

# Table of Contents

The Ostap tutorials	1.1
Getting started	2.1
<a href="#">Values with uncertainties</a>	2.1.1
<a href="#">Simple operations with histograms</a>	2.1.2
<a href="#">Simple operations with trees</a>	2.1.3
<a href="#">Persistency</a>	2.1.4
Using RooFit	3.1
<a href="#">Models</a>	3.1.1
Contributing	4.1
Examples of formatting	4.2
<a href="#">Download PDF</a>	5.1

# The Ostap tutorials

build passing

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 uncertainties: `ValueWithError`

One of the central object in `ostap` is C++ class `Gaudi::Math::ValueWithError`, accessible in python via shortcut `VE`. This class stands for 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 'Error 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 expressions:

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

Note that for trivial cases the correlations are properly 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

## Histogram content

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

- by bin index, using operator `[]` : for 1D histogram 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-histogram can be done easily using native `slice` in python

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

For 2D and 3D-cases 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 defined according 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  $a \text{ oper } b$ , where:
  - $\text{oper}$  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 operations:

- `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 histograms
- `chi2` : bin-by-bin chi2-tension between two histograms
- ... 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 histogram (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 histogram to be rebinned
template = ... ## histograms that defined 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 binning, choose *template* histogram 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 coincide 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 histograms 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 addition to *trivial* scaling operations `h *= 3` and `h /= 10` there are several dedicated methods for scaling

- `scale` : it scales the histogram 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 histograms with non-uniform binning

### Density

There is method `density` that converts the histogram into *density* histogram. The density histogram (being interpreted as *function*) has unit integral. It is different from the simple rescaling for histograms 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 = ... ## distribution 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 indirectly 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 straightforward

```
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 histogram as function or e.g. in `ROOT.TH1.Fit` method for fitting of other histograms

## Efficiencies

There are several special cases to get the efficiency-histograms

```

accepted = ... ## histogram with accepted sample
rejected = ... ## histogram with rejected sample
total    = ... ## histogram 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 addition to the methods described above, few more sophisticated treatments of *binomial efficiencies* 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.agrestiCoulEff ( total ) ## only for natural histograms

```

For *natural* histograms only one can use even more [sophisticated methods](#), that evaluates the interval. Each method returns *graph*, and the graphs can be visualised 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 these 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 binning of the histogram that ensures almost equal bin populations. This task could be solved using `equal_bins` method

```

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

```







# Contributing

[ostap-tutorials](#) is an open source project, and we welcome contributions of all kinds:

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

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

## Getting Started

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

## Dependencies

To build the lessons locally, install the following:

1. [Gitbook](#)

Install the Gitbook plugins:

```
$ gitbook install
```

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

```
$ gitbook serve
```

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

# The title

## Learning Objectives

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

## Basic formatting

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

Have bulleted lists:

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

Use numbered lists:

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

## LaTeX

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

## Code highlighting

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

```
import sys

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

stderr_print("Hello world")
```

bash:

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

and more!

# Callouts

Prerequisites

- Prerequisite 1
- Prerequisite 2

Objectives

- Objective 1
- Objective 2

Challenge

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

- How to print?

Solution

The answer is:

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

Extra details that are hidden by default

Some extra details

Keypoints

- Summary point 1



- Summary point 2

## Quotes

This was said by someone

## Tables

Simple tables are possible

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

## Images



## Section types

This is a section

### Subsections

And a subsection

### Subsubsections

And a subsubsection