pyroot-plotscripts framework

Documentation

September 30, 2018

Contents

1 Plottingscripts

1.1 LimitsAll_v20.py

The following preprocessing steps are executed:

- **initialization of analysisConfig:**
  `analysis = analysisClass.analysisConfig`
  `analysis.initArguments`
  `analysis.initAnalysisOptions`
  `analysis.initPlotConfig`
- **preparation of configData:**
  `configData = configClass.configData`
  `configData.initData`
  `configData.genDiscriminatorPlots`
- **definition of additional variables:**
  `configData.getAddVariables`
  `configData.getMEPDFAddVariables`
  `configData.getAdditionalDiscrimiatorPlots`
- **initialization of samples:**
  `configData.initSamples`

If no `plotNumber` is chosen (i.e. the script is executed normally), the following steps can be enabled independently:

- **plotParallel:**
  `pP = plotParallel.plotParallel`
  `pP.setJson`
  `pP.setDataBases`
  `pP.setAddInterfaces`
  `pP.setCatNames`
  `pP.setCatSelections`
  `pP.setMaxEvts`
  `pP.run`
  `pP.checkHaddFiles`
  `pP.checkTermination`
- **optimizedRebinning:**
  `optBinning.optimizeBinning`
- **renameHistograms:**
  `pP.setRenameInput`
  `renameHistos.renameHistos`
  `pP.addData`

- **makeDatacards:**
  `makeDatacards.makeDatacardsParallel`
- **drawParallel:**
  `drawParallel.drawParallel`

If a `plotNumber` is chosen (i.e. the script is executed via `drawParallel` or is a single execute script), the following steps can be enabled independently:

- **drawParallel:**
  `configData.getDiscriminatorPlotByNumber`
- If any of the following options is activated, the first step is creating the needed lists:
  `gP = genPlots.genPlots`
  `gP.genList`
- **makeSimplePlots:**
  `gP.makeSimpleControlPlots`
  `gP.makeSimpleShapePlots`
- **makeMCControlPlots:**
  `gP.genNestedHistList`
  `gP.makeControlPlots`
- **makeEventYields:**
  `gP.makeEventYields`

2 Util

2.1 class `analysisConfig.analysisConfig`

The `analysisConfig` class is used to store most of the settings for the plotting processes. Some options are used for enabling certain steps of the script, some options are used for skipping certain steps under certain circumstances and other options are used to store paths or variables, which are used throughout the script.

2.1.1 *init*

`analysisConfig(*args, signalProcess = "ttbb", discrName = "finaldiscr")`

workdir	absolute path to working directory determines the place where all output is stored.
pyrootdir	absolute path to `pyroot-plotscripts` directory.
rootPath	absolute path to desired output root file. determines in- and outputs during different steps of the script a reasonable choice is `<workdir>L/output_limitInput.root`.
signalProcess	name of the chosen signal process (default `ttbb`) determines plot configuration and position of ttH-samples viable choices are `ttbb`, `ttH` or `DM`.
discrName	name of the discrimnator (default `finaldiscr`) used to determine the discriminator plots.

2.1.2 *initArguments*

`analysisConfig.initArguments(argv = list())`

argv	list of arguments, usually uses `sys.argv` as input.

2.1.3 *initAnalysisOptions*

`analysisConfig.initAnalysisOptions(analysisOptions = {})`

analysisOptions	dictionary of options, mostly booleans.

The possible options and their explanations are summarized in table 1

2.1.4 *initPlotConfig*

`analysisConfig.initPlotConfig()`
takes the plotconfig determined by the signal process and imports the config in `<pyrootdir>/configs/` directory as a `analysisConfig` intern module.

2.1.5 *setLimitPath*

`analysisConfig.setLimitPath(**kwargs)`

`name`	part of the name for the ROOT file used as input for limit calculations. is determined via `plotParallel.ppRootPath_<name>.root`. This (default `limitInput`).

