MPF Documentation

hans

Sep 04, 2019

Contents:

MPF is a plotting framework that is meant to simplify the creation of typical "ATLAS-style" plots using pyROOT. Setup instructions and the code itself can be obtained via https://gitlab.com/nikoladze/MPF.

CHAPTER 1

Plotting from histograms

Plots can currently be created either from ROOT histograms. For example:

```
from MPF.plot import Plot

p = Plot()
p.registerHist(hist1, style="background", process="Bkg1")
p.registerHist(hist2, style="background", process="Bkg2")
p.registerHist(hist3, style="signal", process="Signal")
p.registerHist(hist4, style="data", process="Data")
p.SaveAs(outputFilename)
```

See *plotStore* for details and examples.

CHAPTER 2

Plotting from ntuples

Plots can also be generated from flat ntuples (ROOT TTree). There is an extensive interface wrapping around TTree::Draw to ease the creation of multiple similiar histograms from different ntuples. For example:

```
from MPF.treePlotter import TreePlotter

tp = TreePlotter()
tp.addProcessTree("Bkg1", rootfile1, treename1, style="background")
tp.addProcessTree("Bkg2", rootfile2, treename2, style="background")
tp.addProcessTree("Signal", rootfile3, treename3, style="signal")
tp.addProcessTree("Data", rootfile4, treename4, style="data")
tp.plot(outputFilename, varexp=var, xmin=xmin, xmax=xmax, nbins=nbins)
```

See *treePlotter* for details and examples.

Also, there is an option to create multiple histograms from a single TTree by looping the tree only once. For example:

```
tp.registerPlot(outputFilename1, varexp=var1, xmin=xmin1, xmax=xmax1, nbins=nbins1)
tp.registerPlot(outputFilename2, varexp=var2, xmin=xmin2, xmax=xmax2, nbins=nbins2)
tp.plotAll()
```

CHAPTER 3

Global options

Most options are directly passed to the Functions/Classes that create the plots. A few, more global options can also be set via the *MPF.globalStyle* module.

CHAPTER 4

Memoization

All histograms that are created from ntuples are cached across multiple executions (using the meme module), so once created, modifications to the plot (e.g. style) can be made without having to loop the ntuples again. This is very useful, however currently meme is not thread-safe - so in case you are about to run MPF in parrallel in the same folder you should deactivate the meme module:

```
from MPF import meme
meme.setOptions(deactivated=True)
```

By default the cache will be stored in a folder `_cache` in the directory your code is executed. This can be changed by:

```
meme.setOptions(overrideCache=myCachePath)
```

To enforce re-execution of cached functions either delete the cache folder or use:

```
meme.setOptions(refreshCache=True)
```

CHAPTER 5

Verbosity/Logging

MPF uses the builtin logging from python. Both MPF and meme loggers are preconfigured if no handlers exist. You can set the logging levels by retrieving the loggers. For example to deactivate the INFO messages:

```
import logging
logging.getLogger("MPF").setLevel(logging.WARNING)
logging.getLogger("meme").setLevel(logging.WARNING)
```

The submodules of MPF use child loggers of the `MPF` logger. The naming scheme is `MPF.MPF.<submodule-name>`. For example to activate the DEBUG messages only for the *`histProjector`* module do:

```
logging.getLogger("MPF.MPF.histProjector").setLevel(logging.DEBUG)
```

If you want to configure your own loggers you can do this by either adding handlers to the MPF/meme or root logger prior to any import of MPF modules. Alternatively you can (also prior to any import of MPF modules) add a NullHandler to the MPF logger and configure logging at any time later:

```
logging.getLogger("MPF").addHandler(logging.NullHandler())
```

CHAPTER 6

Table of contents

6.1 MPF package

6.1.1 Submodules

MPF.IOHelpers module

class MPF.IOHelpers.**Capturing**
Bases: list

context manager to capture stdout of the encapsulated command(s). Might be useful to get some Information that ROOT won't give us, but print. See http://stackoverflow.com/questions/16571150/how-to-capture-stdout-output-from-a-python-function-call#16571630 NOT WORKING YET SINCE IT DOESN'T CAPTURE OUTPUT FROM C++ CALLS

class MPF.IOHelpers.**ROpen**(*fileName*, *mode='READ'*)
Bases: object

context manager to open root files

class MPF.IOHelpers.**cd**(*newPath*)
Context manager for changing the current working directory

class MPF.IOHelpers.**workInTempDir**(*baseDir=None*, *skipCleanup=False*, *prefix='tmp_'*, *cleanAtExit=False*)
Context manager for changing to a temporary directory that will be deleted afterwards. The given dir will be the base directory in which the tempdir is created. If not given, the system tmp directory will be used. If *skipCleanup* is given the directory is not deleted afterwards. Use *prefix* to control the naming format. If *cleanAtExit* is set the tmp directory is cleaned at exit of the script, not when exiting the context.

cleanup()

MPF.arrow module

class MPF.arrow.**Arrow**(*x1*, *y1*, *x2*, *y2*, *size*, *option*)
Bases: ROOT.TArrow

draw(**kwargs*)

MPF.atlasStyle module

This file contains functions to set the recommended default settings for the ROOT plotting style for ATLAS plots

MPF.atlasStyle.**getAtlasStyle**(*th2=False*)

class MPF.atlasStyle.**info**

isAtlasStyle = False

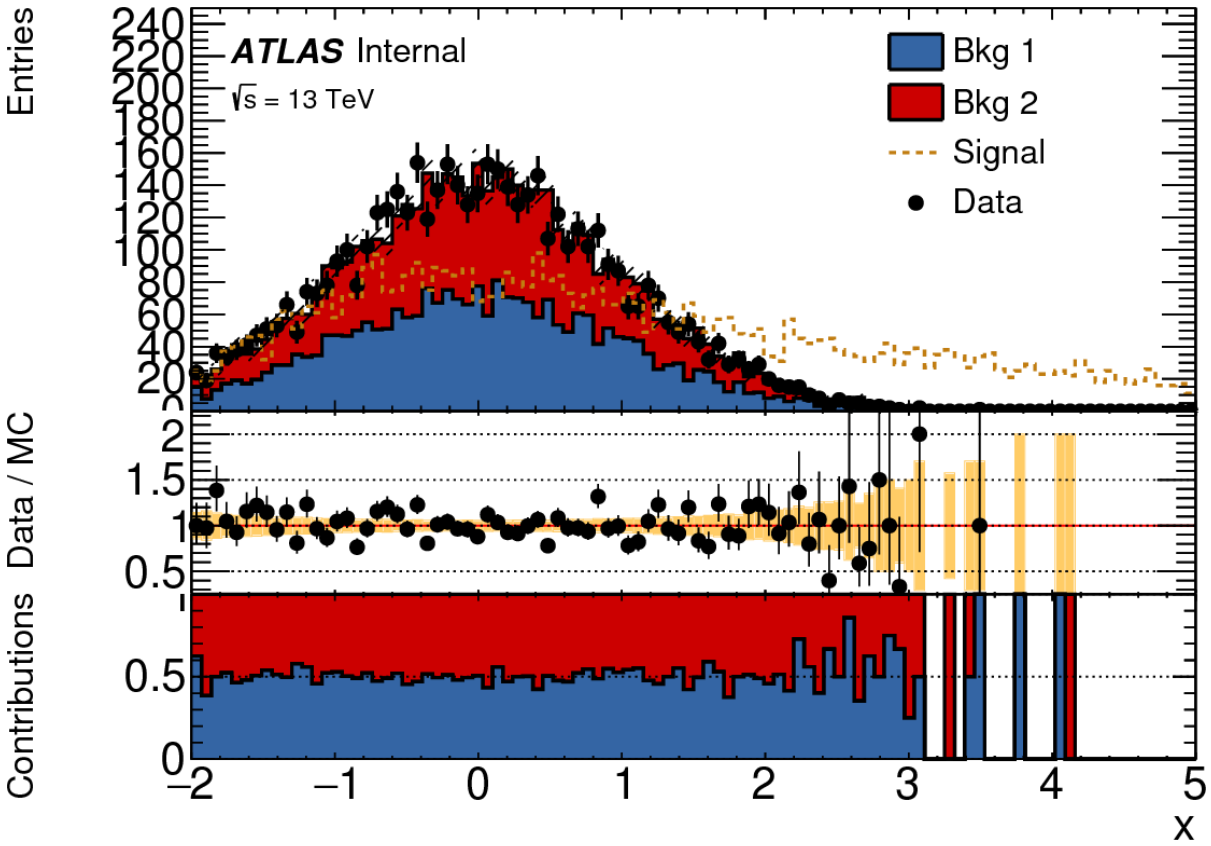
MPF.atlasStyle.**setAtlasStyle**(*th2=False*, *reset=False*, *batch=True*)

MPF.bgContributionPlot module

Similiar to *DataMCRatioPlot()*, but also shows the relative contributions of the background processes in each bin in a 3rd pad.

Example

```
#!/usr/bin/env python

import ROOT
from MPF.bgContributionPlot import BGContributionPlot
from MPF.atlasStyle import setAtlasStyle

setAtlasStyle()
bkg1 = ROOT.TH1F("bkg1", "", 100, -2, 5)
bkg2 = ROOT.TH1F("bkg2", "", 100, -2, 5)
data = ROOT.TH1F("data", "", 100, -2, 5)
signal = ROOT.TH1F("signal", "", 100, -2, 5)

bkg1.FillRandom("gaus")
bkg1.Scale(0.5)
bkg2.FillRandom("gaus")
bkg2.Scale(0.5)
data.FillRandom("gaus")
signal.FillRandom("landau")

p = BGContributionPlot(xTitle="x")
p.registerHist(bkg1, style="background", process="Bkg 1")
p.registerHist(bkg2, style="background", process="Bkg 2")
p.registerHist(data, style="data", process="Data")
p.registerHist(signal, style="signal", process="Signal")
p.saveAs("plot.pdf")
```



class MPF.bgContributionPlot.**BGContributionPlot**(*bgContributionPadTitle='Contributions'*, *noData=False*, *ratioTitle='Data / MC'*, ***kwargs*)

Bases: *MPF.plotStore.PlotStore*

Parameters

- **noData** – Don't show a data/MC ratio
- **bgContributionPadTitle** – Title on the pad showing the relative contributions

Overwrites the defaults for the following *PlotStore()* parameters:

Parameters

- **ignoreNumErrors** – default: False
- **ignoreDenErrors** – default: True
- **ratioMode** – default: "rawpois"
- **ratioUp** – default: 2.25
- **ratioDown** – default: 0.25
- **ratioTitle** – Title for the ratio pad if data/MC ratio is shown (default: "Data / MC")

For further options see *PlotStore()*

saveAs(*path*, ***kwargs*)

Save the canvas. Arguments are passed to *MPF.canvas.Canvas.saveAs()*

MPF.canvas module

class MPF.canvas.**Canvas**(*splitting*, *xAxisLabelsOption=None*)
Bases: ROOT.TCanvas

drawPads(***kwargs*)

saveAs(*path*, ***kwargs*)
Save the canvas, takes care of drawing everything and calls the ROOT Print function

Parameters

- **promptDir** – if save path does not exist, ask if it should be created. If set False, the dir is created without asking (default: True)
- **noSave** – if set True the canvas is only drawn, but not saved and left open, so custom modifications can be made (default: False)

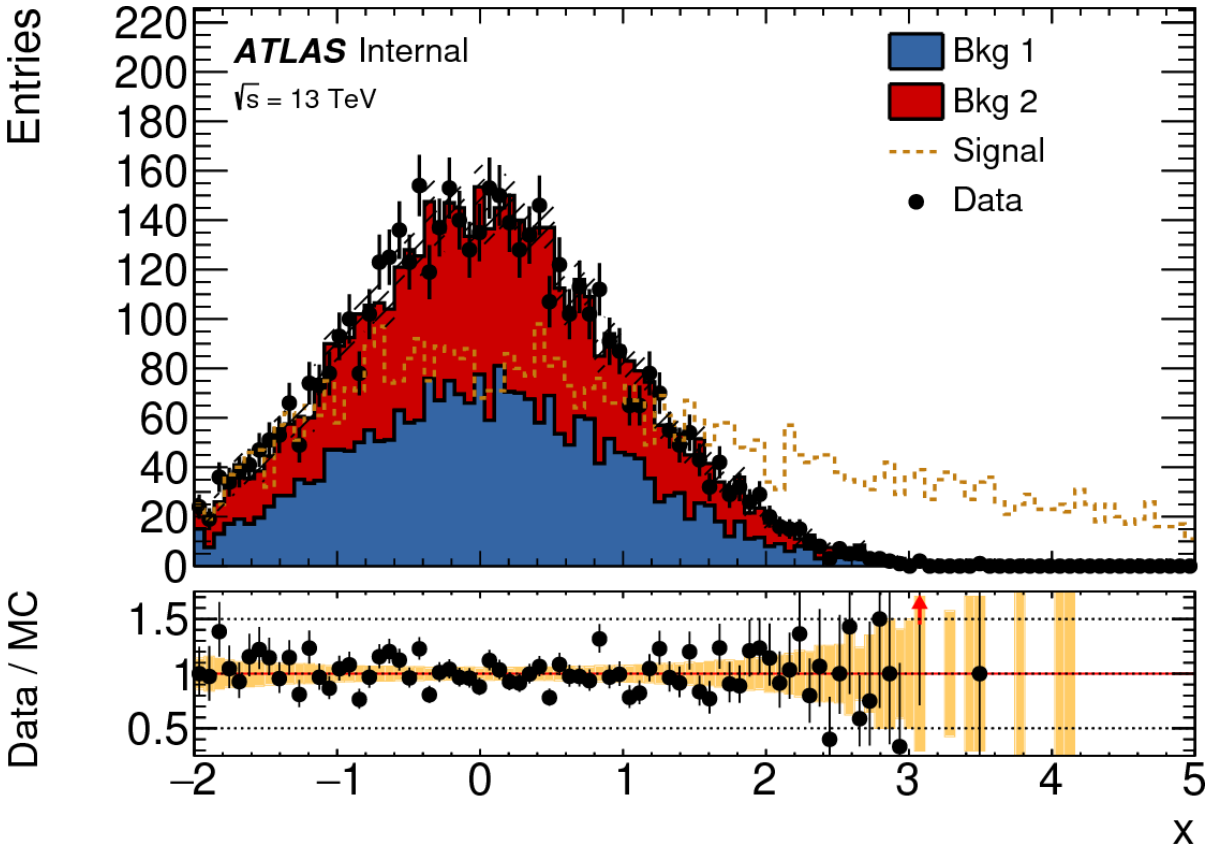
setup()

MPF.dataMCRatioPlot module

Similiar to *plot()*, but also shows a ratio of data and the total background in a bottom pad.

