of values should be small enough

```
v1,  v2,  v3 = ....
mean = mlp.mean ( v1 , v2 , v3 ) ## get a mean-value over  different  categories
stat = mlp.stat ( v1 , v2 , v3 ) ## get a statistics  (mean,rms, min/max,...) of  responces
```

Reweighting

Ostap offers set of utlities to *reweight* the distributions. Typical use-case is

- one has set of *data* distributions
- and *simulation* does not describe these distributions well, and one needs to *reweight* simulation to describe all distributions

It is relatively easy procedure in Ostap, however it requires some code writing.

Data and *simulated* distributions

First, one needs to specify *data* distributions. It can be done in form of 1D,2D and 3D histograms, or as 1,2, or 3-argument functions or even all these techniques could be used together. It is important that these data distributions should be strickly positive for the corresponding range of variables. E.g. in case of histograms, there should be no empty or negative bin content.

```
hdata_x = ROOT.TH1D ( ... )               ## e.g. use the histogram
hdata_x = lambda x :  math.exp (-x/10 ) ) ## or use a function
...
hdata_y = ...
```

Second, for each *data distribution* one needs to prebook the corresponding template histogram that will be filled from *simulated* sample. This template histogram should have the same dimensionality (1,2,3) and the corresponidg *data distribtion*. If *data distribution* is specified in a form of historgam, the edges of prebooked template histogram should correspond to the edges of data distribution, but there is no requirements for binning. Binning could be arbtrary, provided that there are no empty bins.

```
hmc_x = ROOT.TH1D ( ... )
hmc_y = ....
```

Iterations

Third, one needs to create *empty* database where *the iterative weights* are stored:

```
import Ostap.ZipShelve as DBASE
dbname = 'weights.db'
with DBASE.open( dbname ,'c') as db :
     pass
```

Since *Reweighting* is essentially iterative procedure, we need to define some maximal number of iterations

```
iter_max =   10
for iter in range(iter_max) :
   ...
```

Weighter object

And for each iteration we need to create *weighting* object, that reads the current weights from database `weight.db`

```
from Ostap.Reweighting import Weight
weightings = [
    ##         accessor function    address indatabase
    Weight.Var ( lambda s : s.x  , 'x-reweight'  ) ,
    ...
]
weighter   = Weight ( dbname , weightings )
```

What is it?

The accessor function is used to get the variable from *simulated sample*. E.g. in this form, `TTree` / `TChain` / `RooDataSet` / `RooArgSet` can be used as source of *simulated* data. but it could be also e.g. some table, `numpy` array or any other storage. In this case the accessor function needs to be modified accordingly. The second parameter specify the location in (newly created empty) database, where the current weights are to be taked from. Since the newly created database is empty, for the first iteration all weights are *trivial* and equal to 1:

```
mc_tree =  ...
for i in range(100):
  mc_tree.GetEntry(i)
  print ' weight for event %d is %s' % ( i , weighted (  mc_tree ) )
```

Weighted simulated sample

As the next step one needs to prepare *simulated dataset*, `RooDataSet` , that

- contains all varables for reweighting
- the current values of *weights*, provided by `weighter` -object above

There are many ways to achive this. E.g. one can use `SelectorWithVars` -utility to decode data from input `TTree` / `TChain` into `RooDataSet` :

```
from Ostap.Selectors   import SelectorWithVars, Variable
## variables to be used in MC-dataset
variables  = [
   Variable ( 'x'      , 'x-var'  , 0  , 20 , lambda s : s.x ) ,
   ...
   Variable ( 'weight' , 'weight' , accessor = weighter       )
   ]
## create new "weighted" mcdataset
selector = SelectorWithVars (
   variables ,
   '0<x && x<20 && 0<y && y<20'
   )
## process
mc_tree.process ( selector )
mcds = selector.data      ## newly created simulated dataset
print mcds
```

Calculate the updated weights and store them in database

At the next step we calculate the updated weights and store them in database

```
from Ostap.Reweighting import makeWeights,  WeightingPlot
plots  = [
  ##            what      how        where          data      simulated-template
  WeightingPlot ( 'x'   , 'weight' , 'x-reweight'  , hdata_x , hmc_x       ) ,
  ...
  ]
## calculate updated weights and store them in database
more = makeWeights ( mcds , plots , dbname , delta = 0.01 ) ## <-- HERE
```

The object `WeightingPlot` defines the rule to fill *simulated* histogram from *simulated* dataset and associated the filled *simulated* histogram with *data distribution*. The actual correction to the weights is calculated according to the rule `w = dd / mcd` , where `dd` is a *density* for the data distribution and `mcd` is a *density* for *simulated* distribution. The *weights* `w` are calculated for each entry in array `plots` , they

are properly normalized and stored in database `dbname` to be used for the next iteration. The function `makeWeights` also print the *statistic* of *normalized weights*:

```
# Ostap.Reweighting         INFO    Reweighting:           ``x-reweight'': mean/(min,max):        (1.00+-0.00)/(0.985,1.012) R
MS:(0.74+-0.00)[%]
```