Table 1: **analysis options.**

option	default	
`plotParallel`	True	activate `plotParallel` step
`drawParallel`	True	activate `drawParallel` step
`makeEventYields`	True	activate `makeEventYields` step
`makeDataCards`	True	activate `makeDataCards` step
`makeSimplePlots`	True	activate `makeSimplePlots` step
`makeMCControlPlots`	True	activate `makeMCControlPlots` step
`optimizedRebinning`	""	activate `optimizedRebinning` step
`additionalPlotVariables`	[]	add more variables
`plotNumber`	None	set `plotNumber` variable for `drawParallel`
`singleExecute`	False	execute drawing steps without `drawParallel`
`cirun`	False	fast test run with less events
`plotBlinded`	False	perform plotting steps blinded
`useOldRoot`	False	use the existing root file in workdir
`stopAfterCompile`	False	stop script after compiling the cpp program
`skipPlotParallel`	False	try to skip `plotParallel` NAF submission
`skipHaddParallel`	False	try to skip `haddParallel` NAF submission
`skipHaddFromWildcard`	False	try to skip `haddFromWildcard` NAF submission
`skipRenaming`	False	try to skip `parallelRenaming` NAF submission
`skipDatacards`	False	try to skip `makeDatacards` NAF submission

funcion is used before the `renameHistos` step. The path is set and it is checked, wheter it already exists. If it does, the `renameHistos` step is skipped, as the output is already present.

2.1.6 *getPlotPath*

`analysisClass.getPlotPath()`
Returns path to `<workdir>/outputPlots` if `analysisConfig.plotNumber` is specified. Otherwise returns nothing.

2.2 class `configClass.configData`

The `configData` class is used to initialize the data needed for the main steps of the script. It handles the initialization of discriminator plots and samples.

2.2.1 *init*

`configData(*args, configDataBaseName = "")`

`analyisClass`	fully initialized instance of `analysisClass`
`configDataBaseName`	name of config files in `<pyrootdir>/configs/` directory for example `controlPlotsv13`.

2.2.2 *initData*

`configData.initData()`
Initializes an instance of `catData` class, which is used to store plots. It consists of following lists:

- `discrs`
- `nhistobins`
- `minxvals`
- `maxxvals`
- `categories`
- `plotPreselections`
- `binlabels`

2.2.3 *genDiscriminatorPlots*

`configData.genDiscriminatorPlots(*args)`

`memexp`	string type expression for MEM-variables.

Imports `<pyrootdir>/configs/<basename>_plots` as config file for generating the discriminator plots and calls its functions `getDiscriminatorPlots` and `evtYieldCategories`, thereby creating the lists `discriminatorPlots` and `evtYieldCategories`.
For the documentation of `<pyrootdir>/configs/<basename>_plots` files see 4.1.

2.2.4 *writeConfigDataToWorkdir*

`configData.writeConfigDataToWorkdir()`
Creates csv file `<workdir>/configData.csv` and writes content of `catData` class to it.

2.2.5 *getAddVariables*

`configData.getAddVariables()`
Imports `<pyrootdir>/configs/<basename>_addVariables` as config file for generating the discriminator plots and calls its function `getAddVars`, thereby creating the list `addVars`, containing additional variables which will be considered when writing the cpp file.
For the documentation of `<pyrootdir>/configs/<basename>_addVariables` files see 4.2.

2.2.6 *getMEPDFAddVariables*

`configData.getMEPDFAddVariables(*args)`

`csvfile`	path to csv-type file containing names, weights and factors for matrix element pdf variables.

Adds more variables to `addVars` list with `GetMEPDFadditionalVariablesList` function from `PDFutils` (see 3.3).

Table 2: **sample lists.**

<basename>_samples function	content
getSamples()	samples from pltcfg
getControlSamples()	control samples from pltcfg
getSystSamples()	systematic samples from pltcfg
getAllSamples()	list of samples written to cpp program
getAllSystNames()	names of samples, e.g. for renameHistos step
getWeightSystNames()	names of systematic weights, e.g. for plotParallel
getOtherSystNames()	other systematic names, e.g. for plotParallel
getSystWeights()	list of systematic weights, e.g. for plotParallel

2.2.7 *getAdditionalDiscrimiatorPlots*

```
configData.getAdditionalDiscrimiatorPlots(alwaysExecute = False)
```

alwaysExecute	forces the script to continue, even if no additionalPlotVariablesMap was found.

Is only executed, if analysis.additionalPlotVariables is set to True.
Creates list of plot for additional input variables. The function checks, if the file additionalPlotVariablesMap.py already exists.
If yes, it tries to determine the number of bins and plot range from that file.
If no, it will construct a dictionary and save it to the file and exit.
When additional plots are found, it extends the previously created discriminatorPlots list.

2.2.8 *initSamples*

```
configData.initSamples()
```

Imports <pyrootdir>/configs/<basename>_samples as config file for generating the needed samples and calls the function in table 2.