Example

```
#!/usr/bin/env python

import ROOT
from MPF.dataMCRatioPlot import DataMCRatioPlot
from MPF.atlasStyle import setAtlasStyle

setAtlasStyle()
bkg1 = ROOT.TH1F("bkg1", "", 100, -2, 5)
bkg2 = ROOT.TH1F("bkg2", "", 100, -2, 5)
data = ROOT.TH1F("data", "", 100, -2, 5)
signal = ROOT.TH1F("signal", "", 100, -2, 5)

bkg1.FillRandom("gaus")
bkg1.Scale(0.5)
bkg2.FillRandom("gaus")
bkg2.Scale(0.5)
data.FillRandom("gaus")
signal.FillRandom("landau")

p = DataMCRatioPlot(xTitle="x")
p.registerHist(bkg1, style="background", process="Bkg 1")
p.registerHist(bkg2, style="background", process="Bkg 2")
p.registerHist(data, style="data", process="Data")
p.registerHist(signal, style="signal", process="Signal")
p.saveAs("plot.pdf")
```



class MPF.dataMCRatioPlot.**DataMCRatioPlot**(*ratioTitle='Data / MC'*, ***kwargs*)

Bases: *MPF.plotStore.PlotStore*

Overwrites the defaults for the following *PlotStore()* parameters:

Parameters

- **ignoreNumErrors** – default: False
- **ignoreDenErrors** – default: True
- **ratioMode** – default: "rawpois"
- **ratioTitle** – default: "Data / MC"

For further options see *PlotStore()*

saveAs(*path*, ***kwargs*)

Save the canvas. Arguments are passed to *MPF.canvas.Canvas.saveAs()*

MPF.errorBands module

class MPF.errorBands.**AE**(*histo=None*, *relErr=False*, *customDrawString=None*)

Bases: ROOT.TGraphAsymmErrors

addRatioArrows(*up*, *low*)

Add a list of arrows to a ratio graph. The ratio thresholds for which an arrow is drawn are given by up, low

draw(*drawString=''*, ***kwargs*)

static makeArrow(*bincenter*, *up*, *low*, *pointUp=True*)

truncateErrors()

MPF.errorBands.**getPoissonRatio**(*num*, *den*, *ignoreDenErrors=True*, *ignoreNumErrors=False*)
Returns a ratio graph of 2 independend histograms. For the error calculation the Yield per bin is assumed to be the mean of a poisson distribution (histogram error is ignored).

Optional keyword arguments: ignoreDenErrors: Ignore the denominator for error calculation (default True) ignoreNumErrors: Ignore the numerator for error calculation (default False)

MPF.globalStyle module

Some fixed settings to make the plots look nice by default. You can import this module and change some settings if you like - for Example:

```
import MPF.globalStyle as gst

gst.ratioErrorBandFillStyle = 3354
gst.ratioErrorBandColor = ROOT.kBlack
gst.drawATLASLabel = False
```

Or better use *useOptions* which will raise an AttributeError if options are misspelled:

```
from MPF.globalStyle import useOptions

useOptions(ratioErrorBandFillStyle = 3354,
           ratioErrorBandColor = ROOT.kBlack,
           drawATLASLabel = False)
```

Alterantively you can also temporarily set options - e.g. for one plot - by using *useOptions* as a context manager:

```
from MPF.globalStyle import useOptions

with useOptions(ratioErrorBandFillStyle=3354, ratioErrorBandColor=ROOT.kBlack):
    p.plot("output.pdf")
```

When *TreePlotter* is used, a dictionary of globalStyle options can be directly passed to be used in one or multiple plots.

MPF.globalStyle.**CMELabelTextSize = 18**
Text size for CMELabel

MPF.globalStyle.**CMELabelX = 0.352**
Default x position for CMELabel

MPF.globalStyle.**CMELabelY = 0.83**
Default y position for CMELabel

MPF.globalStyle.**TLineColor = 1**
Default TLine color (not really used yet)

MPF.globalStyle.**TLineStyle = 2**
Default TLine style (not really used yet)

MPF.globalStyle.**TLineWidth = 2**
Default TLine width (not really used yet)

MPF.globalStyle.**atlasLabelDelX = 0.1**
Measure for distance between “ATLAS” and the text

MPF.globalStyle.**atlasLabelTextSize = 0.04**
Text size for atlasLabel

MPF.globalStyle.**atlasLabelX = 0.19**
Default x position for atlasLabel

MPF.globalStyle.**atlasLabelY = 0.88**
Default y position for atlasLabel

MPF.globalStyle.**bottomMargin1Pad = None**
If given, explicitely set bottom margin in main pad for 1 Pad case

MPF.globalStyle.**bottomPadSize3Pad = 0.3**
Relative size of the second bottom pad for the splitting with 3 pads

MPF.globalStyle.**bottomPadsNoExponent = True**
No Exponent on y-axis of bottom pads - this won't look nice if there are large numbers involved (and no log scale is used), but the exponent doesn't fit there

MPF.globalStyle.**canvasHeight = 600**
vertical plot size - changing this might require to change also some other options

MPF.globalStyle.**canvasWidth = 800**
horizontal plot size - changing this might require to change also some other options

MPF.globalStyle.**customTextFont = 43**
Font of custom text labels

MPF.globalStyle.**customTextSize = 18**
Size of custom text labels

MPF.globalStyle.**cutLineArrowPos = 0.7**
Position of the cutline arrow relative to its height

MPF.globalStyle.**cutLineArrows = False**
Draw arrows indicating selected region for cut lines

MPF.globalStyle.**cutLineColor = 1**
TLine color for cut lines

MPF.globalStyle.**cutLineHeight = 0.75**
Height of cutlines relative to the pad min/max

MPF.globalStyle.**cutLineStyle = 2**
TLine style for cut lines

MPF.globalStyle.**cutLineWidth = 2**
TLine width for cut lines

MPF.globalStyle.**defaultLogyYmin = 0.1**
Default y mininum on log scale plots (in case not explicitely set or automatically determined)

MPF.globalStyle.**drawATLASLabel = True**
Draw AtlasLabel in plots?

MPF.globalStyle.**infoLabelTextSize = 18**
Text size for InfoLabel

MPF.globalStyle.**infoLabelX = 0.19**
Default x position for processLabel

MPF.globalStyle.**infoLabelY = 0.78**
Default y position for processLabel

MPF.globalStyle.**labelFont = 43**
Font number for axis and labels (43 is fixed size, atlasStyle default is taken if set to None)

MPF.globalStyle.**labelFontSize = 30**
Font size for axis and labels (atlasStyle default taken if set to None)

MPF.globalStyle.**legendBorderSize = 0**
Useful to set this nonzero for debugging

MPF.globalStyle.**legendEventCountFormat = ' ({:.1g})'**
Format for showing event counts in the legend (if set)

MPF.globalStyle.**legendEventCountFormatRaw = ' ({:.0g})'**
Format for showing (raw) event counts in the legend (if set)

MPF.globalStyle.**legendFont = 42**
Legend Font

MPF.globalStyle.**legendLongTitleThreshold = 11**
Length starting from which a title in the legend is considered long and text will be scaled

MPF.globalStyle.**legendShowScaleFactors = True**
Show scale factors of processes in legend?

MPF.globalStyle.**legendTextSize = 0.04**
Default Text size for legend (if None, text will be scaled to fit on legend size)

MPF.globalStyle.**legendXMin = 0.7**
Legend default x position

MPF.globalStyle.**legendYMax = 0.92**
Legend default y position

MPF.globalStyle.**lumiLabelTextSize = 18**
Text size for lumiLabel

MPF.globalStyle.**lumiLabelX = 0.19**
Default x position for lumiLabel

MPF.globalStyle.**lumiLabelY = 0.83**
Default y position for lumiLabel

MPF.globalStyle.**mainPadSize2Pad = 0.7**
Relative size of the mainpad for the splitting with 2 pads

MPF.globalStyle.**mainPadSize3Pad = 0.5**
Relative size of the mainpad + first bottom pad for the splitting with 3 pads

MPF.globalStyle.**mainPadTopMargin = 0.06**
Top margin in main pad (relative to absolute canvas height)

MPF.globalStyle.**maximumWithErrors = True**
Loop over bins to find maximum with errors for histograms

MPF.globalStyle.**mergeCMEIntoLumiLabel = False**
Only plot a merged Lumi and CME label without "#int L dt" (new ATLAS convention)

MPF.globalStyle.**minimumWithErrors = False**
Loop over bins to find minimum with errors for histograms

MPF.globalStyle.**noLinesForBkg = False**
Set true to set LineWidth to 0 for bkg hists (get rid of the black lines in between processes in stack)

MPF.globalStyle.**poissonIntervalDataErrors = False**
Use the asymmetric 68% poisson interval for drawing data errors

MPF.globalStyle.**processLabelTextSize = 18**
Text size for the ProcessLabel

MPF.globalStyle.**processLabelX = 0.165**
Default x position for infoLabel

MPF.globalStyle.**processLabelY = 0.96**
Default y position for infoLabel

MPF.globalStyle.**ratioBottomMargin = 0.4**
bottom margin in ratio plots

MPF.globalStyle.**ratioErrorBandColor = 796**
Color in ratio error bands

MPF.globalStyle.**ratioErrorBandFillStyle = 1001**
Fill style for ratio error bands

MPF.globalStyle.**ratioPadGridy = 1**
draw vertical lines on the yAxis ticks of the ratio pad This is used by default for the first bottom pad - assuming it will be some ratio like graph in there (individual plots can overwrite this option)

MPF.globalStyle.**ratioPadNDivisions = 504**
Axis tick divisions on the ratio pads n = n1 + 100*n2 + 10000*n3

MPF.globalStyle.**ratioPlotMainBottomMargin = 0.04**
Bottom Margin on the main pad in ratio plots

MPF.globalStyle.**ratioXtitleOffset = 3**
Ratio xTitle offset

MPF.globalStyle.**thankYou = True**
Thank MPF at exit of your script

MPF.globalStyle.**thirdPadGridy = 1**
draw vertical lines on the yAxis ticks of the third bottom pad (individual plots can overwrite this option)

MPF.globalStyle.**totalBGErrorColor = <Mock id='139831485890448'>**
Color for totalBG error bands

MPF.globalStyle.**totalBGFillStyle = 3354**
Fill style for totalBG error

MPF.globalStyle.**totalBGLineWidth = 3**
In case noLinesForBkg is set, the totalBG hist will get a line of this thickness

class MPF.globalStyle.**useOptions**(*optObject=<module 'MPF.globalStyle' from '/home/docs/checkouts/readthedocs.org/user_builds/mpf-plotting/checkouts/latest/pythonpath/MPF/globalStyle.pyc'>*, ***kwargs*)
Set options of this module. The advantage w.r.t. explicitely setting them is that an AttributeError will be raised if options are misspelled.

Can also be used as a context manager for temporarily setting options.

MPF.globalStyle.**xTitleOffset3Pad = 2.4**
Ratio xTitle offset

MPF.globalStyle.**yTitleOffset = 1.6**
default yTitle offset for all pads

MPF.globalStyle.**yTitleScale2Pad = 0.9**
Scale y-Axis Titles for 3 pad plots

MPF.globalStyle.**yTitleScale3Pad = 0.8**
Scale y-Axis Titles for 3 pad plots

MPF.histProjector module

MPF.histProjector.**HP**
alias of *MPF.histProjector.HPDefaults*

class MPF.histProjector.**HPDefaults**

aliases = None

autoBinning = False

binLowEdges = None

cut = '1'

nbins = 1

varexp = '1'

weight = '1'

xmax = 1.5

xmin = 0.5

class MPF.histProjector.**HistProjector**
Helper class for projecting histograms from trees. Will mostly be used by *ProcessProjector()* and classes inheriting from it.

fillHists(*compile=False*)
Fill registered histograms using *MultiHistDrawer()*

static getHash(***kwargs*)
Returns a hash of the given kwargs. Only takes kwargs - to be independend of how exactly the functions are called (the kwargs are sorted)

static getMHDKwargs(***kwargs*)
Returns only the kwargs which are supposed to be put into multihistdrawer. Also applies defaults to have a unique description.

getTH1Path(***kwargs*)
Projects a TTree from a path or multiple paths (ROOT file(s)) into a histogram. Returns a prefilled histogram if it exists (see *registerTH1Path()*).

Parameters

- **treeName** – name of the TTree inside the ROOT file(s)
- **cut** – selection expression
- **paths** – path(s) to the ROOT file(s) containing the TTree
- **varexp** – expression to be filled into the histogram, default "1"
- **weight** – additional expression, to be multiplied with cut, default "1"
- **xmin** – minimum value to be filled into the histogram, default 0.5

- **xmax** – maximum value to be filled into the histogram, default 1.5
- **nbins** – number of bins between xmin and xmax, default 1
- **binLowEdges** – list of low edges for variable binning, the first value is the lower bound of the first bin, the last value the upper bound of the last bin. If given, xmin, xmax and nbins are ignored.
- **autoBinning** – use "free" TTree::Draw instead of fixed binning. If set True, all other binning options are ignored, default False

getTH1PathTrees (*paths_treeNames*, *cut*, ***kwargs*)

Returns a merged hist for list of different trees in different files. The list is expected in the following format:

```
[
  (path1, treeName1),
  (path2, treeName2),
  ...
]
```

See *getTH1Path()* for parameters.

getYieldPath (***kwargs*)

Projects a TTree from a path and returns a tuple of selected entries, the weighted entries and the quadratically summed error

Parameters

- **paths** – path(s) to the ROOT file(s) containing the TTree
- **treeName** – name of the TTree inside the ROOT file(s)
- **cut** – selection expression
- **weight** – additional expression, to be multiplied with cut, default "1"

getYieldsDict (*treeName*, *cutsDict*, **paths*, ***kwargs*)

Fetches yields for each selection in cutsDict and returns a dictionary containing the unweighted, weighted yields and the error.