The last entries in this row summarize the statistics of corrections to the current weight. In this example, the mean correction is `1.00` , the minimal correction is `0.985` , the maximal correction is `1.012` and rms for corrections is `0.74\%` . In tihs example one sees that for this paricualr iteration th ecorrections are rather small, and probably one can stop iterations. Two parameters `delta` and `minmax` of `makeWeights` function allows to automatized th emakinnng the decison. If calculated rms for all corrections is less than specified `delta` parameter and for each correction minnmax-difference deos not exceeed the specified `minmax` -parameter (the default value is `0.05`), function return `False` (meaning *no more iterations are needed*), otherwise it returns `True` . And using this hint one can stop iterations or go further:

```
if not more and iter > 2 :
    print  'No more iteratinos  are needed!'
    break
```

Compare data and simulated distributions for each iteration (optional)

In practice it is useful (and adviseable) to compare the *data* and *simulated* distributions at each iteration to hjave better control over the iteration process. One can make this comparion using zillions of the ways, but for the most imnportant case in practice, where *data distribution* is specified in a form of histogram, there are some predefined utilities

```
## prepare simulated distribution with current weights:
mcds.project ( hmc_x , 'x' , 'weight' )

## compare the basic properties: mean, rms, skewness and kurtosis
hdata_x.cmp_prnt ( hmc_x , 'DATA' , 'MC' , 'DATA(x) vs MC(x)' )

## calculate the ``distance``:
dist = hdata_x.cmp_dist ( hmc_x , density = True )
print "DATA(x)-MC(x)  ``distance''        %s" % dist

## calculate the 'orthogonality'
cost = hdata_x.cmp_cos  ( hmc_x , density = True )
print "DATA(x)-MC(x)  ``orthogonality'' %s" % cost

## find the points of the maximal difference
mn,mx = hdata_x.cmp_minmax ( hmc_x   , diff = lambda a,b : a/b , density = True )
print "DATA*(x)/MC(x) ``min/max-distance''[%%] (%s)/(%s) at x=%.1f/%.1f" % (
        (100*mn[1]-100) , (100*mx[1]-100) , mn[0]  , mx[0] )
```

Using the result

```
from Ostap.Reweighting import Weight
weightings = [
    ##         accessor function    address indatabase
    Weight.Var ( lambda s : s.x  , 'x-reweight'  ) ,
    ...
]
weighter = Weight ( dbname , weightings )
mc_tree  =  ...
for i in range(100):
  mc_tree.GetEntry(i)
  print ' weight for event %d is %s' % ( i , weighted (  mc_tree ) )
```

Note that due to explicit specification of *accessor function*, *reweighter* can be customised to work with any type of input *events/records*. e/g/ assuem that *event* is a plain *array*, and `x` -variable corresponds to index `0` :

```
from Ostap.Reweighting import Weight
weightings = [
    ##         accessor function    address indatabase
    Weight.Var ( lambda s : s[0]  , 'x-reweight'  ) ,
    ...
]
weighter = Weight ( dbname , weightings )
mc_tree  =  ...
for event in  events :
  print ' weight for event %s is %s' % ( event , weighted ( event ) )
```

Abstract reweighting

Abstract reweighting

Due to the freadom in choosing the accessor function, one can apply reweighting procedure to the absolutely abstract samples. E.g. consider the follwing case

- *data distribution* : simple function
- *simulated sample* : random number generator

As a result of *reweighting* procedure, we'll get *reweighted simulated sample*, that will be just a random number generator, that produces the *weighted* distribution according to the specified function. For this case, the code is very transparent and compact:

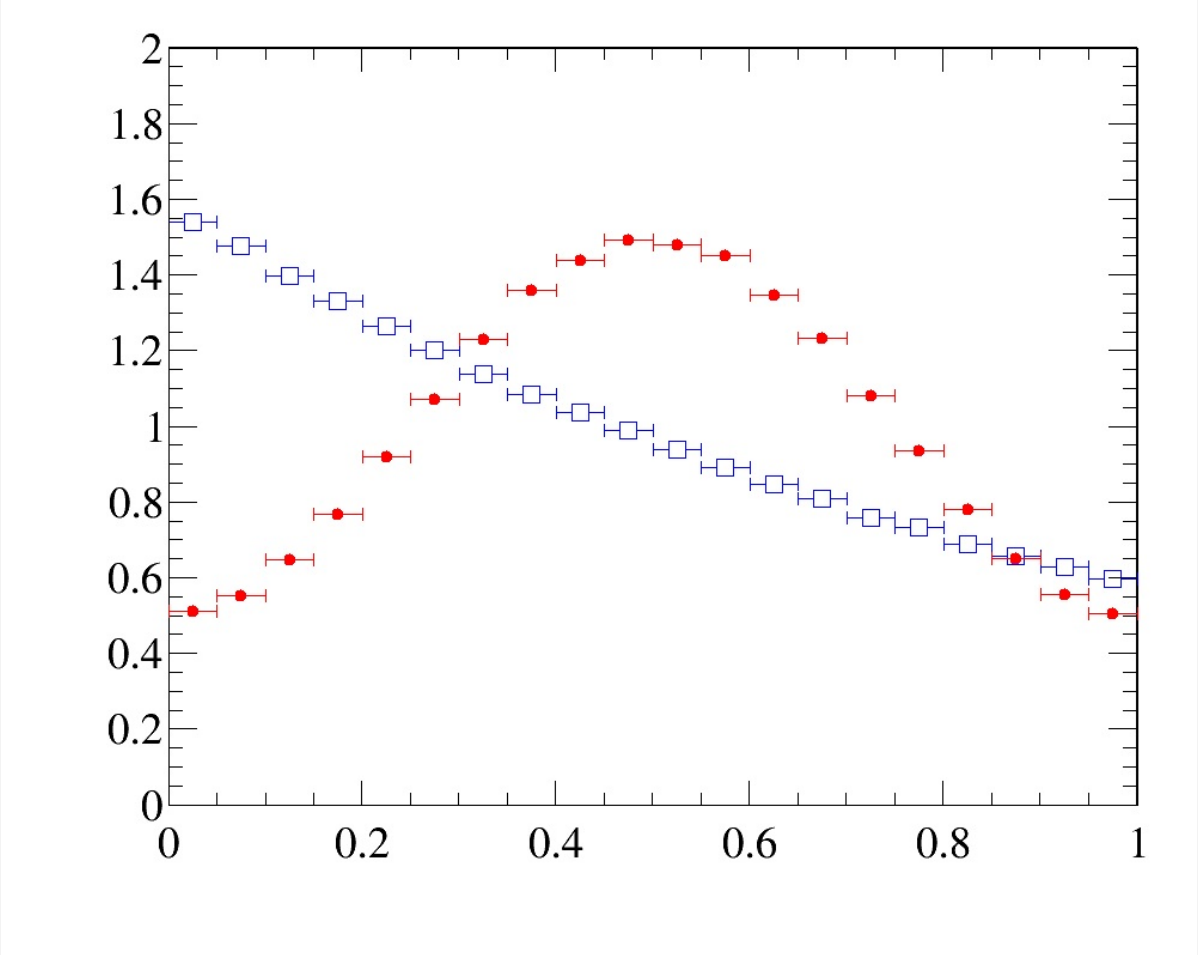
```
# ==============================================================================
## 1)  ``DATA distribution''  - plain function
def data ( x ) :
    return 0.5 + math.sin ( x * math.pi )**2
# ==============================================================================
## 2) ``simulation template'' -  histogram template for simulated sample
mc_hist = ROOT.TH1F ( 'hMC', '', 20 , 0 , 1  )
# ==============================================================================
def mc_sample () :
    x = random.expovariate ( 1 )
    while x  > 1  : x -=1
    return x
# ==============================================================================
## 3) create empty database with initial weights
# ==============================================================================
import Ostap.ZipShelve as DBASE
if os.path.exists ( dbname ) : os.remove ( dbname )
with DBASE.open( dbname ,'c') as db :
```

And then one starts iterations:

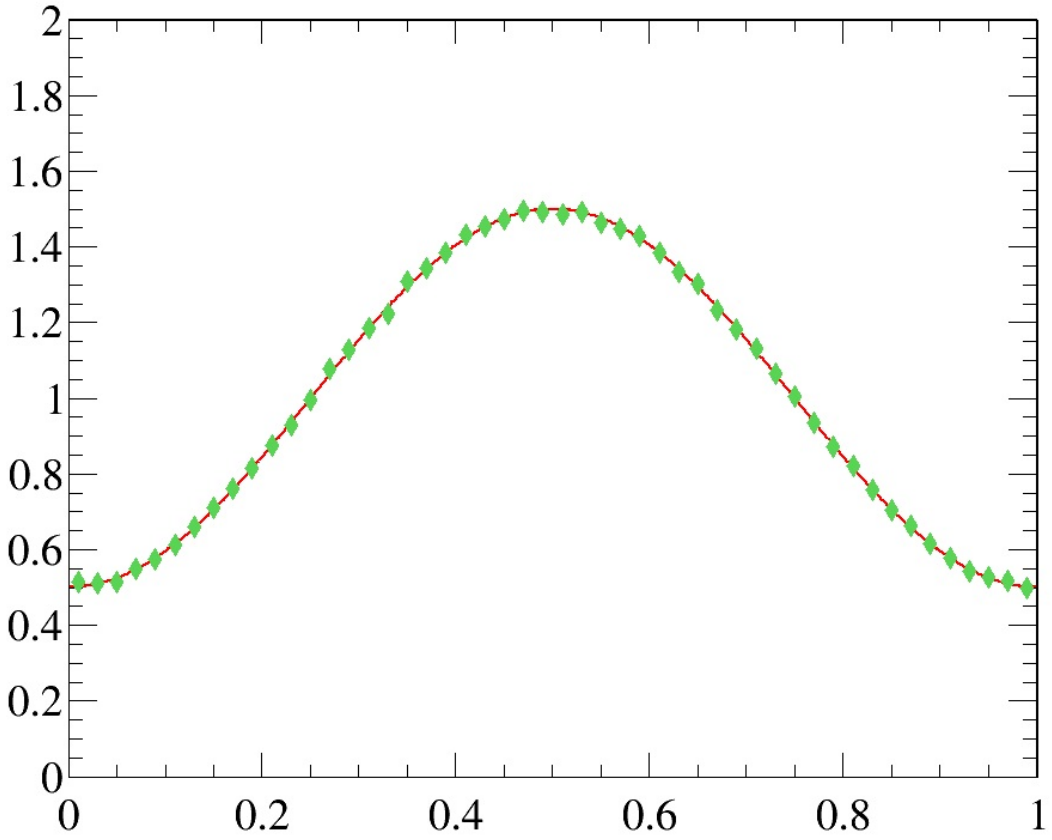
```
# ============================================================================
## 4) prepare reweigthing iterations
# ============================================================================
from Ostap.Reweighting import Weight, makeWeights,  WeightingPlot
from Ostap.Selectors   import SelectorWithVars, Variable
for iter in range ( 100 ) :
    ##                                          accessor       address in DB
    weighter   = Weight( dbname ,  ( Weight.Var ( lambda x : x , 'x-reweight'  ) , ) )

    ## create ``weighted'' simulated  dataset using the current weights
    selector = SelectorWithVars (
        selection  = '1<2' , ## fake one :-( to be removed soon
        silence    = True  ,
        variables  = [ Variable ( 'x'      , 'x-var'  , 0  , 1  , lambda x : x ) ,
                       Variable ( 'weight' , 'weight' , accessor = weighter    ) ] )
    for  i in  range ( 1000000 ) :
        x = mc_sample ()
        selector ( x )
    mcds = selector.data
    ##  update weights: the rule to create weighted simulated histogram
    plots = [ WeightingPlot ( 'x' , 'weight' , 'x-reweight' , data , mc_hist  ) ]
    ## calculate the updated weights and add them into   database
    more = makeWeights ( mcds , plots , dbname , delta = 0.01 )
    if not more and 2 <= iter :
        logger.info ( 'No more iterations are needed #%d' % iter )
        break
```