For the documentation of <pyrootdir>/configs/<basename>_samples files see 4.3

2.2.9 *getDiscriminatorPlotByNumber*

```
configData.getDiscriminatorPlotByNumber()
```

Sets the variable plotParallel.discriminatorPlotByNumber depending on analysisClass.plotN
This should be used before calling the drawParallel class init, such that the plots for the appropriate discriminator plots can be created.

2.2.10 *getDiscriminatorPlots*

```
configData.getDiscriminatorPlots()
```

Returns a single discriminator plot as list when analysisClass.plotNumber is specified.
Returns all discriminator plots from configData.discriminatorPlots when no plot number is specified.

2.3 class `plotParallel.plotParallel`

2.3.1 *init*

`plotParallel.plotParallel(*args)`

`analysis`	`analysisClass` class instance.
`configData`	`configData` class instance.

Saves `analysisClass` and `configData` as members, sets `analysisClass.ppRootPath` default to `output.root`. Also initializes default values for the options, which can be adjusted via setter functions.

2.3.2 *setJson*

`plotParallel.setJson(*args)`

`jsonFile`	path to json file which stores tree information.

The json file can be used to speed up the counting of events in root trees, as it contains the tree names and contents.
Default value is `''`.

2.3.3 *setDataBases*

`plotParallel.setDataBases(*args)`

`dataBases`	list of lists of information about MEM databases.

Each list of information about a database contains its name, path and flag to skip non existing events.
The databases are included when writing the cpp program.
Default value is `[]`.

2.3.4 *setAddInterfaces*

`plotParallel.setAddInterfaces(*args)`

`interfaces`	list of paths to DNNInterfaces

Loads the interfaces as module and stores them in `addInterfaces`.
Default value is `[]`.

2.3.5 *setCatNames*

`plotParallel.setCatNames(*args)`

`categoryNames`	list of category names.

Default value is `['']`.

2.3.6 *setCatSelections*

`plotParallel.setCatSelections(*args)`

`categorySelections`	list of category selections.

Default value is `[1.]`.

2.3.7 *setMaxEvts*

`plotParallel.setMaxEvts(*args)`

`maxevts`	number of maximum events per file.

Default value is `50000000`.

2.3.8 *run*

`plotParallel.run()`
Environment for setting up the cpp programm with `scriptWriter` (see 3.4.2), creating the rename script (see 3.4.5), writing run scripts (see 3.4.6) and executing the run scripts (i.e. the cpp programm) on the NAF batch system (see 3.8.1).
Upon termination of the run script this function also calls the `haddParallelInterface` function to start adding the created histograms together.
With the previously defined flag `analysisClass.skipPlotParallel` the submission of the run scripts to the NAF batch system can be skipped if the output files of `plotParallel` are already present in the working directory.
With the previously defined flag `analysisClass.useOldRoot` the execution of `plotParallel` can be skipped, if an output root file already exists in the working directory. This options does not check the content of the root file, it only checks its existence.
With the previously defined flag `analysisClass.stopAfterCompile` the script can be stopped after successfully compiling the cpp program.

The following functions are called during the execution of `plotParallel.run()`:

- `scriptWriter.scriptWriter`
- `scriptWriter.writeCC`
- `scriptWriter.writeRenameScript`
- `scriptWriter.writeRunScripts`
- `nafInterface.plotInterface`
- `plotParallel.haddParallelInterface`

2.3.9 *haddParallelInterface*

`plotParallel.haddParallelInterface(*args)`

`writer`	instance if `scriptWriter` class.

Initializes a `haddParallel.haddParallel` class and runs it (see 2.4).

2.3.10 *checkTermination*

`plotParallel.checkTermination()`
Checks if the `plotParallel.finished` flag is set to True, which should happen, if the process has finished successfully.

2.3.11 *checkHaddFiles*

`plotParallel.checkHaddFiles()`
Checks, if the list `plotParallel.haddFiles` exists and returns True/False.

2.3.12 *setRenameInput*

`plotParallel.setRenameInput()`
Attempts to set an input for the `renameHistos` step. If `plotParallel.run` was executed the usual way, it produces a number of hadd files (`plotParallel.haddFiles`) which should be used as the input for `renameHistos`.
If `plotParallel.run` was not executed as usual the `analysisClass.limitPath` ROOT file should be used as an input.

2.3.13 *addData*

`plotParallel.addData(*args)`

`samples`	list of samples for which to add data.

Takes the nick names of all input samples and loops over the binlabels specified with `configData.binlabels` to add data to the histograms. (?)