Parameters **weight** – if given, apply this weight for all selections

getYieldsHist (*treeName*, *cutsDict*, **paths*, ***kwargs*)

Fetches yields for each selection in cutsDict and fills a histogram with one bin per selection - carrying the dict key as label.

Parameters **weight** – if given, apply this weight for all selections

static projectTH1PathStatic (**args*, ***kwargs*)

Projects a TTree from a path or multiple paths (ROOT file(s)) into a histogram. Recommended usage via *getTH1Path()*

static projectTH1Static (*tree*, *cut='1'*, *weight='1'*, *xmin=0.5*, *xmax=1.5*, *nbins=1*, *binLowEdges=None*, *autoBinning=False*, *varexp='1'*, *aliases=None*)

Base functon to project from a TTree into a histogram Recommended usage via *getTH1Path()*

static projectTH2Static (*tree*, *cut*, *weight*, ***kwargs*)

registerTH1Path (*treeName*, *cut*, **paths*, ***kwargs*)

Register a histogram to be projected later. See *getTH1Path()* for parameters.

All registered histograms are filled when *fillHists()* is called.

registerYieldPath(*treeName*, *cut*, **paths*, ***kwargs*)
Register a histogram for a treeName and path(s) to be prefilled later (for retrieving a yield). Can be fetched later by calling getYieldPath

registerYieldsDict(*treeName*, *cutsDict*, **paths*, ***kwargs*)
Register a yields dict. Can be fetched later by calling *getYieldsDict()* or *getYieldsHist()*

registerYieldsHist(*treeName*, *cutsDict*, **paths*, ***kwargs*)
Register a yields dict. Can be fetched later by calling *getYieldsDict()* or *getYieldsHist()*

setAlias(*aliasName*, *aliasFormula*)
Define an alias that is added to each tree that is drawn. Currently not working for multiHistDraw (*fillHists()*).

exception MPF.histProjector.**TTreeProjectException**
Bases: exceptions.Exception

exception MPF.histProjector.**noTreeException**
Bases: exceptions.Exception

MPF.histograms module

class MPF.histograms.**Graph**(***kwargs*)
Bases: object

draw(***kwargs*)

setColor()

class MPF.histograms.**Histogram**
Bases: object

add(*other*)

addOverflowToLastBin()
Add histograms overflow bin content (and error) to the last bin

addSystematicError(**hists*, ***kwargs*)
Add systematic variations to the errorband based on given variational histogram(s).

Parameters

- **hists** – one ore more histograms to be added
- **mode** – how to add and symmetrise the errors?
  - symUpDown (default): independently add up and down variations quadratically and symmetrise afterwards
  - largest: also add quadratically up and down variations, but then use the max(up, down) as the error

clone()
Clones a histogram. Until i find out how to do this properly (e.g. with deepcopy) do some stuff manually here

static cloneAttributes(*tohist*, *fromhist*)

color

createPoissonErrorGraph()

draw(*drawString=''*)

fillColor

firstDraw(***kwargs*)

getXTitle()

getYTitle()

lineColor

markerColor

markerStyle

overflow()

rebin(*rebin=1*)

setColorAndStyle()

truncateErrors(*value=0*)

underflow()

yieldBinNumbers(*overFlow=False*, *underFlow=False*)

class MPF.histograms.**HistogramD**(*histogram*, ***kwargs*)
Bases: ROOT.TH1D, *MPF.histograms.Histogram*

class MPF.histograms.**HistogramF**(*histogram*, ***kwargs*)
Bases: ROOT.TH1F, *MPF.histograms.Histogram*

class MPF.histograms.**HistogramStack**(*histogram*)
Bases: ROOT.THStack

Add(*hist*)

add(*hist*)

checkSet(*attr*, *to*)

draw(*drawString=''*)

firstDraw(***kwargs*)

getXTitle()

getYTitle()

class MPF.histograms.**WrapTGraphAsymmErrors**(*graph*, ***kwargs*)
Bases: ROOT.TGraphAsymmErrors, *MPF.histograms.Graph*

MPF.histograms.**getHM**(*histogram*)

MPF.labels module

class MPF.labels.**ATLASLabel**(*text='Internal'*, *color=<Mock id='139831485820624'>*, *scale=1.0*, *xOffset=0.0*, *yOffset=0.0*)
Bases: object

The ATLAS Label

Parameters

- **text** – Text next to "ATLAS"
- **color** – Text color

- **scale** – Scale overall text size by this factor
- **xOffset** – Shift by this amount in x-Direction
- **yOffset** – Shift by this amoutn in y-Direction

draw(**args*, ***kwargs*)

class MPF.labels.**CMELabel**(*cme=13*, *unit='TeV'*, *scale=1.0*, *xOffset=0.0*, *yOffset=0.0*)
Bases: object

Label showing the center of mass energy

draw(**args*, ***kwargs*)

class MPF.labels.**InfoLabel**(*text=''*, *scale=1.0*, *xOffset=0.0*, *yOffset=0.0*)
Bases: object

Label used to provide more information, e.g. 'Normalized to unity'. Shown underneath the Lumilabel.

draw(**args*, ***kwargs*)

class MPF.labels.**LumiLabel**(*lumi=1.0*, *unit='fb^{-1}'*, *scale=1.0*, *xOffset=0.0*, *yOffset=0.0*, *cme=13*, *cmeUnit='TeV'*)
Bases: object

Label showing the luminosity

draw(**args*, ***kwargs*)

class MPF.labels.**ProcessLabel**(*text=''*, *scale=1.0*, *xOffset=0.0*, *yOffset=0.0*)
Bases: object

Label shown at the very top of the histogram. Usually used to show the signal process, e.g. #tilde(g)-#tilde(g) #rightarrow qqWW#chi#chi

draw(**args*, ***kwargs*)

MPF.labels.**scaleYPosTopMargin**(*y*)
Rescale the y position (NDC) in current pad to have the same distance to the top in different pad heights

MPF.legend module

class MPF.legend.**Legend**(*scaleLegend=1*, *scaleLegendX=None*, *scaleLegendY=None*, *drawstring=''*, *nColumns=1*, *xOffset=0*, *yOffset=0*, *forceDynamicFontSize=False*, *noDynamicFontSize=False*, *addEventCount=False*, *eventCountCutflow=False*, *eventCountGen=False*)
Bases: ROOT.TLegend

Parameters

- **scaleLegend** – Scale Legend by this factor
- **scaleLegendX** – Scale Legend by this factor in x-direction
- **scaleLegendY** – Scale Legend by this factor in x-direction
- **drawstring** – Drawstring for ROOT TLegend
- **nColumns** – Number of columns
- **xOffset** – Shift legend in x direction
- **yOffset** – Shift legend in y direction
- **forceDynamicFontSize** – Always use dynamic font size (scale text to fit legend)

- **noDynamicFontSize** – Never use dynamic font size (by default used for legends containing long titles)
- **addEventCount** – Add the total event counts to the legend
- **eventCountCutflow** – Use the content of the last bin instead of the integral when showing total event counts
- **eventCountGen** – In addition to the integral, also show the number of raw events

addEntry(*obj*, *option*)

addEventCount(*label*, *obj*)

draw()

hasLongTitleEntry()

MPF.line module

class MPF.line.**CutLine**(**args*, ***kwargs*)
Bases: *MPF.line.Line*

draw(***kwargs*)

class MPF.line.**Line**(**args*, ***kwargs*)
Bases: ROOT.TLine

draw(***kwargs*)

MPF.multiHistDrawer module

This module tries to implement a TTree::Draw like functionality with the addition that multiple histograms can be filled at once.

Note: For most use cases it is better to use the functionality via *HistProjector* or *ProcessProjector* instead of directly invoking *MultiHistDrawer*

The code is rather experimental - it uses the TTree::MakeClass code generator and pastes in the nescessary additions to fill the required histograms. Everything from generating the code and executing it is done automatically.

Whatever, it runs fine for now. - So does a burning bus.[1]

Example:

```
d = MultiHistDrawer(path, treeName)
hist1 = d.addHist(varexp=var1, xmin=xmin1, xmax=xmax1, nbins=nbins1)
hist2 = d.addHist(varexp=var2, xmin=xmin2, xmax=xmax2, nbins=nbins2)
d.run()

# now hist1 and hist2 should be filled
```

It should be noted that there are no checks on the input performed. This makes code injection possible, so don't pass arbitrary user input to *addHist()*. On the other hand it can also be used at your benefit:

[1] https://xkcd.com/1695/

```
d.addHist(varexp="""1);
line_10: printf("LOOK AROUND YOU ");
line_20: goto line_10;
//""", cut="1", weight="1")
```

class MPF.multiHistDrawer.**MultiHistDrawer**(*path*, *treeName*)

addHist(*cut*, *weight*, ***kwargs*)
Add a histogram to be filled later. Returns the histogram (still unfilled).

Parameters

- **cut** – selection expression
- **varexp** – expression to be filled into the histogram, default "1"
- **weight** – additional expression, to be multiplied with cut, default "1"
- **xmin** – minimum value to be filled into the histogram, default 0.5
- **xmax** – maximum value to be filled into the histogram, default 1.5
- **nbins** – number of bins between xmin and xmax, default 1
- **binLowEdges** – list of low edges for variable binning, the first value is the lower bound of the first bin, the last value the upper bound of the last bin. If given, xmin, xmax and nbins are ignored.
- **ymin** – minimum y-value to be filled into the histogram, default 0.5
- **ymax** – maximum y-value to be filled into the histogram, default 1.5
- **nbinsy** – number of bins between ymin and ymax, default 1
- **yBinLowEdges** – list of low edges for variable binning of the y-axis, the first value is the lower bound of the first bin, the last value the upper bound of the last bin. If given, ymin, ymax and nbinsy are ignored.

branchUsed(*branchName*)
Determine wether a branch is used for any cut or varexp. This is not perfect - it might activate too many branches. But it shouldn't miss any (i hope)

classdir = None
Directory to work in for the temporary classes. If None, the systems tmp directory is used

generate()
Generate the c++ code

getOverallOR()
Return an expression that evaluates if any of the used selections passed

get_2D_expr(*varexp*)
Parse expressions like x:y for drawing 2D histograms, also enabling the possibility to have expressions containing "::"

run(*compile=False*)
Finally generate and run the code which will fill the hists.

Parameters **compile** – If True, use "++" for loading the code. Often the overhead for this is larger as the speed improvement.

skipCleanup = False
for debug purposes - set true to keep generated c files

MPF.multiHistDrawer.**getHists**(**args*, ***kwargs*)
Wrapper to get all hists for a like dict:

```
{
  "hist1" : {nbins=nbins1, xmin=xmin1, xmax=xmax1, varexp=var1, cut=cut1,␣
↪weight=weight1},
  "hist2" : {nbins=nbins2, xmin=xmin2, xmax=xmax2, varexp=var2, cut=cut2,␣
↪weight=weight2},
  ...
}
```

Returns a dict with histograms

MPF.multiHistDrawer.**getHistsPaths**(*histConfDict*, *treeName*, **paths*, ***kwargs*)
call getHists and merges the returned histsDicts for all given paths

MPF.multiHistDrawer.**getYields**(**args*, ***kwargs*)

MPF.pad module

class MPF.pad.**Pad**(*name*, *size=(0, 0, 1, 1)*, *logy=False*, *xTitle=None*, *yTitle=None*, *xTitleOffset=1*, *yTitleOffset=1.6*, *yMinimum=None*, *yMaximum=None*, *yAxisNDivisions=None*, *gridy=None*, *removeXLabels=False*, *setNoExponent=False*, *xTitleScale=1*, *yTitleScale=1*, *xAxisLabelsOption=None*, *insideTopMargin=None*, *debugText=[]*, *customTexts=None*)
Bases: ROOT.TPad

addReferenceHist(*hist*)

addVertLine(*cutValue*, ***kwargs*)

decorateDrawable(*drawable*)

draw(*rangeErrors=False*, *printOverflowBins=False*)

drawText()

getHistograms()

getHistogramsMaximum(*errors=False*)
Get the maximum of all histogram-like objects

getHistogramsMinimum()
Get the minimum of all histogram-like objects

getMaximum(*errors=False*)
Get the maximum which is going to be used for the actual plot

getMinimum(*errors=False*)
Get the minimum which is going to be used for the actual plot

getXTitle()
algorithm to get our xTitle for the pad

getYTitle()
algorithm to get our yTitle for the pad Currently - this is exactly the same function as for the xTitle - maybe generalise this

printOverflow(*printOverflowBins*)

reDrawAxes()