The full example for *abstract reweighting*, is accessible here

The *density* distribution for the *simulated* sample for before the first (blue open squares) and after the last (filled red points) iterations are shown here,



while the comparison of the initial *data distribution* (red line) and the *reweighted simulated sample* (greed filled diamonods) are shown here.



Why one needs iterations?

One can argue that low-dimension reweigthing can be done withoky iterations, just in one-go. Why one needs iterations here?

The answer is rather simple: yes for very simple case, like 1D-reweigthing, already the first iteration should provide the exact result. However it is true only if *data dsitribution* is suppleds and the historgam and the template for the *simulated* histogram has the same binning. Otherwise the differnt binning scheme results in non-exact result for 1-step reweighting.

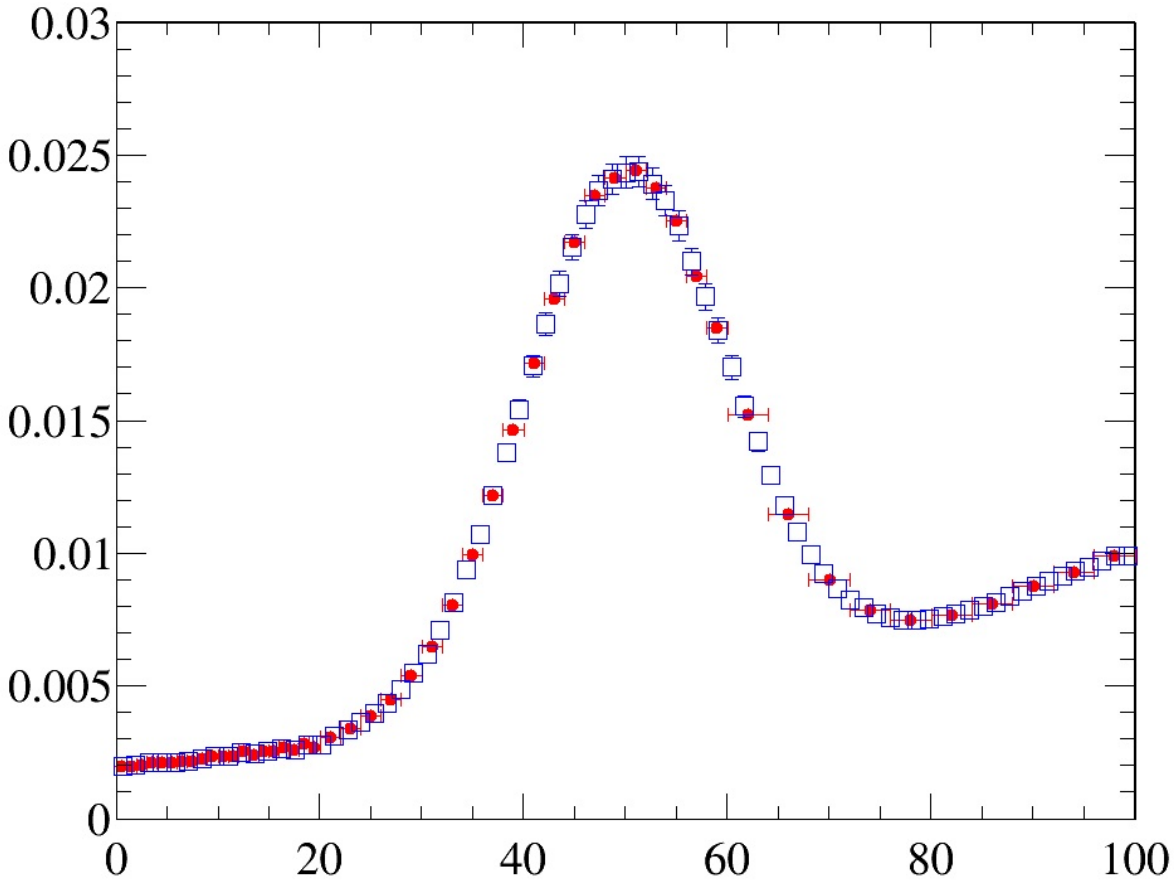
For multidimensional reweighting one can avoid iteration only if all innvolved variables are totally uncorrelated, otherwise the iterative procedure is unavoidable.