2.4 class `haddParallel.haddParallel`

The `haddParallel` class handles the output ROOT files from `plotParallel` and adds the histograms of the ROOT files together. As this has to be done many times it is usually performed via the NAF HTC batch system as parallelly executed jobs.

2.4.1 *init*

`haddParallel.haddParallel(*args)`

`plotParallelClass`	instance of a `plotParallel` class to inherit member functions.

Initializes the `haddParallel` class and inherits the needed functions from `plotParallel`.

2.4.2 *run*

`haddParallel.run(*args)`

`writer`	instance of a `scriptWriter` class to write needed scripts.

Writes the python script `<workdir>/haddScript.py` with `scriptWriter.writeHaddScript`, which is later executed by jobs on the NAF batch system.
Loops over the `plotParallel.samplewiseMaps` created during `scriptWriter.writeRunScripts` and writes shell scripts with the `scriptWriter.writeHaddShell` function for all the samples.
Afterwards, it executes the scripts on the NAF batch system with `nafInterface.haddInterface` (see).
If the `analysisClass.skipHaddParallel` option was activated, the writing of scripts is skipped and only `nafInterface.haddTerminationCheck` is run, to check for the termination of all jobs. If not all jobs have terminated, the `run` function is called iteratively.
If the `analysisClass.haddParallel` option is deactivated, non parallel hadding is performed, but this option is not used anymore.

2.4.3 *haddSplitter*

`haddParallel.haddSplitter(*args, **kwargs)`

`input`	input ROOT files, either as string with wildcards or as list.
`outName`	path to desired output ROOT file.
`subName`	string of naming scheme for bookkeeping when splitting the files.
`nHistosRemainSame`	flag to decide whether number of histograms need to stay the same.
`skipHadd`	flag to enable skippint the hadding process.
`forceHadd`	flag to enable force hadding, thereby overwriting existing histograms.

This script is actually not a part of the `haddParallel` class but rather a standalone hadding function.
It takes a list of ROOT files (either as a list or with a wildcard path) and tries to add all histograms in those files together to form one single output ROOT file.
As the amount of files can be very large sometimes, it adds the input ROOT files in bulks, instead of adding all together at once. For every bulk of ROOT files it calls the function `callHadd`, which executes the `hadd` command for the given files. At the end, all the part-files are combined to the desired output.
If the flag `skipHadd` is activated it does not perform the hadding and only compares the amount of histograms before and after the hadding process and decides whether to redo the hadding process or proceed with the next step depending on `nHistosRemainSame`.

2.5 `renameHistos.py`

2.5.1 *renameHistosParallel*

`renameHistos.renameHistosParallel(*args, **kwargs)`

`inFile`	ROOT file created by cpp program execution.
`outFile`	ROOT file that is supposed to be created.
`systNames`	list of names of systematics considered.
`checkBins`	option to activate checking bins (default `False`).
`prune`	? (default `False`).
`plotParaCall`	indicator, whether the call of the function happened during `plotParallel` (default `False`).
`Epsilon`	? (default `0.0`).

Copies the `inFile` to `outFile` and loops over every key in the ROOT file.
Counts number of systematics per key, renaming the key, as well as removing histograms which have more than two systematics.
If the script is not called via `plotParallel` the bins are checked and adjusted.

2.5.2 *renameHistos*

`renameHistos.renameHistos(*args, **kwargs)`

`inFiles`	list of input ROOT files, usually determined by `plotParallel.setRenameInput(`
`outFile`	output ROOT file, usually determined by `analysisClass.limitPath`.
`systNames`	names of systematic uncertainties, usually determined by `configData.allSystNa`
`checkBins`	option to activate checking bins (default `False`).
`prune`	? (default `False`).
`Epsilon`	? (default `0.0`)
`skipRenaming`	option to skip the renaming process (default `False`).

If the function is called with a single file as input it directly calls `renameHistosParallel`. Otherwise, it writes a rename script with the `writeRenameScript` function and creates shell scripts for each input file which calls the created rename script. Then all the scripts are submitted to the NAF batch system via `nafInterface.renameInterface`. Upon successfull termination of these jobs, the function `haddSplitter` is called to hadd all the renamed ROOT files together to create the output ROOT file.
If the `analysisClass.skipRenaming` option is activated, it does not write the shell scripts and directly checks the already existing output via `nafInterface.renameTerminationCheck`. If all the scripts have terminated successfully, the submission is skipped, otherwise, the function is called iteratively (WIP).

2.5.3 *writeRenameScript*

`renameHistos.writeRenameScript(*args)`

`outFile`	name of output ROOT file of `renameHistos` function.
`skipRenaming`	option to activate skipping the writing of the script.