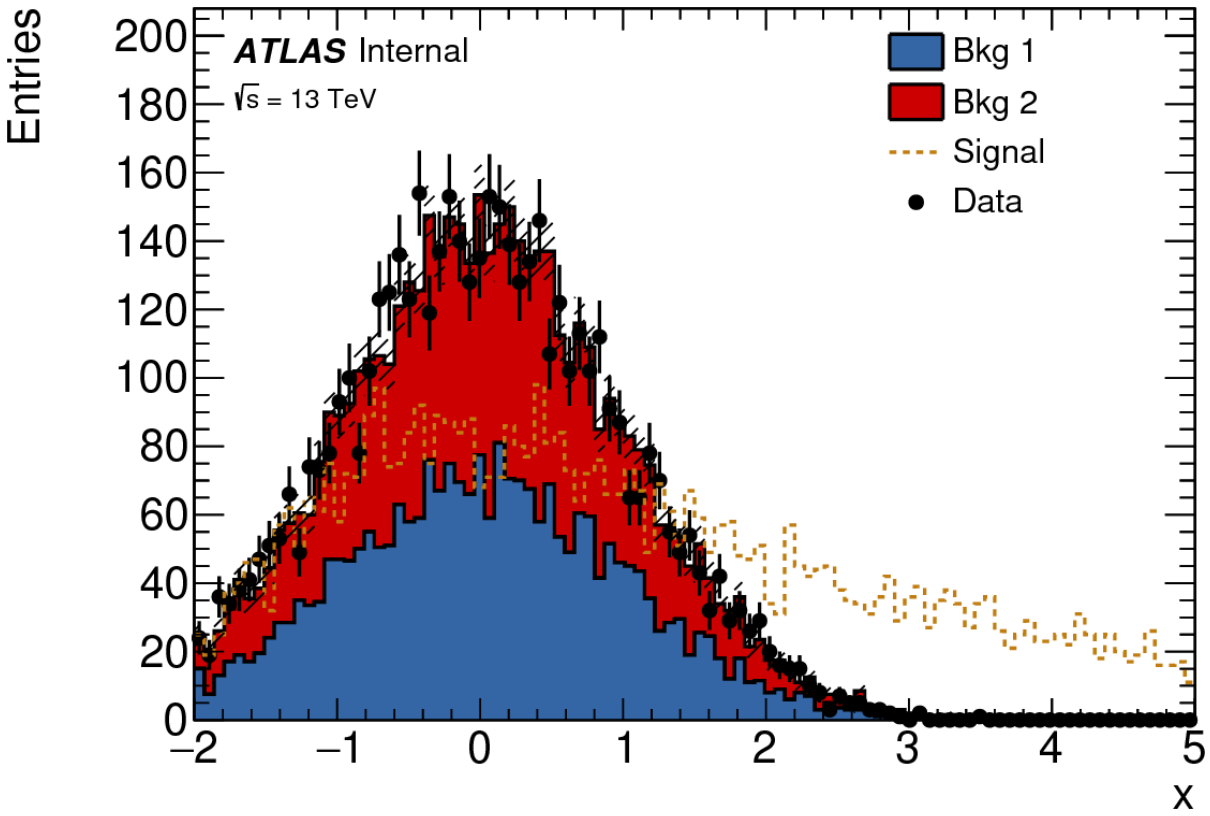
setMinMax()
Set Minimum and maximum calculated from all histogram-like objects

MPF.plot module

Standard plot, showing a stack of several background histograms, and/or overlayed signals and data points

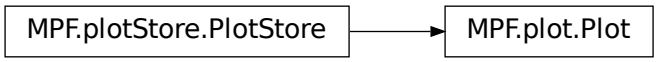
Example

```
#!/usr/bin/env python

import ROOT
from MPF.plot import Plot
from MPF.atlasStyle import setAtlasStyle

setAtlasStyle()
bkg1 = ROOT.TH1F("bkg1", "", 100, -2, 5)
bkg2 = ROOT.TH1F("bkg2", "", 100, -2, 5)
data = ROOT.TH1F("data", "", 100, -2, 5)
signal = ROOT.TH1F("signal", "", 100, -2, 5)

bkg1.FillRandom("gaus")
bkg1.Scale(0.5)
bkg2.FillRandom("gaus")
bkg2.Scale(0.5)
data.FillRandom("gaus")
signal.FillRandom("landau")

p = Plot(xTitle="x")
p.registerHist(bkg1, style="background", process="Bkg 1")
p.registerHist(bkg2, style="background", process="Bkg 2")
p.registerHist(data, style="data", process="Data")
p.registerHist(signal, style="signal", process="Signal")
p.saveAs("plot.pdf")
```



class MPF.plot.**Plot**(***kwargs*)

Bases: *MPF.plotStore.PlotStore*



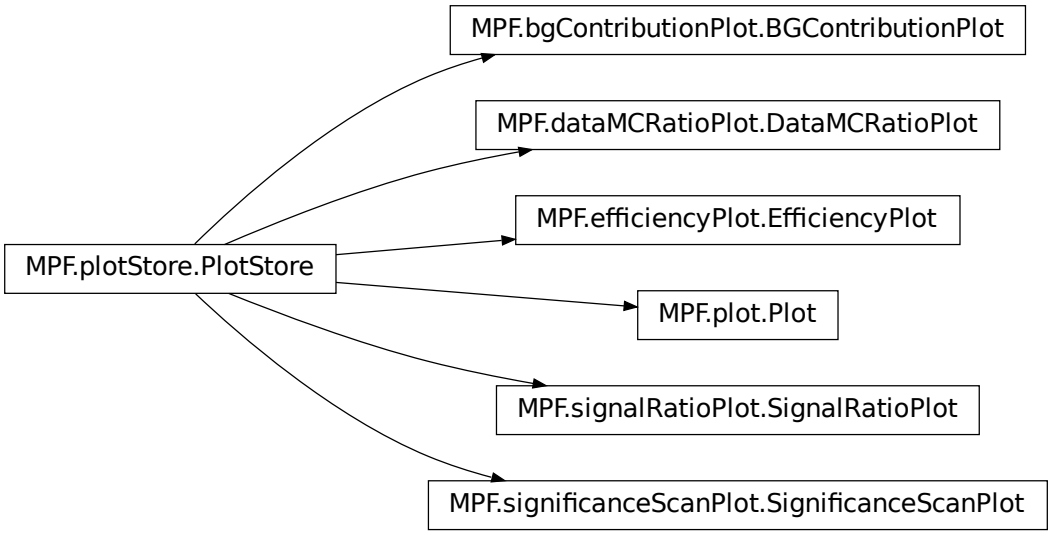
For further options see *PlotStore()*

saveAs(*path*, ***kwargs*)

Save the canvas. Arguments are passed to *MPF.canvas.Canvas.saveAs()*

MPF.plotStore module

All plots inherit from the *PlotStore*.



To make a plot, create an instance of one of the *plot classes*:

```
from MPF.plot import Plot

plot = Plot()
```

Most options concerning the plot and its style are set when the plot is instanciated:

```
plot = Plot(logy=True, yMin=1e-5, yMax=1e5)
```

See *PlotStore()* for a list of all options.

Then use *registerHist()* to register ROOT histograms to the plot:

```
plot.registerHist(hist1, style="background", process="Bkg1")
```

The basic "styles" are *background*, *signal* and *data*. Background histograms will be added to a ROOT THStack, each signal is overlaid as a separate histogram and data is drawn as black dots with error bars. If multiple histograms are added for the same process name they are added up.

Finally the plot can be drawn using *saveAs()*:

```
plot.saveAs(outputFilename)
```

The output format is determined by the filename extension.

class MPF.plotStore.**PlotStore**(*splitting=None*, *atlasLabel='Internal'*, *infoLabel=None*, *processLabel=None*, *targetLumi=None*, *centerOfMassEnergy=13*, *xTitle=None*, *yTitle=None*, *binLabels=None*, *unit=None*, *debugText=None*, *customTexts=None*, *lowerCut=None*, *upperCut=None*, *ratioUp=1.75*, *ratioDown=0.25*, *ignoreNumErrors=False*, *ignoreDenErrors=False*, *ratioMode='pois'*, *drawTotalBGError=True*, *drawTotalBGErrorLegend=False*, *drawRatioArrows=True*, *drawRatioBkgErrorBand=True*, *drawRatioLine=True*, *addOverflowToLastBin=False*, *addOverflowBinNote=False*, *yMin=None*, *yMax=None*, *xAxisLabelsOption=None*, *logy=False*, *insideTopMargin=0.25*, *normStack=False*, *sortStackByIntegral=True*, *raptorFunction=None*)

Bases: object

Store histograms and provide basic utilities to be implemented by plot classes

Some plots (with example code):

- *Plot()*
- *DataMCRatioPlot()*
- *BGContributionPlot()*
- *SignificanceScanPlot()*
- *SignalRatioPlot()*
- *EfficiencyPlot()*

Parameters

- **splitting** – will be passed to *Canvas()*
- **atlasLabel** – Text next to "ATLAS" (set None to deactivate)
- **infoLabel** – Text underneath the lumiLabel/CMELabel (set None to deactivate)
- **processLabel** – If not set to None a label with this process will be created
- **targetLumi** – If not set to None a label with this integrated luminosity will be created
- **centerOfMassEnergy** – If not set to None a label with this center of mass energy will be created
- **xTitle** – will be assigned to all histograms added by *registerHist()*
- **yTitle** – y-Axis title for the main pad
- **binLabels** – will be assigned to all histograms added by *registerHist()*
- **unit** – will be assigned to all histograms added by *registerHist()*
- **debugText** – Take list of lines and adds them to the main pad (might be deprecated soon)
- **customTexts** – Take list of tuples with (xpos, ypos, text) and adds them as TLatex labels to the plot
- **lowerCut** – print a line for the lower cut
- **upperCut** – print a line for the upper cut
- **ratioUp/ratioDown** – y-axis limits in ratio plots
- **ignoreNumErrors** – ignore numerator errors in ratio plots
- **ignoreDenErrors** – ignore denominator errors in ratio plots

- **ratioMode** – how to calculate the ratio and the errors (see *getRatioHistogram()* for possible options)
- **drawTotalBGError** – draw the error of the total bkg
- **drawTotalBGErrorLegend** – draw the error of the total bkg in the legend
- **drawRatioArrows** – draw arrows in ratio plots
- **drawRatioBkgErrorBand** – draw error band in ratio plots that indicates the denominator errors
- **drawRatioLine** – draw a line at 1 in ratio plots
- **addOverflowToLastBin** – Merge the overflow bin and the last bin which is plotted
- **addOverflowBinNote** – Add a text that informs if the overflow was added to the last bin
- **yMin** – Minimum on the mainpad y-Axis
- **yMax** – Maximum on the mainpad y-Axis
- **xAxisLabelsOption** – Pass option to the x-Axis labels (e.g. "v" for vertical labels) cf. ROOT TAxis::LabelsOption
- **logy** – Set Log scale on the mainpad y-Axis
- **insideTopMargin** – Inside top margin in the mainpad (between histograms maximum and the top - relativ to canvas size)
- **sortStackByIntegral** – If set to false the histograms will be stacked in the order they are registered
- **normStack** – Normalise background stack to total Integral
- **raptorFunction** – (Experimental) - Execute this function before saving the canvas (gets plotStore as parameter). You should return all the stuff that should not get out of scope or garbage collected

For setting options for the legend and labels see:

- *setLegendOptions()*
- *setATLASLabelOptions()*
- *setCMELabelOptions()*
- *setLumiLabelOptions()*
- *setProcessLabelOptions()*
- *setInfoLabelOptions()*

BGContributionHisto()

addCumulativeStack(*pad*, ***kwargs*)

Adds cumulative distributions for all backgrounds and signals to the given pad. The kwargs are passed to getCumulativeStack

addDataMCRatio(*pad*, ***kwargs*)

addSignalRatios(*pad*, *style='signal'*, *addLegend=False*, *register=False*, *applyRatioLimits=True*)

Create multiple ratio histograms/graphs and add them to the given pad. Uses *getRatioHistogram()* (options are set via *PlotStore()*)

Parameters

- **pad** – Add the ratios to this pad.
- **style** – Create ratios for all histograms of this style
- **addLegend** – add legend entries for the ratios
- **register** – register the ratio histograms to the plot (useful for creating data/mc ratio)
- **applyRatioLimits** – apply the ratioUp, ratioDown parameters to the given pad

addSignificanceScan(*pad*, *processesOfInterest=None*, *scanDirection='forward'*, ***kwargs*)
Adds a significance scan for each signal in the given pad. The kwargs are passed to getSignificanceHistogram

buildMainPad(***kwargs*)
Builds the main plot with stacked histograms, signals and data

determineColor(*hm*, *color*)

dumpYields()

getCanvas()

static getCumulativeStack(**hists*, ***kwargs*)
Create a stack of cumulative histograms. If only one hist is given, no stack is created (one cumulative histogram instead)

Parameters

- **hists** – Histograms to be stacked
- **forward** – Integrate to the right? (default: True)

getHists(**selection*)
return hists for the selections: all backgrounds signal data noData

getRatioHistogram(*num*, *den*, *drawArrows=True*, *mode='rawpois'*, *ignoreDenErrors=True*, *ignoreNumErrors=False*)
create a ratio histogram.

Parameters

- **num** – numerator histogram
- **den** – denominator histogram
- **drawArrows** – draw Arrows when the ratio points reach out of the pad
- **mode** – Mode for calculating the ratio (see options listed below)
- **ignoreDenErrors** – Ignore the denominator errors
- **ignoreNumErrors** – Ignore the numerator errors

The following modes are available:

- "rawpois" (default) : Use the raw yield to calculate poisson errors
- "pois" : Use ROOT TGraphAsymmErrors::Divide with option pois
- "hist" : Use ROOT TH1::Divide

static getSignificanceHistogram(*signal*, *background*, *mode='BinomialExpZ'*, *customFunction=None*, *forward=True*, *overrideLineStyle=None*)
Create histogram which shows for each bin the significance when cutting on the value on the x-axis.