Moreover in the presense of correlations *oscillation effect* could occur, that prevents the quick convergency of the iterative procedure. To solve this problem, `makeWeighted` -function fior multidimensional case actually *under-correct* the results. It increases the number of nesessary iterations and make the reweighting procedure more slow, but itpractially eliminates the *oscillation effect*

Examples

Simple `1D` *-reweighting*

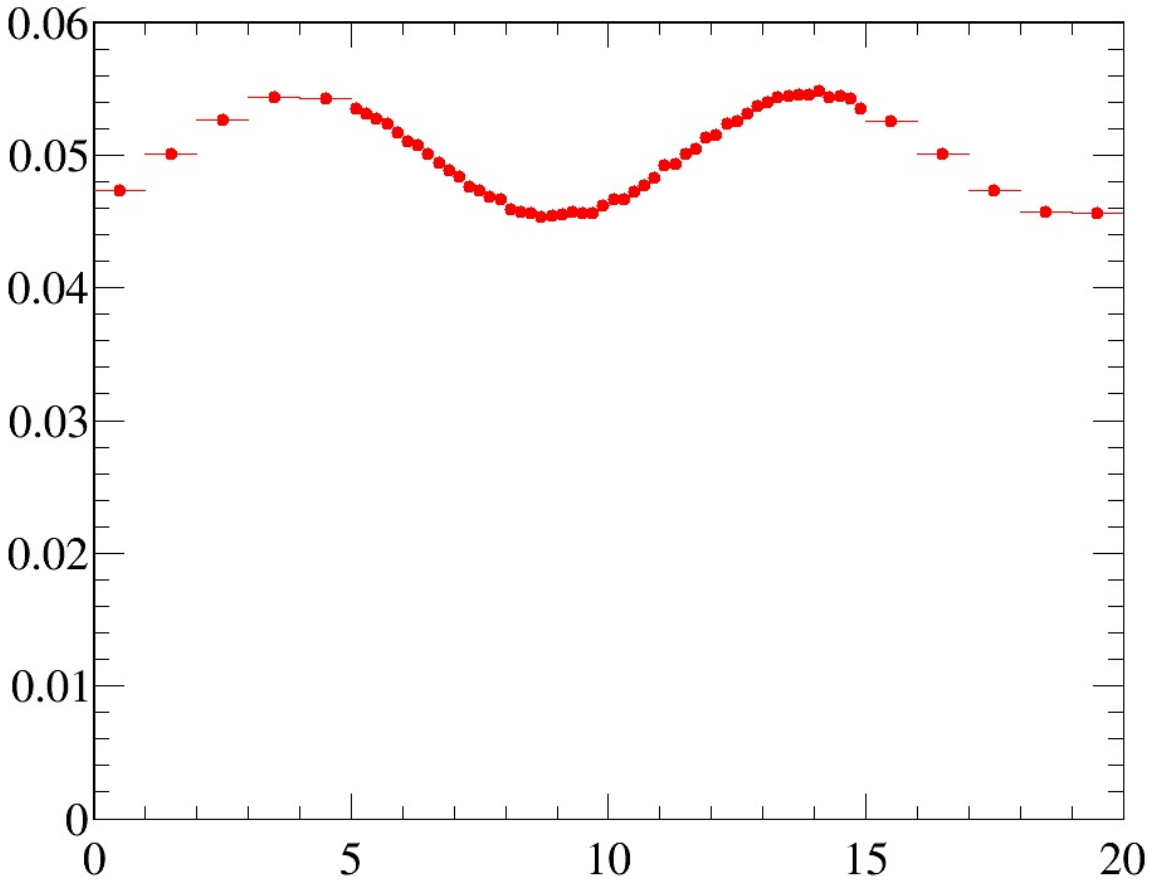
The example of simple 1D-reweighting can be inspected here, while the reweigthing result for the last iteration (blue open squares) are compared with *data distribution* (red filled circled) here:

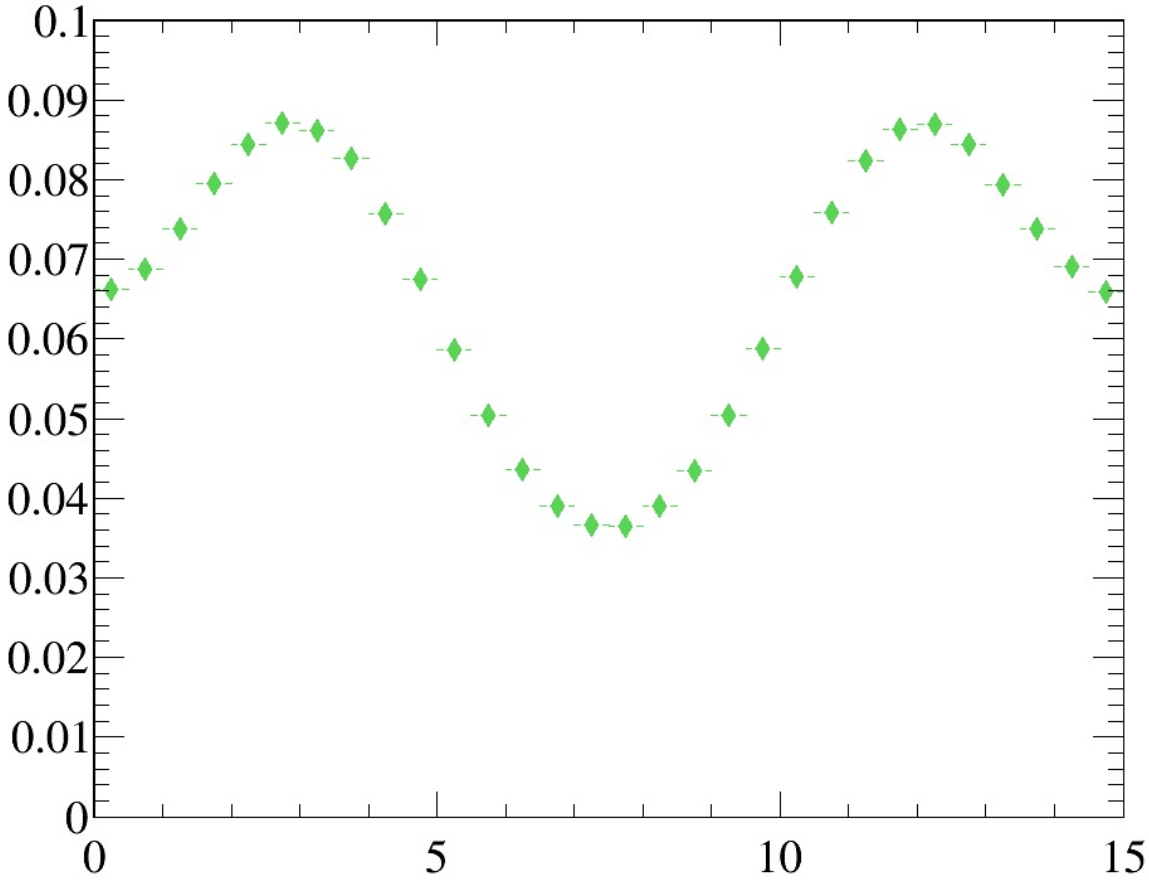


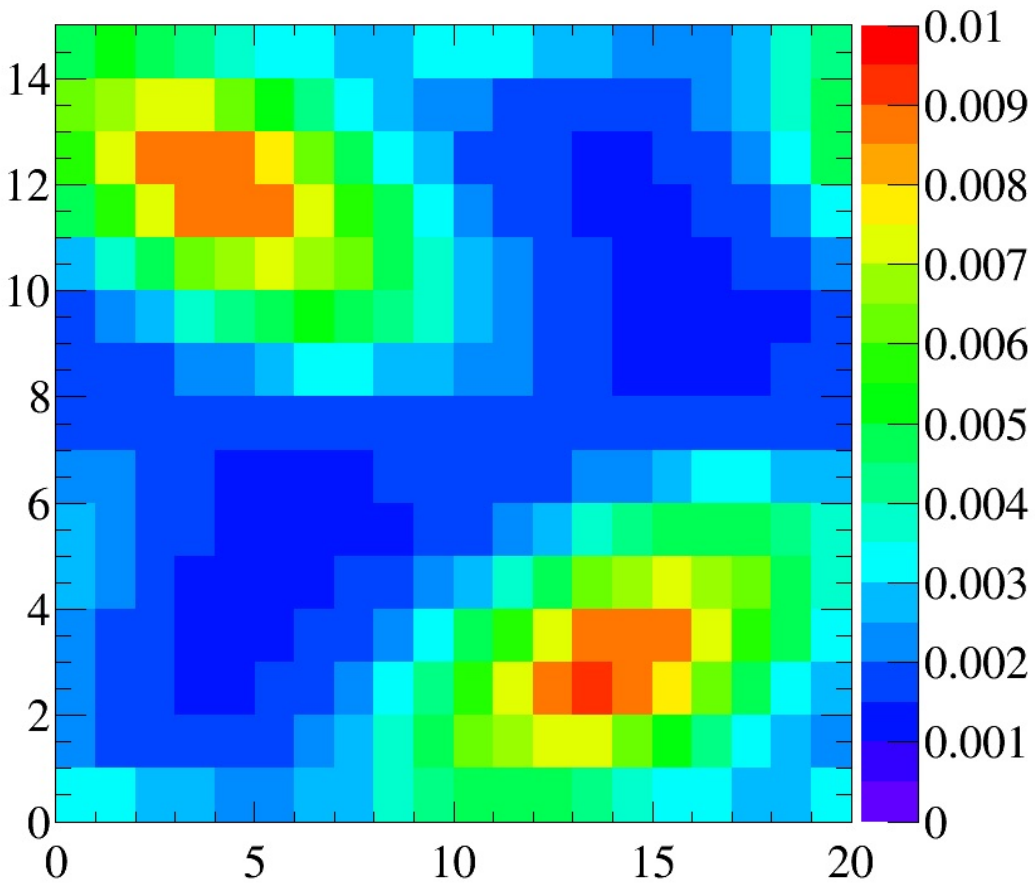
The example also illustrates how to use various *histogram comparison functions* to have better control over the iterative process

