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, trees, frames, graphs etc.
  - operations and operators
  - iteration, element access, etc
  - extended functionality
- *decoration* of many basic `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 `Ostap::Math::ValutWithError`, accessible in python via shortcut `VE`. This class stands fot r a combination of the value with uncertainties:

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

Simple operations with histograms

Historgam content

`ostap.histos.histos` 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 conten
t
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 ostap.math.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+-err
or in original histo
h2 = histo.sample( accept = lambda s : s > 0 ) ##sample but require that sampled values a
re 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% reso
lution
```

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 `Ostap::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-we
ighted
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 pop
ulated bins
edges2 = fine_binned.equal_edges ( 10 , wmax = 5 ) ## try to fing binning with 10 almost
equally populated bins, but avoid bins wider than "wmax"
```

Operations with trees/chains

General

```
tree = ...
print tree.branches()
print tree.leaves()
print 'Number of entries %s' % len ( tree )
```

For large number of bracnhes...

For trees with very large number of bracnhes (*feature* of LHCb) one can improves printout:

```
from   ostap.logger.logger import multicolumn
print 'Branches: \n%s' % muhlticolumn ( tree.branches() )
```

Statistic for the given variable/expression

```
st1 = tree.statVar('m')
st2 = tree.statVar('m','pt>10')
st3 = tree.statVar('m/eff','(pt>10)*trg_eff')
```

The results are in a form of `WStatEntity`, *weighted* `StatEntity`)

```
ncorr = tree.sumVar('S_sw/eff','pt>10')
```

Also one can get statistics and covariances for the pair of variables/expressions:

```
s1 , s2 , cov2 = tree.statCov ( 'pt' , 'p' , 'pt>10' )
```

Or just simple

```
mn , mx = tree.minmax('1/eff')
```

Explicit loops

Explicit loops over the entries in tree/chain are trivial :

```
for i in  range(len(tree)) :
    tree.GetEntry(i)
    if tree.pt <  10 : continue
    print tree.m
```

But the direct looping looks a bit nicer:

```
for entry in tree :
    if entry.pt <  10 : continue
    print entry.m
```

Note that explciit loops are rather CPU-inefficient and slow. One can *drastically* improve performance by e.g embedding the cuts in iterator

```
for entry in tree.withCuts('pt>10') :
    print entry.m
```

One can also specify `first` and `last` entries and display the progress bar

```
for entry in tree.withCuts('pt>10', last = 10000 , progress =  True ) :
    print entry.m
```

Projections

```
h1 = ...
r  = tree.project ( h1 , 'mass' , 'pt>10' )
```

For loooong chains or huge trees...

The module `ostap.paralell.kisa` provides nice functionality for parallell processing of large chains or huge trees for *projections*

```
h1 = ...
long_chain =  ...
huge_tree   =  ...
import ostap.parallel.kisa
r1 = long_chain.pproject ( h1 , 'mass' , 'pt>10' )
r2 = huge_tree .pproject ( h1 , 'mass' , 'pt>10' )
```

For long chains it makes parallelizaion on *per-tree* level, and for huge trees it split the tree into chunks and parallelization is applied on *per-chunk* level.

Data, Data2 and DataAndLumi

There is useful way to collect many ROOT files into single chains, avoiding non-existent, broken and invalid trees (that is not so rare for the outptu of Ganga)

```
from   ostap.trees.data import DataAndLumi as Data
ganga = '/afs/cern.ch/work/i/ibelyaev/public/GANGA/workspace/ibelyaev/LocalXML'
patterns_Y = [
  ganga + '/319/*/output/CY.root' , ## 2k+11,down
  ganga + '/320/*/output/CY.root' , ## 2k+11,up
  ganga + '/321/*/output/CY.root' , ## 2k+12,down
  ganga + '/322/*/output/CY.root' , ## 2k+12,up
 ]
data_D0Y = Data ( 'YD0/CY' , patterns_Y )
print data_D0Y
chain = data_D0Y.chain
lumi  = data_D0Y.getLumi()
```

Or they can be accumulated separately, and combined later:

```
from   ostap.trees.data   import DataAndLumi as Data
ganga  = '/afs/cern.ch/work/i/ibelyaev/public/GANGA/workspace/ibelyaev/LocalXML'
d2011d = Data( 'YD0/CY' ,  ganga + '/319/*/output/CY.root' ) ## 2k+11,down
d2011u = Data( 'YD0/CY' ,  ganga + '/320/*/output/CY.root' ) ## 2k+11,up
d2012d = Data( 'YD0/CY' ,  ganga + '/321/*/output/CY.root' ) ## 2k+12,down
d2012u = Data( 'YD0/CY' ,  ganga + '/322/*/output/CY.root' ) ## 2k+12,up
d2011  = d2011d + d2011u
d2012  = d2012d + d2012u
runI   = d2011  + d2011
```

Persistency

`ostap.io.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.io.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.io.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.io.rootshelve import RooOnlyShelf
db = RooOnlyShelf('mydb.root','c')
h1 = ...
db ['histogram'] = h1
db.ls()
```

`RootShelve`

The module `ostap.io.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.io.rootshelve import RootShelf
db = RootShelf('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-nega
tive decreasing Bernstein sum
m2  = histo.monothonic    ( 6 , increasing = True  ) ## parameterize as degree-6 non-nega
tive increasing Bernstein sum
c1  = histo.convex        ( 6 , increasing = False , convex = True  ) ## parameterize as
degree-6 non-negative decreasing convex  Bernstein sum
c2  = histo.convex        ( 6 , increasing = False , convex = False ) ## parameterize as
degree-6 non-negative decreasing concave Bernstein sum
c3  = histo.convex        ( 6 , increasing = True  , convex = True  ) ## parameterize as
degree-6 non-negative increasing convex  Bernstein sum
c4  = histo.convex        ( 6 , increasing = True  , convex = False ) ## parameterize as
degree-6 non-negative increasing concave Bernstein sum
cc1 = histo.convexpoly    ( 6 ) #  parameterize as degree-6 non-negative convex  Bernstei
n sum
cc2 = histo.concavepoly   ( 6 ) #  parameterize as degree-6 non-negative concave Bernstei
n sum
```

Various types of *splines* are also provided

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

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 Bernst
ein sum
r2  = histo.pdf_positiveeven       ( 5 ) ## parameterize and non-negative degree-5 even B
ernstein polynomial
r3  = histo.pdf_increasing         ( 5 ) ## parameterize and non-negative degree-5 increa
sing Bernstein polynomial
r4  = histo.pdf_decreasing         ( 5 ) ## parameterize and non-negative degree-5 decrea
sing 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 concav
e increasing Bernstein polynomial
r8  = histo.pdf_concave_decreasing ( 5 ) ## parameterize and non-negative degree-5 concav
e decreasing Bernstein polynomial
r9  = histo.pdf_concavepoly        ( 5 ) ## parameterize and non-negative degree-5 concav
e 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*

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.