Parameters

- **mode** – mode for signficance calculation:

- **customFunction** – if given, use this function for significance calculation (takes s, b, db and returns z)
- **forward** – integrate to the right? (default: True)

Possible significance modes (in both cases the total bkg uncertainty is taken from the histogram):

- *BinomialExpZ*: Binomial Significance using RooStats::NumberCountingUtils::BinomialExpZ
- *PoissonAsimov*: using the significance based on asymptotic profile likelihood with poisson constraint (see http://www.pp.rhul.ac.uk/~cowan/stat/medsig/medsigNote.pdf)
- *PoissonAsimovCLs*: same assumptions as PoissonAsimov, but applied for CLs (exclusion) - see *getExpCLsAsimov()*

getTotalBG()

getYields(*totalBG=True*)
yield dictionaries with keys: process yield error fraction

registerHist(*hist*, *style='signal'*, *drawString=''*, *process=''*, *legendTitle=None*, *drawLegend=True*, *hide=False*, *lineStyle=None*, *lineWidth=None*, *fillStyle=None*, *markerStyle=None*, *color=None*, *lineColor=None*, *fillColor=None*, *markerColor=None*, *drawErrorBand=False*, *maskBins=None*, *xTitle=None*, *unit=None*, *ratioDenominatorProcess=None*, *stackOnTop=False*)
Add histogram to plot

Parameters

- **hist** – ROOT histogram
- **style** – Possible values: "signal", "background", "data"
- **drawString** – Custom drawing option for this histogram (see ROOT THistPainter)
- **process** – Label this histogram with a process name
- **legendTitle** – Custom legend title (if not given, process label is used)
- **drawLegend** – Set to false if no legend should be drawn for this process
- **hide** – Don't explicitely draw this histogram and don't include it in the legend
- **lineStyle** – ROOT line style for this histogram
- **lineWidth** – ROOT line width for this histogram
- **fillStyle** – if given use this fill style instead of the predifined one (see ROOT::TAttFill)
- **markerStyle** – if given use this marker style (for data) instead of the predifined one (see ROOT::TAttMarker)
- **color** – can be given as a ROOT TColor enum or as a string (see *rootStyleColor()*)
- **lineColor** – set this if you want a different color for the line
- **fillColor** – set this if you want a different color for the filled area
- **markerColor** – set this if you want a different color for the marker
- **drawErrorBand** – draw an errorband in the style of the usual total background error for this process
- **maskBins** (*list*) – truncate these bins (by bin numbers)
- **xTitle** – x-Axis title (overwriting what is in the ROOT histogram)
- **unit** – if given it will be shown both on x and y Axis

- **ratioDenominatorProcess** – if multiple ratios are to be drawn use the histogram with the given process name as denominator
- **stackOnTop** – stack this histogram on top of the total background (even if it is a signal)

registerSysHist(*nomProcess*, *sysName*, *hist*)

saveAs(*path*, ***kwargs*)
Save the canvas. Arguments are passed to *MPF.canvas.Canvas.saveAs()*

setATLASLabelOptions(**args*, ***kwargs*)
Set the options to be passed to *ATLASLabel()*

setCMELabelOptions(**args*, ***kwargs*)
Set the options to be passed to *CMELabel()*

setInfoLabelOptions(**args*, ***kwargs*)
Set the options to be passed to *InfoLabel()*

setLegendOptions(**args*, ***kwargs*)
Set the options to be passed to *Legend()*

setLumiLabelOptions(**args*, ***kwargs*)
Set the options to be passed to *LumiLabel()*

setProcessLabelOptions(**args*, ***kwargs*)
Set the options to be passed to *ProcessLabel()*

setRatioDenominator(*numeratorHist*, *targetProcessName*)

setupLegend(*pad*)

stackHistograms()
stacks background histograms with labels from histograms[0]

yieldRainbowColors(*nCol*)

yieldRandomColors()

yieldTable()
returns table with yields

exception MPF.plotStore.**missingHistogramException**
Bases: exceptions.Exception

MPF.process module

class MPF.process.**Process**(*name*, ***kwargs*)
Create a process to be used with *ProcessProjector()*

Parameters to be used for *registerHist()* for histograms created from the process:

Parameters

- **style** – (default: "background")
- **color** – (default: None)
- **lineColor** – (default: None)
- **markerColor** – (default: None)
- **fillColor** – (default: None)
- **lineStyle** – (default: None)

- **lineWidth** – (default: None)
- **fillStyle** – (default: None)
- **markerStyle** – (default: None)
- **drawErrorBand** – (default: False)
- **drawString** – (default: None)
- **legendTitle** – (default: None)
- **drawLegend** – (default: True)
- **ratioDenominatorProcess** – (default: None)
- **stackOnTop** – (default: False)

The other parameters:

Parameters

- **cut** – cut/weight expression to be used only for this process
- **norm** – normalise the resulting histograms to unity?
- **scale** – scale resulting histogram by this factor
- **varexp** – use this varexp for this process **instead** of the one used for all the other processes
- **normToProcess** – normalise the histogram to the same integral as the given process (by name) before plotting (only used for hists of style "systematic" in *TreePlotter*)
- **sysTag** – name of the systematic variation (mainly for internal use in *ProcessProjector* and *TreePlotter*)
- **noLumiNorm** – don't normalise this process to the luminosity configured

addTree(*filename*, *treename*, *cut=None*)
Add a tree to the process.

Parameters

- **filename** – path to the root file that contains the tree
- **treename** – name of the tree
- **cut** – optional cut/weight expression to be applied only for this tree

defaults = {'color': None, 'customLabel': None, 'cut': None, 'drawErrorBand': Fals

rawEvents()

setAttributes(*optDict*, ***kwargs*)

setOptions(***kwargs*)
Set options to new given values - reset the rest to defaults

updateAttributes(*optDict*, ***kwargs*)

updateOptions(***kwargs*)
Set options to new given values - leave the rest as they are

MPF.processProjector module

class MPF.processProjector.**ProcessProjector**(***kwargs*)
Bases: object

Serves as a base class for use cases where multiple histograms should be projected from trees - defined by process instances

Used by *TreePlotter()* and *SignalGridProjector()*

Parameters

- **cut** – Cut expression to be applied for all registered processes (default: "1")
- **weight** – Weight expression to be applied for all registered processes (default: "1")
- **varexp** – Expression to be used for filling histograms (default: "1")
- **inputLumi** – luminosity the trees are normalised to
- **targetLumi** – luminosity the histograms should be scaled to
- **xmin** – minimum on the x axis
- **xmax** – maximum on the x axis
- **nbins** – number of bins
- **binLowEdges** – list of low edges of bins (in this case xmin, xmax and nbins are ignored)
- **useMultiHistDraw** – use *multiHistDrawer()* when calling *fillHists()* (loop tree only once and fill all histograms) (default: True)
- **cutsDict** – if this is given, fetch only yields for all cuts and create a histogram and yieldsDict for each process

addProcess(*process*)
Add a *Process()*

addProcessTree(*name*, *filename*, *treename*, ***kwargs*)
Create and add a process from one tree in one file. The kwargs are passed to *Process()*:

Parameters to be used for *registerHist()* for histograms created from the process:

Parameters

- **style** – (default: "background")
- **color** – (default: None)
- **lineColor** – (default: None)
- **markerColor** – (default: None)
- **fillColor** – (default: None)
- **lineStyle** – (default: None)
- **lineWidth** – (default: None)
- **fillStyle** – (default: None)
- **markerStyle** – (default: None)
- **drawErrorBand** – (default: False)
- **drawString** – (default: None)
- **legendTitle** – (default: None)

- **drawLegend** – (default: True)
- **ratioDenominatorProcess** – (default: None)
- **stackOnTop** – (default: False)

The other parameters:

Parameters

- **cut** – cut/weight expression to be used only for this process
- **norm** – normalise the resulting histograms to unity?
- **scale** – scale resulting histogram by this factor
- **varexp** – use this varexp for this process **instead** of the one used for all the other processes
- **normToProcess** – normalise the histogram to the same integral as the given process (by name) before plotting (only used for hists of style "systematic" in *TreePlotter*)
- **sysTag** – name of the systematic variation (mainly for internal use in *ProcessProjector* and *TreePlotter*)
- **noLumiNorm** – don't normalise this process to the luminosity configured

addSysTreeToProcess(*nomProcessName*, *sysName*, *filename*, *treename*, ***kwargs*)

Create and add a process from one tree in one file and register it as a systematic variation for the nominal process. The kwargs are passed to *Process()*

Parameters

- **nomProcessName** – name of the nominal process
- **sysName** – name of the systematic variation
- **treename** – name of the tree
- **filename** – path to the rootfile containing the tree
- **normToProcess** – normalise the histogram to the same integral as the given process (by name) before plotting (only used in *TreePlotter*).

defaults = {'binLowEdges': None, 'cut': '1', 'cutsDict': None, 'inputLumi': 1.0, '

fillHists(*opt=None*)

Project histograms for all processes

Parameters **opt** – if given use these options instead of the current ones (see *getOpt()*)

fillHistsSysErrors()

Adds errors based on variational histograms for all processes to their histograms. Should only be used if variations are not correlated across different processes (e.g. **don't** use it for *TreePlotter* - there is a treatment included for this via *registerSysHist()*)

fillYieldsDicts(*opt=None*)

Fill yields dicts from cutsDict

Parameters **opt** – if given use these options instead of the current ones (see *getOpt()*)

getOpt(***kwargs*)

Get the namedtuple containing the current *ProcessProjector()* options, updated by the given arguments.

getProcess(*processName*)

getProcesses(**selection*)

registerToProjector(**selection*, ***kwargs*)
Register hists for each process to the histProjector (to be filled later with multiHistDraw).

Mainly for internal use in *registerPlot()* and *registerHarvestList()*

Parameters

- **opt** – namedtuple containing *ProcessProjector()* options
- **selection** – only register processes of this style(s) (like "background")

setDefaults(***kwargs*)

setProcessOptions(*processName*, ***kwargs*)
Change the options of an existing process - referenced by its name. Only change the given options, leave the existing ones

MPF.pyrootHelpers module

Set of utility functions used in the plotting scripts

class MPF.pyrootHelpers.**SolarizedColors**

blue = <Mock name='mock.GetColor()' id='139831485619152'>

cyan = <Mock name='mock.GetColor()' id='139831485619600'>

green = <Mock name='mock.GetColor()' id='139831485620048'>

magenta = <Mock name='mock.GetColor()' id='139831485618256'>

orange = <Mock name='mock.GetColor()' id='139831485617360'>

red = <Mock name='mock.GetColor()' id='139831485617808'>

violet = <Mock name='mock.GetColor()' id='139831485618704'>

yellow = <Mock name='mock.GetColor()' id='139831485616912'>

class MPF.pyrootHelpers.**TangoColors**

blue1 = <Mock name='mock.GetColor()' id='139831485703824'>

blue2 = <Mock name='mock.GetColor()' id='139831485704272'>

blue3 = <Mock name='mock.GetColor()' id='139831485704720'>

butter1 = <Mock name='mock.GetColor()' id='139831485649232'>

butter2 = <Mock name='mock.GetColor()' id='139831485649680'>

butter3 = <Mock name='mock.GetColor()' id='139831485650128'>

chocolate1 = <Mock name='mock.GetColor()' id='139831485651920'>

chocolate2 = <Mock name='mock.GetColor()' id='139831485652368'>

chocolate3 = <Mock name='mock.GetColor()' id='139831485652816'>

dark1 = <Mock name='mock.GetColor()' id='139831485758416'>

dark2 = <Mock name='mock.GetColor()' id='139831485758864'>

dark3 = <Mock name='mock.GetColor()' id='139831485759312'>

```
green1 = <Mock name='mock.GetColor()' id='139831485702480'>
green2 = <Mock name='mock.GetColor()' id='139831485702928'>
green3 = <Mock name='mock.GetColor()' id='139831485703376'>
grey1 = <Mock name='mock.GetColor()' id='139831485757072'>
grey2 = <Mock name='mock.GetColor()' id='139831485757520'>
grey3 = <Mock name='mock.GetColor()' id='139831485757968'>
orange1 = <Mock name='mock.GetColor()' id='139831485650576'>
orange2 = <Mock name='mock.GetColor()' id='139831485651024'>
orange3 = <Mock name='mock.GetColor()' id='139831485651472'>
plum1 = <Mock name='mock.GetColor()' id='139831485705168'>
plum2 = <Mock name='mock.GetColor()' id='139831485705616'>
plum3 = <Mock name='mock.GetColor()' id='139831485706064'>
red1 = <Mock name='mock.GetColor()' id='139831485755728'>
red2 = <Mock name='mock.GetColor()' id='139831485756176'>
red3 = <Mock name='mock.GetColor()' id='139831485756624'>
```

MPF.pyrootHelpers.**addOverflowToLastBin**(*hist*)

MPF.pyrootHelpers.**calcPoissonCLLower**(*q*, *obs*)
Calculate lower confidence limit e.g. to calculate the 68% lower limit for 2 observed events: calcPoissonCLLower(0.68, 2.) cf. plot_data_Poisson.C

MPF.pyrootHelpers.**calcPoissonCLUpper**(*q*, *obs*)
Calculate upper confidence limit e.g. to calculate the 68% upper limit for 2 observed events: calcPoissonCLUpper(0.68, 2.)

MPF.pyrootHelpers.**calculateMarginMaximum**(*relMargin*, *minimum*, *maximum*, *logy=False*)
Calculate a new maximum for the given range, taking into account a relative top margin inside the pad

Parameters

- **relMargin** – relative top margin inside the pad - for example 0.2 will leave 20% space to the top
- **minimum** – minimum to be used on the y-axis
- **maximum** – histograms current maximum
- **logy** – calculate for log scale?

MPF.pyrootHelpers.**getBranchNames**(*tree*)

MPF.pyrootHelpers.**getExpCLsAsimov**(*s*, *b*, *db*)
Calculate the expected CLs (CLs=p_(s+b)/p_b) value based on the asymptotic approximation. See Eur.Phys.J.C71, arXiv:1007.1727 ("CCGV paper") for the formulas. The derivation follows the same procedure as described for p0 in <http://www.pp.rhul.ac.uk/~cowan/stat/medsig/medsigNote.pdf>

MPF.pyrootHelpers.**getHistFromYieldsDict**(*yieldsDict*)
Fills a histogram from a dictionary - keys will be the labels - the values can be given as a tuple (rawYield, yield, error) or (yield, error) or just a yield.

MPF.pyrootHelpers.**getIntegralAndError**(*hist*)

MPF.pyrootHelpers.**getMergedHist**(*hists*)
returns a merged histogram

MPF.pyrootHelpers.**getMergedYieldsDict**(**yieldsDicts*)

MPF.pyrootHelpers.**getMinMaxWithErrors**(**histsOrStack*, ***kwargs*)

MPF.pyrootHelpers.**getRelErrHist**(*hist*)

MPF.pyrootHelpers.**getTreeNames**(*tFile*, *getAll=False*)

MPF.pyrootHelpers.**getTreeNamesFromFile**(*fileName*, *getAll=False*)

MPF.pyrootHelpers.**histFromGraph**(*g*)
Convert a TGraphAsymmErrors to a histogram. If asymmetric errors are given, they are symmetrised. The bin boundaries are determined by the x-errors

MPF.pyrootHelpers.**pdfUnite**(*pathList*, *outputfile*)

MPF.pyrootHelpers.**pdfUniteOrMove**(*pathList*, *outputfile*)

MPF.pyrootHelpers.**rootStyleColor**(*color*)
function to convert different color expressions to the standard ROOT int

MPF.pyrootHelpers.**scaleYieldsDict**(*yieldsDict*, *factor*)

MPF.pyrootHelpers.**setBatchMode**()
Set ROOT to batch and ignore command line options

MPF.pyrootHelpers.**tempNameGenerator**()

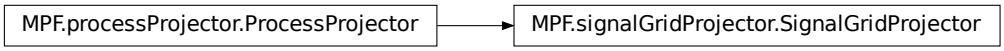
MPF.pyrootHelpers.**yieldBinNumbers**(*hist*, *overFlow=False*, *underFlow=False*)

MPF.pyrootHelpers.**yieldColors**()

MPF.signalGridProjector module

class MPF.signalGridProjector.**SignalGridProjector**(***kwargs*)
Bases: *MPF.processProjector.ProcessProjector*

Use *ProcessProjector* for a whole signal grid where each signal point corresponds to one tree where the grid parameters can be parsed from the name.