Writes a python script that calls the `renameHistos.renameHistosParallel` function for the jobs created in the `renameHistos` function.

2.6 class optBinning.optimizeBinning

2.6.1 *init*

optBinning.optimizeBinning(*args, **kwargs)

infname	input ROOT file.
signalsamples	list of samples declared as signal samples, this depends on the signal proces (analysisConfig.signalProcess).
backgroundsamples	list of samples declared as background samples, this depends on the signal (analysisConfig.signalProcess).
additionalSamples	list of additional samples also considered during the rebinning.
plots	list of discriminator plots.
systnames	list of considered systematics (e.g. configData.allSystNames).
minBkgPerBin	minimal number of events per bin.
optMode	type of optimization, determined by analysisClass.optimizedRebinning
considerStatUnc	flag to decide whether to also consider statistical uncertainties.
maxBins	maximum number of bins per histogram.
minBins	minimal number of bins per histogram.

Loops over all discriminator plots and all samples and adds them to lists for signal and background clones. Then calls the function getOptimizedBinEdges per discriminator plot to get the optimized bin edges.
Next it loops over all samples and systematics and rebins all the histograms.

2.6.2 *getOptimizedBinEdges*

optimizeBinning.getOptimizedBinEdges(*args, **kwargs)

signalHisto	clone of the signal histogram.
bkgHisto	clone of the background histogram.
optMode	type of optimization for the rebinning process.
minBkgPerBin	minimal number of events per bin.
maxBins	maximum number of bins per histogram.
minBins	minimal number of bins per histogram.
considerStatUnc	flag to decide whether to also consider statistical uncertainties.

Depending on the optimization mode this function searches for the optimal bin edges by analyzing the input histograms and returns the bin edges as a list.

2.7 makeDatacards.py

2.7.1 *makeDatacardsParallel*

`makeDatacards.makeDatacardsParallel(*args, **kwargs)`

filePath	path to input ROOT file to make the datacards, usually analysisClass.limitI
outPath	path to output folder for datacards, usually <workdir>/datacards/.
categories	list of categories to consider, usually configData.binlabels.
doHdecay	option to consider Higgs decays (default True).
discrname	name of discriminator (default finaldiscr).
datacardmaker	name of datacard maker (default mk_datacard_hdecay13TeVPara).
skipDatacards	option to skip making the datacards.

This function writes a shell script for each category which calls the actual datacard maker and submits them to the NAF batch system via nafInterface.datacardInterface. After termination it adds the bin-by-bin ROOT files, created during datacard making to the input file with haddBinByBinFiles.
If the option analysis.skipDatacards was activated, it skips the creation of the shell scripts and NAF submission and directly checks if the datacards already exist with nafInterface.datacardTerminationCheck. If all datacards exist, the submission is skipped, otherwise, the function is called iteratively.

2.7.2 *haddBinByBinFiles*

`makeDatacards.haddBinByBinFiles(*args)`

bbbFiles	list of bin-by-bin ROOT files created during makeDatacardsParallel.
filePath	path to ROOT file which was used to create the datacards.

Moves the bin-by-bin files to a separate folder <workdir>/binbybinfiles/ and adds the histograms in these files to the main ROOT file via hadd.

2.8 drawParallel.py

This function is used to manage drawing the actual histograms to pdf/etc files. This is done parallely via submission of multiple jobs to the batch system. The scripts created re-call the top-level script but with a specific plot number, thereby skipping to the genPlots dependent functions.

2.8.1 *drawParallel*

`drawParallel.drawParallel(*args, **kwargs)`

ListOfPlots	list of discriminator plots, usually configData.discriminatorPlots.
workdir	path to working directory, usually analysisClass.workdir.
PathToSelf	path to top-level script.
opts	analysis options, usually analysisClass.opts

Creates shell scripts with createDrawScripts for each discriminator plot, which are submitted to the batch system via nafInterface.drawInterface.

2.8.2 *createDrawScripts*

`drawParallel.createDrawScripts(*args, **kwargs)`

`iPlot`	number of discriminator plot to determine the `analysisClass.plotNumber`.
`Plot`	discriminator plot.
`PathToSelf`	path to top-level script.
`scriptPath`	path to shell script to be written.
`opts`	analysis options, usually `analysisClass.opts`

Writes the shell script to re-execute the top-level script for the specific discriminator plot.