More complicated case of non-factorizeable `2D` *-reweighting*

The example of advanced 2D-reweighting can be inspected here. In this example we have three *data distributions* fro two variables 1 one-dimensional `x` -distribution with fine binninig 1 one-dimensional `y` -distribution with fine binninig 1 two-dimensional `y:x` -distribution with coarse binning


0.06
0.05
0.04
0.03
0.02
0.01
0
0
5
10
15
20


0.1
0.09
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0
0
5
10
15

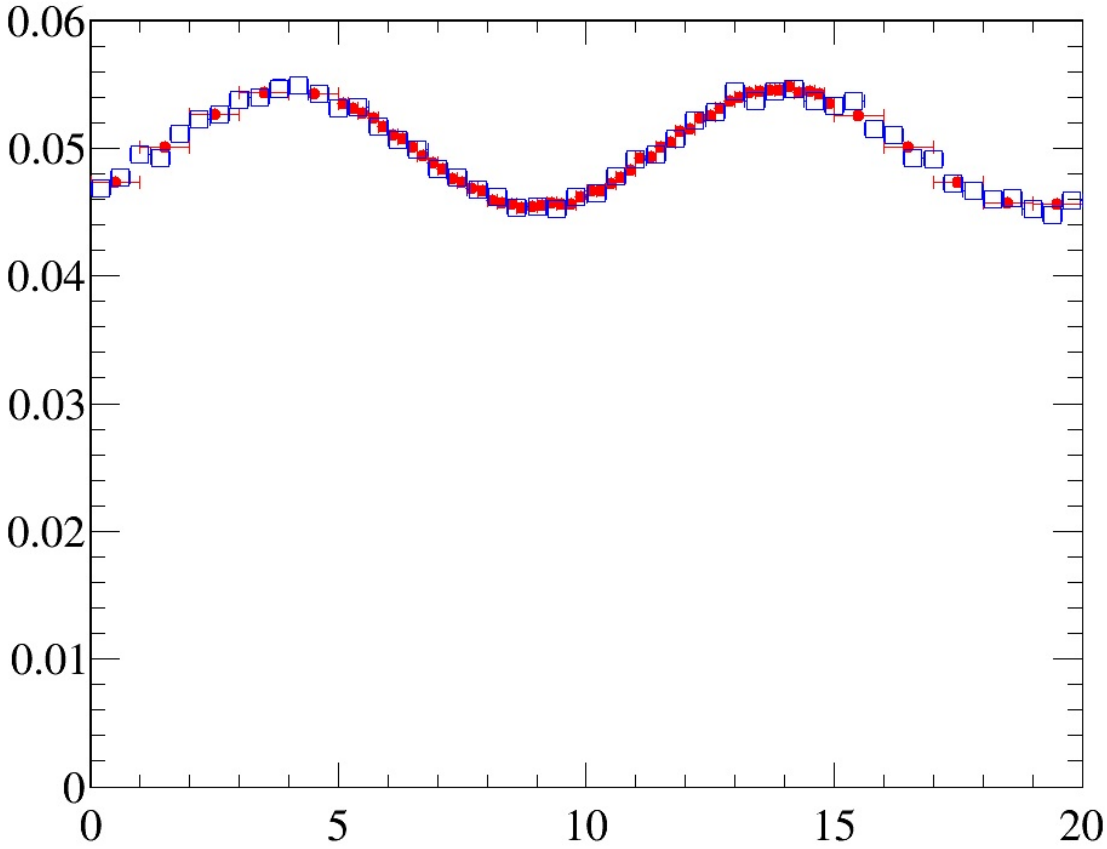


It reflects relatively frequent case of *kinematic reweighting* using the transverse momentum and rapidity. Typically one has enough events to make fine-binned one-dimensional reference distributions, but two-dimensional distributions can be obtained only with relatively coarse binning scheme.