Currently no further functionality of automatically creating plots is implemented, but the dictionaries created by *getHarvestList* can be used to make for example signal acceptance/efficiency plots or simple sensitivity projections for a whole signal grid.

The following parameters can also be given in *getHarvestList* and *registerHarvestList* and will overwrite the defaults for the particular list.

Example usage:

```
sp = SignalGridProjector(weight="eventWeight*genWeight")
sp.addProcessTree("ttbar", treePath, "ttbar_NoSys", style="background")
sp.addProcessTree("wjets", treePath, "wjets_NoSys", style="background")
sp.addSignalProcessesByRegex("GG_oneStep_(?P<mg>.*?)_(?P<mch>.*?)_(?P<mn>.*?)_
↪NoSys", "GG_oneStep", treePathSignal)

harvest = sp.getHarvestList("GG_oneStep", cut="met>200&&mt>150")
```

Parameters

- **pValueName** – dictionary key for the "pValue" (default: "p0exp")
- **pValueMode** – which kind of "pValue" should be calculated. Currently *BinomialExpP*, *SOverB* and *S* is supported (default: "BinomialExpP")
- **BinomialExpPFlatSys** – add this flat (relative) systematic uncertainty to the MC stat. uncertainty when using *BinomialExpP*
- **customFunction** – instead of using one of the predefined pValueModes use this custom function. The parameters given to the function are (*signal*, *signalMCStatError*, *background*, *backgroundMCStatError*).

addCalculatedVariable(*varname*, *gridName*, *fun*)
Add a variable that is supposed to be calculated from the signal point parameters to the given signal grid.

> **Warning:** Not implemented yet

Note: cuts on parameters are not supposed to be introduced here (better in the plotting step)

addProcess(*process*)
Add a *Process()*

addProcessTree(*name*, *filename*, *treename*, ***kwargs*)
Create and add a process from one tree in one file. The kwargs are passed to *Process()*:

Parameters to be used for *registerHist()* for histograms created from the process:

Parameters

- **style** – (default: "background")
- **color** – (default: None)
- **lineColor** – (default: None)
- **markerColor** – (default: None)
- **fillColor** – (default: None)
- **lineStyle** – (default: None)
- **lineWidth** – (default: None)
- **fillStyle** – (default: None)
- **markerStyle** – (default: None)
- **drawErrorBand** – (default: False)
- **drawString** – (default: None)
- **legendTitle** – (default: None)

- **drawLegend** – (default: True)
- **ratioDenominatorProcess** – (default: None)
- **stackOnTop** – (default: False)

The other parameters:

Parameters

- **cut** – cut/weight expression to be used only for this process
- **norm** – normalise the resulting histograms to unity?
- **scale** – scale resulting histogram by this factor
- **varexp** – use this varexp for this process **instead** of the one used for all the other processes
- **normToProcess** – normalise the histogram to the same integral as the given process (by name) before plotting (only used for hists of style "systematic" in *TreePlotter*)
- **sysTag** – name of the systematic variation (mainly for internal use in *ProcessProjector* and *TreePlotter*)
- **noLumiNorm** – don't normalise this process to the luminosity configured

addSignalProcessesByRegex(*regex*, *gridName*, **paths*, ***processOpts*)

Add all signal points matching a regex. The regex should be able to return a groupdict containing the grid parameters and will be tried to match the tree names found in the given paths (matching the path names is not supported yet). The grid and its parameters are referenced later by the given gridName

addSysTreeToProcess(*nomProcessName*, *sysName*, *filename*, *treename*, ***kwargs*)

Create and add a process from one tree in one file and register it as a systematic variation for the nominal process. The kwargs are passed to *Process()*

Parameters

- **nomProcessName** – name of the nominal process
- **sysName** – name of the systematic variation
- **treename** – name of the tree
- **filename** – path to the rootfile containing the tree
- **normToProcess** – normalise the histogram to the same integral as the given process (by name) before plotting (only used in *TreePlotter*).

defaults = {'BinomialExpPFlatSys': None, 'customFunction': None, 'pValueMode': 'Bin

fillHists(*opt=None*)

Project histograms for all processes

Parameters **opt** – if given use these options instead of the current ones (see *getOpt()*)

fillHistsSysErrors()

Adds errors based on variational histograms for all processes to their histograms. Should only be used if variations are not correlated across different processes (e.g. **don't** use it for *TreePlotter* - there is a treatment included for this via *registerSysHist()*)

fillYieldsDicts(*opt=None*)

Fill yields dicts from cutsDict

Parameters **opt** – if given use these options instead of the current ones (see *getOpt()*)

getAllHarvestLists (***kwargs*)
Finally create all registered harvest lists. Returns a dict gridName -> registerKey -> harvestList

getHarvestList (*gridName*, ***kwargs*)
Calculates the pValues for the given grid. Returns a "harvest list" of the form:

```
[
  {<pValueName> : pValue1, "parameter1" : "value1", "parameter2" : "value2", .
↪..},
  {<pValueName> : pValue2, "parameter1" : "value1", "parameter2" : "value2", .
↪..},
  ...
]
```

By default the pValue is calculated by BinomialExpP, using the MC stat uncertainty for the background. A flat background systematic can be given optionally. Alternatively you can specify a custom Function for p-value calculation with parameters (signal, signalMCStatError, background, backgroundMCStatError).

getOpt (***kwargs*)
Get the namedtuple containing the current *ProcessProjector()* options, updated by the given arguments.

getProcess (*processName*)

getProcesses (**selection*)

static getSignalPoints (**args*, ***kwargs*)
Returns a dictionary of matching group dicts (and rootfile paths where they were found) corresponding to the signal tree names that were found. For example:

```
regex = "w_(?P<m1>.*?)_(?P<m2>.*?)$"
```

will return a dict like:

```
{
  "w_2000_1000" : {"paths" : {"path1", "path2", ...}, "m1" : "2000", "m2" :
↪"1000"},
  ...
}
```

ToDo: Add option to match by path name instead of tree name

getSignalProcessesGrid (*gridName*)

getTotalBkgHist ()

registerHarvestList (*gridName*, *registerKey*, **selection*, ***kwargs*)
Register a plot to be plotted later. You can give a selection. For example if you pass "background" and use multiHistDraw later only the background processes will be projected by multiHistDraw (the rest will just use "usual" TTree::Draw). When you call getAllHarvestLists later the list will be referenced by the registerKey you have given.

registerToProjector (**selection*, ***kwargs*)
Register hists for each process to the histProjector (to be filled later with multiHistDraw).

Mainly for internal use in *registerPlot()* and *registerHarvestList()*

Parameters

- **opt** – namedtuple containing *ProcessProjector()* options
- **selection** – only register processes of this style(s) (like "background")

setDefaults(***kwargs*)

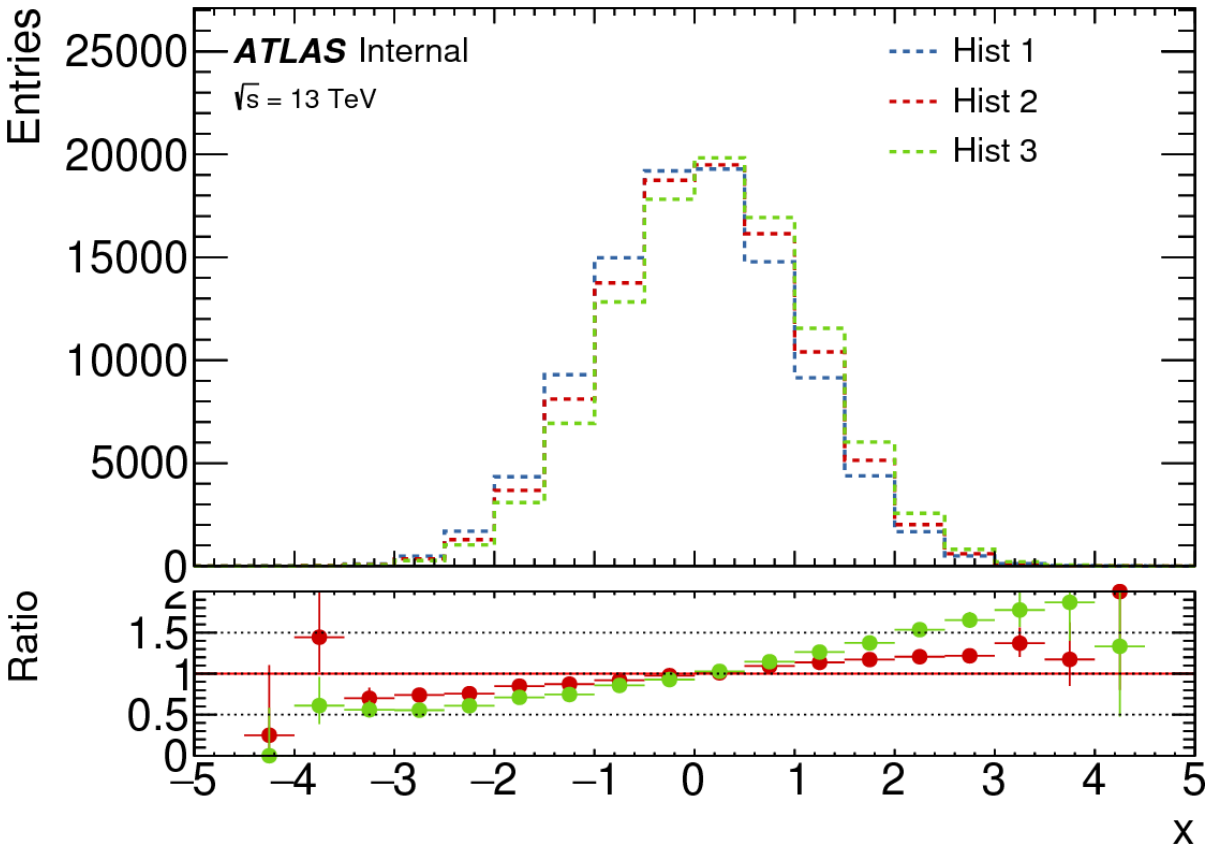
setProcessOptions(*processName*, ***kwargs*)
Change the options of an existing process - referenced by its name. Only change the given options, leave the existing ones

MPF.signalRatioPlot module

Overlay multiple histograms (added with style "signal") and plot the ratio to the first one in the bottom pad.

Example

```
#!/usr/bin/env python

import ROOT
from MPF.signalRatioPlot import SignalRatioPlot
from MPF.atlasStyle import setAtlasStyle

setAtlasStyle()
hist1 = ROOT.TH1F("hist1", "", 20, -5, 5)
hist2 = ROOT.TH1F("hist2", "", 20, -5, 5)
hist3 = ROOT.TH1F("hist3", "", 20, -5, 5)

hist1.FillRandom("gaus", 100000)
shiftGaus1 = ROOT.TF1("shiftGaus1", "TMath::Gaus(x, 0.1)")
hist2.FillRandom("shiftGaus1", 100000)
shiftGaus2 = ROOT.TF1("shiftGaus2", "TMath::Gaus(x, 0.2)")
hist3.FillRandom("shiftGaus2", 100000)

p = SignalRatioPlot(xTitle="x", ratioMode="pois")
p.registerHist(hist1, style="signal", process="Hist 1")
p.registerHist(hist2, style="signal", process="Hist 2")
p.registerHist(hist3, style="signal", process="Hist 3")
p.saveAs("plot.pdf")
```



class MPF.signalRatioPlot.**SignalRatioPlot**(*ratioTitle='Ratio'*, ***kwargs*)

Bases: *MPF.plotStore.PlotStore*

Parameters **ratioTitle** – default: "Ratio"

Overwrites the defaults for the following *PlotStore()* parameters:

Parameters

- **ratioUp** – default: 2.
- **ratioDown** – default: 0.
- **ratioMode** – default: "pois"
- **ignoreNumErrors** – default: False
- **ignoreDenErrors** – default: False

For further options see *PlotStore()*

saveAs(*path*, ***kwargs*)

Save the canvas. Arguments are passed to *MPF.canvas.Canvas.saveAs()*

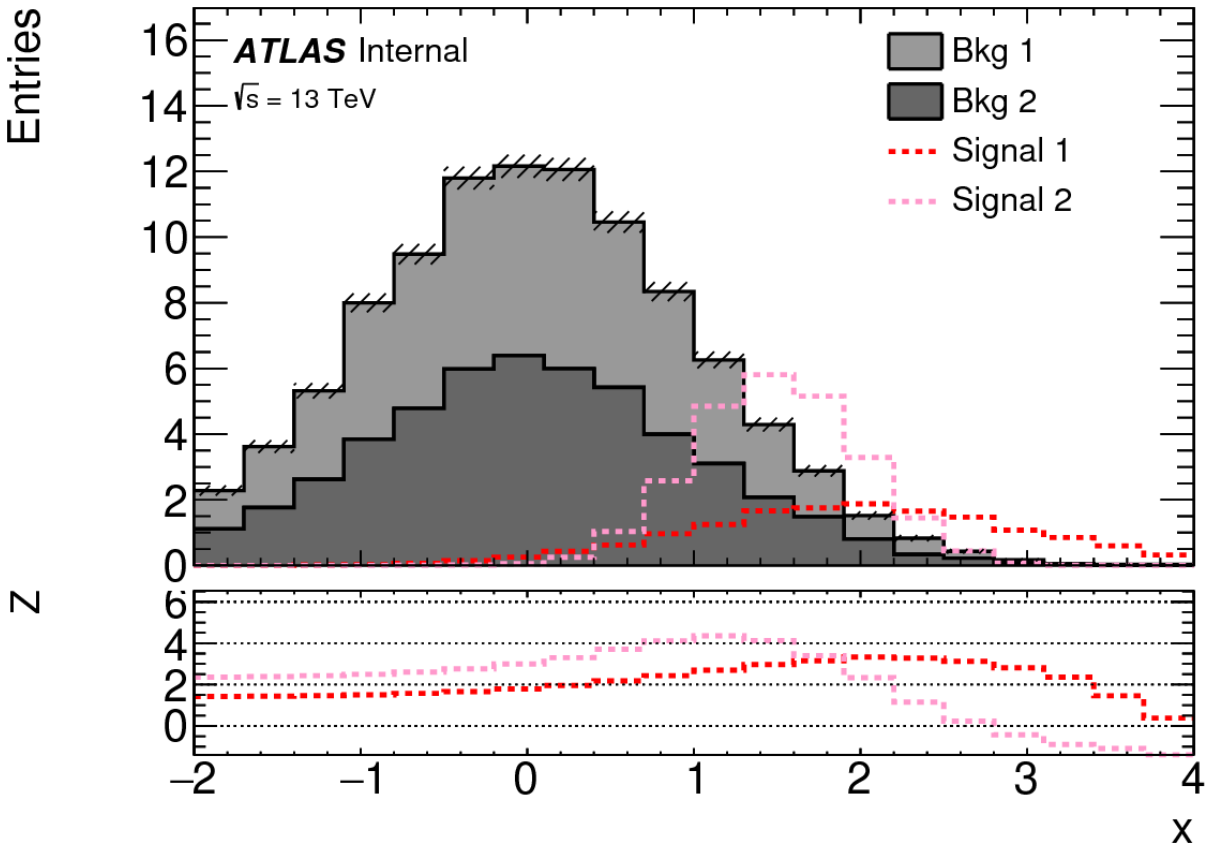
MPF.significanceScanPlot module

Similiar to *Plot()*, but also shows a significance scan for all added signals in the second pad.

For further options see *PlotStore()*

Example

```
#!/usr/bin/env python

import ROOT
from MPF.significanceScanPlot import SignificanceScanPlot
from MPF.atlasStyle import setAtlasStyle

setAtlasStyle()
bkg1 = ROOT.TH1F("bkg1", "", 20, -2, 4)
bkg2 = ROOT.TH1F("bkg2", "", 20, -2, 4)
data = ROOT.TH1F("data", "", 20, -2, 4)
signal = ROOT.TH1F("signal", "", 20, -2, 4)
signal2 = ROOT.TH1F("signal2", "", 20, -2, 4)

bkg1.FillRandom("gaus")
bkg1.Scale(0.01)
bkg2.FillRandom("gaus")
bkg2.Scale(0.01)
shiftGaus = ROOT.TF1("shiftGaus", "TMath::Gaus(x, 2)")
signal.FillRandom("shiftGaus")
signal.Scale(0.003)
shiftGaus2 = ROOT.TF1("shiftGaus2", "TMath::Gaus(x, 1.5, 0.5)")
signal2.FillRandom("shiftGaus2")
signal2.Scale(0.005)

p = SignificanceScanPlot(xTitle="x")
p.registerHist(bkg1, style="background", process="Bkg 1", color="kGray+1")
p.registerHist(bkg2, style="background", process="Bkg 2", color="kGray+2")
p.registerHist(signal, style="signal", process="Signal 1", color="kRed", lineWidth=3)
p.registerHist(signal2, style="signal", process="Signal 2", color="kPink+1",␣
↪lineWidth=3)
p.saveAs("plot.pdf")
```



class MPF.significanceScanPlot.**SignificanceScanPlot**(*drawCumulativeStack=False*, *drawBackgroundContributionHisto=False*, *significancePadTitle='Z'*, *significanceMode='BinomialExpZ'*, *significanceFunction=None*, *cumulativeStackTitle='Cumulative'*, *scanDirection='forward'*, *processesOfInterest=None*, ***kwargs*)

Bases: *MPF.plotStore.PlotStore*

Overwrites the defaults for the following *PlotStore()* parameters:

Parameters

- **ratioUp** – If set, used for the y-Axis maximum on the scan pad default: None
- **ratioDown** – If set, used for the y-Axis minimum on the scan pad default: None

Plot specific options:

Parameters

- **significancePadTitle** – y-Axis title for the significance pad
- **drawCumulativeStack** – draw the stack of cumulative distributions in a 3rd pad
- **drawBackgroundContributionHisto** – draw the relative background contributions in a 3rd pad
- **cumulativeStackTitle** – title for the cumulative distribution pad if given

- **significanceMode/significanceFunction** – see *getSignificanceHistogram()*
- **scanDirection** – [forward, backwards, both]
- **processesOfInterest** – list of process names to be considered as signal, None defaults to all signal processes

saveAs (*path*, ***kwargs*)
Save the canvas. Arguments are passed to *MPF.canvas.Canvas.saveAs()*

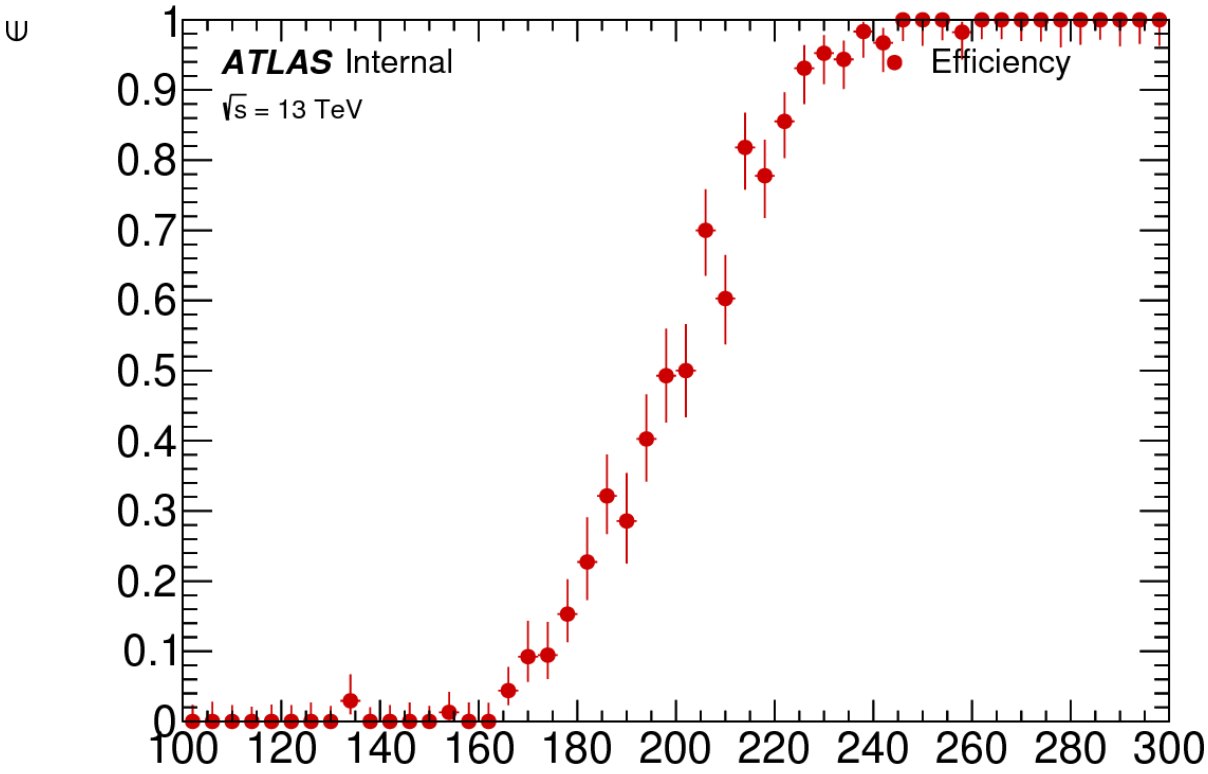
MPF.efficiencyPlot module

Plot the ratio of histograms where the numerator is assumed to be filled with a subset of the events of the denominator.

The first registered histogram is the denominator, all further histograms are treated as numerators.

Example

```
#!/usr/bin/env python

import ROOT
from MPF.efficiencyPlot import EfficiencyPlot
from MPF.atlasStyle import setAtlasStyle

setAtlasStyle()

passed = ROOT.TH1F("passed", "", 50, 100, 300)
total = ROOT.TH1F("total", "", 50, 100, 300)
passed.Sumw2()
total.Sumw2()
for i in range(10000):
    val = ROOT.gRandom.Gaus(100, 200)
    shift = ROOT.gRandom.Gaus(0, 20)
    total.Fill(val)
    if (val+shift) > 200:
        passed.Fill(val)

p = EfficiencyPlot(ratioMode="bayes")
p.registerHist(total)
p.registerHist(passed, legendTitle="Efficiency")
p.saveAs("plot.pdf")
```



class MPF.efficiencyPlot.**EfficiencyPlot**(*ratioTitle='Data / MC'*, ***kwargs*)

Bases: *MPF.plotStore.PlotStore*

Parameters

- **doDataMCRatio** – if both data and MC is added, also plot the data/MC ratio (default: False)
- **ratioModeDataMC** – ratioMode if data/MC is to be drawn default: "hist"
- **ratioTitle** – Title for the ratio pad if data/MC ratio is shown (default: "Data / MC")

Overwrites the defaults for the following *PlotStore()* parameters:

Parameters

- **ratioMode** – default: "binomial"
- **drawRatioLine** – default: False
- **yTitle** – default: #epsilon

For further options see *PlotStore()*

registerHist(*hist*, ***kwargs*)

Overwrites the defaults for the following *registerHist()* parameters:

Parameters

- **hide** – default: True
- **style** – default: "signal"
- **markerStyle** – default: 20

For further options see *registerHist()*

saveAs (*path*, ***kwargs*)
Save the canvas. Arguments are passed to *MPF.canvas.Canvas.saveAs()*

MPF.treePlotter module

Interface to generate plots (see *plotStore*) directly from ROOT TTrees.

Options for plotting can already be set when the *TreePlotter* is initialised. They will serve as defaults for any plot that doesn't set these options:

```
from MPF.treePlotter import TreePlotter

tp = TreePlotter(cut=myCut, weight=myWeight)
```

All options that don't correspond to *TreePlotter* or *ProcessProjector* will be passed to the *plot* (*plotType*) that is created in the end. Defaults can also be changed at any time by calling *setDefaults()*.

Before creating plots, add several processes to it:

```
tp.addProcessTree(processName, treePath, treeName)
```

All further options are passed to *Process()* - to be used later for *registerHist()*.

You can also explicitely create the *Process()* and add it. By doing so you can assign multiple trees in multiple files to one process:

```
from process import Process

p = Process(processName)
p.addTree(treePath1, treeName1)
p.addTree(treePath2, treeName2)
tp.addProcess(p)
```

Finally plots are created by calling *plot()*:

```
tp.plot(outputName)
```

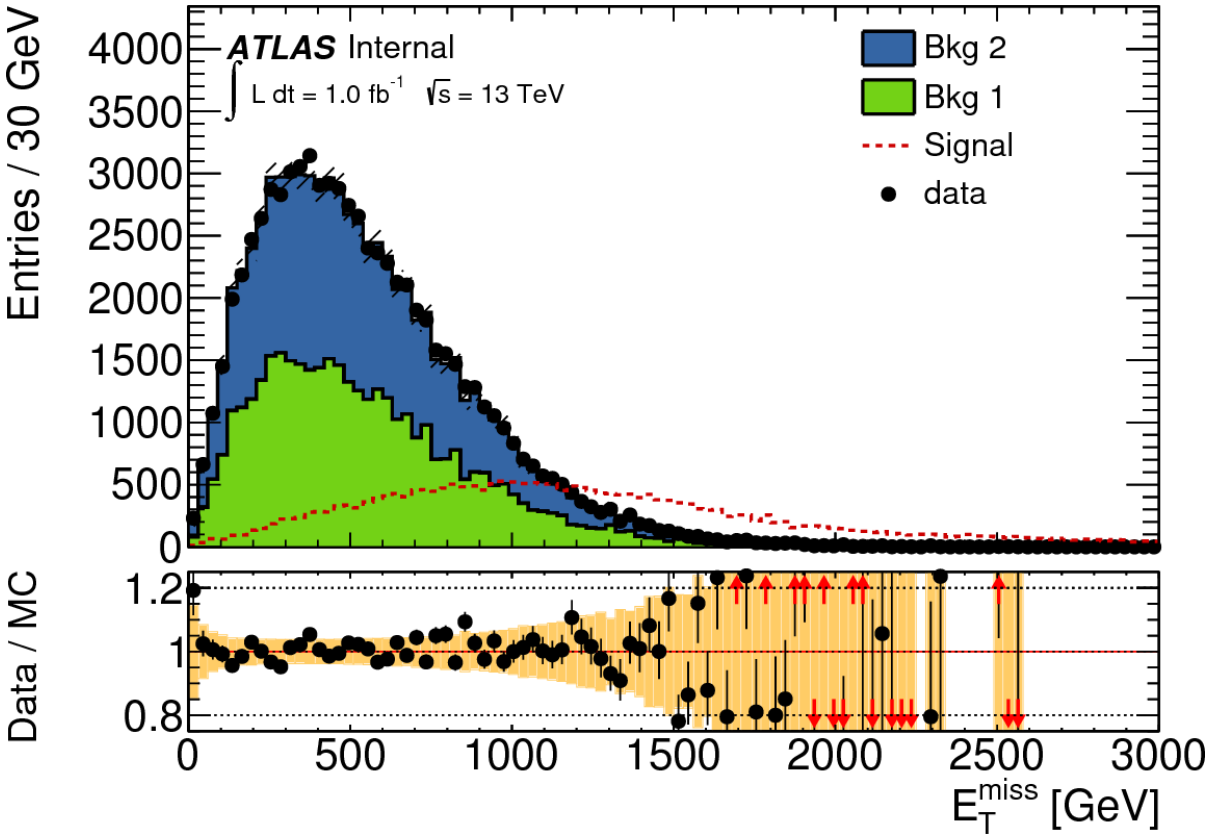
Further options will overwrite the default ones for this particular plot.