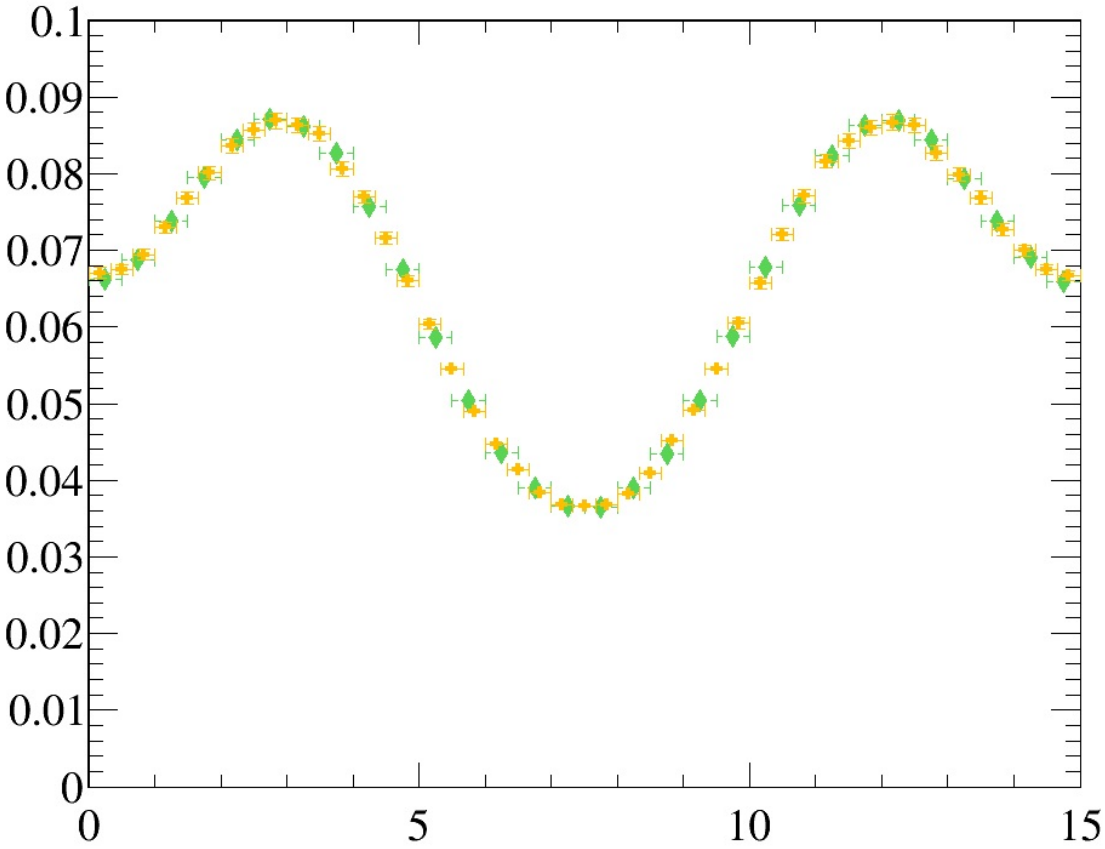
Simulated sample is a simlpe 2D-uniform distribution. Note that the *data distributions* are non-factorizeable, and simple 1D-reweightings here is not enought. In this example, for the first five iteration only 2D-reweighting `y:x` is applied, and then two 1D-reweighting `x` and `y` are added.

After the reweighting the simulated distributins are

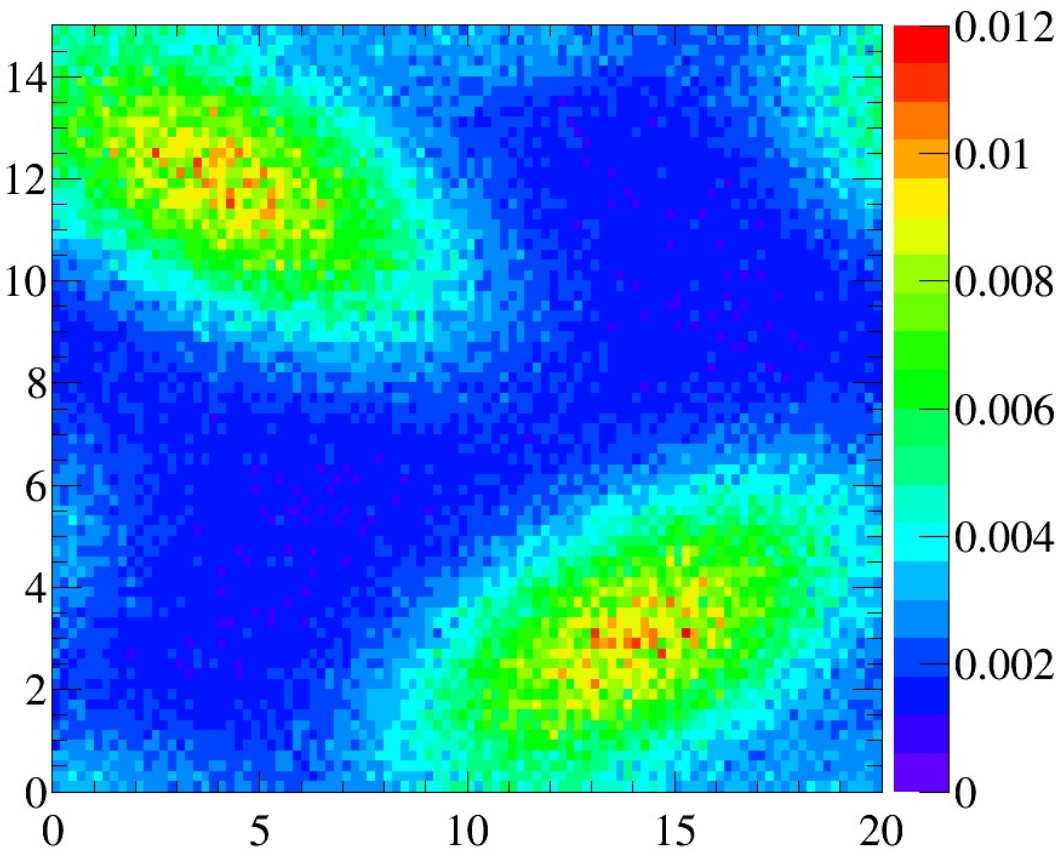
- for `x` -variable: *data distribution* (red filled circled) vs *simulated sample* (blue open squares)



- for `y` -variable: *data distribution* (green filled diamonds) vs *simulated sample* (orange filled swiss-crosses)



- for `y:x` -variables


0.012
0.01
0.008
0.006
0.004
0.002
0
14
12
10
8
6
4
2
0
0
5
10
15
20

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.