Instead of running *plot()* you can also run *registerPlot()* to create all plots at once later with *plotAll()*. This will use *multiHistDrawer* by default to loop over each tree only once. For example:

```
tp.registerPlot(outputFilename1, varexp=var1, xmin=xmin1, xmax=xmax1, nbins=nbins1)
tp.registerPlot(outputFilename2, varexp=var2, xmin=xmin2, xmax=xmax2, nbins=nbins2)
tp.plotAll()
```

Example

```
#!/usr/bin/env python

import os

import ROOT
from MPF.examples import exampleHelpers
from MPF.treePlotter import TreePlotter
from MPF.process import Process

testTreePath = "/tmp/testTreeMPF.root"

if not os.path.exists(testTreePath):
    exampleHelpers.createExampleTrees(testTreePath)

# Default Options can already be set when the plotter is initialized
tp = TreePlotter(cut="ht>1000", varexp="met", xTitle="E_{T}^{miss}", unit="GeV",␣
↪plotType="DataMCRatioPlot")

# Processes and their corresponding trees can be added via addProcessTree
tp.addProcessTree("Bkg 2", testTreePath, "w2", style="background")
tp.addProcessTree("Signal", testTreePath, "w4", style="signal")

# Processes can have custom cuts that are only applied for them
# Here we use this to scatter the background a bit, so our "data"
# which will be in the end from the same tree looks a bit more
# different ;)
tp.addProcessTree("Bkg 1", testTreePath, "w1", cut="(1-50*(1-w))", style="background")

# If multiple trees/files correspond to one process then you can
# explicitely create the Process object and add it to the plotter
data = Process("data", style="data")
data.addTree(testTreePath, "w1")
data.addTree(testTreePath, "w2")
tp.addProcess(data)

# options given to plot are only valid for the particular plot (and temporarily can␣
↪overwrite the defaults)
tp.plot("plot.pdf", xmin=0., xmax=3000, nbins=100, ratioUp=1.25, ratioDown=0.75)
```



class MPF.treePlotter.**TreePlotter**(***kwargs*)

Bases: *MPF.processProjector.ProcessProjector*

Generate plots from trees

Parameters

- **plotType** – Name of the Plot class to be created - see *PlotStore()* for a list
- **legendOptions** – dictionary of options to be passed to *legend()*
- **globalStyle** – Dictionary of *globalStyle* options that should be used for the plots

You can pass *ProcessProjector* arguments:

Parameters

- **cut** – Cut expression to be applied for all registered processes (default: “1”)
- **weight** – Weight expression to be applied for all registered processes (default: “1”)
- **varexp** – Expression to be used for filling histograms (default: “1”)
- **inputLumi** – luminosity the trees are normalised to
- **targetLumi** – luminosity the histograms should be scaled to
- **xmin** – minimum on the x axis
- **xmax** – maximum on the x axis
- **nbins** – number of bins
- **binLowEdges** – list of low edges of bins (in this case xmin, xmax and nbins are ignored)

- **useMultiHistDraw** – use *multiHistDrawer()* when calling *fillHists()* (loop tree only once and fill all histograms) (default: True)
- **cutsDict** – if this is given, fetch only yields for all cuts and create a histogram and yieldsDict for each process

All further arguments are passed to *PlotStore()* (or the plot class you are creating by *plotType*)

All parameters can also be set when calling *plot()* or *registerPlot()* - in that case they are only valid for this particular plot and temporarily overwrite the defaults. The defaults can be also overwritten by calling *setDefaults()*

addProcess(*process*)
Add a *Process()*

addProcessTree(*name*, *filename*, *treename*, ***kwargs*)
Create and add a process from one tree in one file. The kwargs are passed to *Process()*:

Parameters to be used for *registerHist()* for histograms created from the process:

Parameters

- **style** – (default: "background")
- **color** – (default: None)
- **lineColor** – (default: None)
- **markerColor** – (default: None)
- **fillColor** – (default: None)
- **lineStyle** – (default: None)
- **lineWidth** – (default: None)
- **fillStyle** – (default: None)
- **markerStyle** – (default: None)
- **drawErrorBand** – (default: False)
- **drawString** – (default: None)
- **legendTitle** – (default: None)
- **drawLegend** – (default: True)
- **ratioDenominatorProcess** – (default: None)
- **stackOnTop** – (default: False)

The other parameters:

Parameters

- **cut** – cut/weight expression to be used only for this process
- **norm** – normalise the resulting histograms to unity?
- **scale** – scale resulting histogram by this factor
- **varexp** – use this varexp for this process **instead** of the one used for all the other processes
- **normToProcess** – normalise the histogram to the same integral as the given process (by name) before plotting (only used for hists of style "systematic" in *TreePlotter*)

- **sysTag** – name of the systematic variation (mainly for internal use in *ProcessProjector* and *TreePlotter*)
- **noLumiNorm** – don't normalise this process to the luminosity configured

addSysTreeToProcess(*nomProcessName*, *sysName*, *filename*, *treename*, ***kwargs*)
Create and add a process from one tree in one file and register it as a systematic variation for the nominal process. The kwargs are passed to *Process()*

Parameters

- **nomProcessName** – name of the nominal process
- **sysName** – name of the systematic variation
- **treename** – name of the tree
- **filename** – path to the rootfile containing the tree
- **normToProcess** – normalise the histogram to the same integral as the given process (by name) before plotting (only used in *TreePlotter*).

defaults = {'legendOptions': None, 'plotKwargs': None, 'plotType': 'Plot'}

fillHists(*opt=None*)
Project histograms for all processes

Parameters **opt** – if given use these options instead of the current ones (see *getOpt()*)

fillHistsSysErrors()
Adds errors based on variational histograms for all processes to their histograms. Should only be used if variations are not correlated across different processes (e.g. **don't** use it for *TreePlotter* - there is a treatment included for this via *registerSysHist()*)

fillYieldsDicts(*opt=None*)
Fill yields dicts from cutsDict

Parameters **opt** – if given use these options instead of the current ones (see *getOpt()*)

getHists(*processName*, **args*, ***kwargs*)
Retrieve all histograms from a previous call of *plotAll()* for the given process name matching all the given options

Parameters **processName** – name of the process

Example:

```
tp.registerPlot("plot1.pdf", varexp="met", xmin=0, xmax=50)
tp.registerPlot("plot2.pdf", varexp="met", xmin=10, xmax=50)
tp.registerPlot("plot3.pdf", varexp="mt", xmin=0, xmax=150)
tp.plotAll()

# will give 2 hists
for hist in tp.getHists("bkg 1", varexp="met"):
    # do something with hist
    pass

# will give 1 hist
for hist in tp.getHists("bkg 1", varexp="met", xmin=0):
    # do something with hist
    pass

# will give 1 hist
```

(continues on next page)

(continued from previous page)

```
for hist in tp.getHists("bkg 1", "plot3.pdf")
    # do something with hist
    pass
```

Note: This is a generator - if the options or process name match multiple histograms, all of them are yielded. If you are sure only one matches you can do:

```
hist = next(tp.getHists(processName, **myOptions))
```

getOpt (***kwargs*)
Get the namedtuple containing the current *ProcessProjector()* options, updated by the given arguments.

getProcess (*processName*)

getProcesses (**selection*)

plot (*savename*, ***kwargs*)
Create the plot with the current options - updating with the given *TreePlotter()* parameters. The plot will be saved as the given savename.

In addition the following parameters can be given:

Parameters **noSave** – Don't save the plot and draw the *Canvas()* with the "noSave" option. The underlying plot can be accessed via TreePlotter.plotStore (default: False)

plotAll (*useMultiHistDraw=True*, *compile=False*)
Finally create all registered plots

Parameters

- **useMultiHistDraw** – Use *MultiHistDrawer()* to fill the histograms (default: True)
- **compile** – When using *MultiHistDrawer()* use the compile option (usually not recommended)

registerPlot (**args*, ***kwargs*)
Register a plot to be plotted later. The arguments corresponding to this plot will be stored and passed to *plot()* when the plot is created

registerSysToPlot (*plot*)
Register all systematic histograms to the plot

registerToPlot (*plot*, *opt=None*)
Register all process histograms to the given plot

registerToProjector (**selection*, ***kwargs*)
Register hists for each process to the histProjector (to be filled later with multiHistDraw).

Mainly for internal use in *registerPlot()* and *registerHarvestList()*

Parameters

- **opt** – namedtuple containing *ProcessProjector()* options
- **selection** – only register processes of this style(s) (like "background")

setDefaults (***kwargs*)
Update the defaults for the *TreePlotter()* parameters

setProcessOptions(*processName*, ***kwargs*)
Change the options of an existing process - referenced by its name. Only change the given options, leave the existing ones

MPF.n1plotter module

Create "n-1" plots (plots where a set of cuts is applied except the cut on the plotted distribution). This is in particular useful for signal region optimization when used together with *SignificanceScanPlot*. However, you can create every plot that *TreePlotter* can create.

To create multiple n-1 plots first set up a TreePlotter in the usual way - for Example:

```
from MPF.treePlotter import TreePlotter

tp = TreePlotter(plotType="SignificanceScanPlot", inputLumi=0.001, targetLumi=36.1,
                 cut="lep1Pt>35&&nLep_base==1&&nLep_signal==1",
                 weight="eventWeight*genWeight")
```

Afterwards add your backgrounds and signals in the usual way (have a look at *treePlotter*).

Then create the N1Plotter and pass it your *TreePlotter*:

```
from MPF.n1plotter import N1Plotter

np = N1Plotter(tp)
```

The output format can also be specified - for example:

```
np = N1Plotter(tp, outputFormat="<mydir>/{name}.pdf")
```

Now you can add several cuts (for each of which a plot should be created). You can give arbitrary custom plotting options (see *TreePlotter*) for each cut. For Example:

```
np.addCut(varexp="met", value=350,
          plotOptions=dict(xmin=0, xmax=1000, nbins=20))
np.addCut(varexp="met/meffInc30", value=0.1, name="met_over_meff",
          plotOptions=dict(xmin=0, xmax=0.4, nbins=50))
```

If you choose to change a few options (like the cut value) afterwards you can do that by using *getCut()*. For Example:

```
np.getCut("met").value = 200
np.getCut("met_over_meff").value = 0.2
```

Finally create all the plots:

```
np.makePlots()
```

Or, alternatively, register them and plot them at once (using *multiHistDrawer*):

```
np.makePlots(registerOnly=True)
tp.plotAll()
```

class MPF.n1plotter.**Cut**(*varexp*, *value*, *comparator='>'*, *name=None*, *plotOptions=None*, *removeCuts=None*)

Parameters

- **varexp** – the expression to be plotted/cut on
- **value** – the cut value
- **comparator** – how to compare the varexp to the value?
- **name** – name for the cut (will be used to name the output file and for referencing - see *getCut()*). If no name is given, the varexp will be used.
- **removeCuts** – when creating the n-1 expression, in addition also remove these cuts
- **plotOptions** – dictionary of arbitrary options for the plot that corresponds to this cut

name

class MPF.n1plotter.**N1Plotter**(*plotter*, *outputFormat='{name}.pdf'*)

addCut(**args*, ***kwargs*)
Add a cut for which an n-1 plot should be created. The arguments are passed to *Cut*.

getCut(*name*)
Get the cut referenced by the given name. You can use this to change the options for a particular cut. For example:

```
np.getCut("met").value = 200
```

getN1Expr(**names*)
Return cut expression for all cuts except the ones referenced by the given names and the cuts that are supposed to be removed with them

makePlots(*registerOnly=False*)
Create all plots. If registerOnly is set, then the plots are only registered to the *TreePlotter* and can be plotted later by using *plotAll()*

Returns the list of filenames plotted (or to be plotted)

6.1.2 Subpackages

MPF.commonHelpers package

Submodules

MPF.commonHelpers.interaction module

MPF.commonHelpers.interaction.**humanReadableSize**(*size*, *suffix='B'*)

class MPF.commonHelpers.interaction.**outputHelper**(*rows*)
Bases: object

printTable()

MPF.commonHelpers.interaction.**printTable**(*table*)
takes list of rows and turns it to printable string

MPF.commonHelpers.interaction.**promptInput**(*message*)

MPF.commonHelpers.interaction.**promptYN**(*text*, *trials=3*)

MPF.commonHelpers.logger module

Logger template

MPF.commonHelpers.logger.**done**()

MPF.commonHelpers.options module

MPF.commonHelpers.options.**checkUpdateDict**(*optDict*, ***kwargs*)
Returns a dict containing the options from optDict, updated with kwargs. Raises typeError if an option in kwargs doesn't exist in optDict

MPF.commonHelpers.options.**checkUpdateOpt**(*optDict*, ***kwargs*)
Returns a namedtuple containing the options from optDict, updated with kwargs. Raises typeError if an option in kwargs doesn't exist in optDict

MPF.commonHelpers.options.**popUpdateDict**(*optDict*, ***kwargs*)
Returns a dict containing the options from optDict, updated with kwargs and the remaining kwargs that don't exist in optDict

MPF.commonHelpers.options.**setAttributes**(*obj*, *optDict*, ***kwargs*)

MPF.commonHelpers.options.**updateAttributes**(*obj*, *optDict*, ***kwargs*)

MPF.commonHelpers.pathHelpers module

MPF.commonHelpers.pathHelpers.**checkPath**(*path*)

MPF.commonHelpers.pathHelpers.**cleanAndCheckPath**(*path*)

MPF.commonHelpers.pathHelpers.**cleanPath**(*path*)

MPF.commonHelpers.pathHelpers.**ensurePathExists**(*path*, *ask=False*)

MPF.commonHelpers.pathHelpers.**filesInDir**(*directory, patterns=[''], matchAll=True*)

Module contents

MPF.examples package

Submodules

MPF.examples.exampleHelpers module

MPF.examples.exampleHelpers.**createExampleSignalGrid**(*filename*, *nevents=1000*, *m1range=(1000, 2000, 100)*, *m2range=(1000, 2000, 100)*)

MPF.examples.exampleHelpers.**createExampleTrees**(*filename*)

MPF.examples.exampleHelpers.**histogramFromList**(*myList*, *nbins*)

MPF.examples.exampleHelpers.**loadExampleTreeCode**()

MPF.examples.exampleHelpers.**parseExampleArgs**()

MPF.examples.exampleHelpers.**randHist**()

MPF.examples.exampleHelpers.**sampleFunc**(*func*, ***kwargs*)

Module contents

6.1.3 Module contents

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

m

Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X

Y