2.9 class `genPlots.genPlots`

2.9.1 *init*

`genPlots.genPlots(*args)`

`outPath`	path to input ROOT file, usually `analysisConfig.limitPath`.
`plots`	list of discriminator plots, determined usually by `configData.getDiscriminatorPlots`
`plotdir`	output path of created plots, usually determined by `analysisClass.getPlotPath`.
`rebin`	option to perform rebinning (default `-1`).

Inherits the given arguments and creates empty dictionaries `lists`, `samples` and `nestedhistLists`, which are needed during the later steps.

2.9.2 *genList*

`genPlots.genList(*args, **kwargs)`

`samples`	list of samples for which the list class should be created.
`listName`	name for the class.
`catNames`	list of category names (default `['']`).
`doTwoDim`	option for two dimensional histograms `ROOT.TH2`.

Initiates a `genPlots.List` class instance which contains lists of histograms in different configurations, depending on the included samples and keys of the ROOT file.

2.9.3 *makeSimpleControlPlots*

`genPlots.makeSimpleControlPlots(*args, **kwargs)`

`dataConfig`	`genPlots.Config` class instance for configuration of used data.
`options`	dictionary of plotting options (see 3).

Loops over the histograms specified in `dataConfig` and proceeds to draw simple control histograms.

2.9.4 *makeSimpleShapePlots*

`genPlots.makeSimpleShapePlots(*args, **kwargs)`

`dataConfig`	`genPlots.Config` class instance for configuration of used data.
`options`	dictionary of plotting options (see 3).

Table 3: **options for simple plots.**

option	default	
`factor`	`-1`	
`logscale`	`False`	plot with logarithmic y-scale.
`canvasOptions`	`'histo'`	???
`normalize`	`False`	normalize y-scale.
`stack`	`False`	stack histograms.
`ratio`	`False`	add ratio plot.
`doProfile`	`False`	???
`statTest`	`False`	perform Kolmogorov and χ^2 tests.
`sepaTest`	`False`	calculate ROC-AUC value.
`blinded`	`True`	blind signal region.

Loops over the histograms specified in `dataConfig` and proceeds to draw simple shape comparison histograms.

2.9.5 *genNestedHistList*

`genPlots.genNestedHistList(*args)`

`dataConfig`	`genPlots.Config` class instance for configuration of used data.
`systNames`	list of names of systematic uncertainties to be considered when making control plots
`outName`	name of created object.

Prepares a nested list of histograms for use in `makeControlPlots`.
For every list created an entry should be added to a `nestedHistsConfig` containing the desired configurations of the plots.

2.9.6 *makeControlPlots*

`genPlots.makeControlPlots(*args, **kwargs)`

`dataConfig`	`genPlots.Config` class instance for configuration of data samples.
`controlConfig`	`genPlots.Config` class instance for configuration of control plots.
`sampleConfig`	`genPlots.Config` class instance for configuration of samples.
`headHist`	list which contains the first histogram.
`headSample`	list which contains the first sample.
`nestedHistsConfig`	config for nested histogram list.
`options`	dictionary of plotting options (see 3).

Creates super duper plots. TODO.

2.9.7 *makeEventYields*

`genPlots.makeEventYields(*args, **kwargs)`

`categories`	list of categories for which event yields should be calculated.
`listName`	list which contains the data of histograms.
`dataName`	list which contains some other data?.
`nameRequirement`	name requirement for histograms.

TODO.

3 Tools

3.1 class `plotClasses.Plot`

3.1.1 *init*

`plotClasses.Plot(*args, variable = "", selection = "", label = "")`

`histo`	`ROOT.TH1`-type instance which contains the plot.
`variable`	name of `discr`.
	if no argument is given, `histo.GetName()` is chosen as `variable`.
`selection`	name of `plotPreselection`.
`label`	name of `binlabel`.

The variables `discr`, `plotPreselection` and `binlabel` are usually defined in the `<pyrootdir>/configs/<basename>_plots` files (see 4.1).

3.2 class `plotClasses.Sample`

3.2.1 *init*

`plotClasses.Sample(*args, **kwargs)`

`name`	name of sample.
`color`	plotting color (default `ROOT.kBlack`).
`path`	path to samples, supports wildcards (default `''`).
`selection`	selection weight (default `''`).
`nick`	nick of sample (default `''`).
`listOfShapes`	list of shape samples (default `[]`).
`up`	? (default `0`).
`down`	? (default `None`).
`samDict`	`plotClasses.SampleDictionary` instance (default `''`).
`readTrees`	allows globbing samples from different paths (default `False`).
`filterFile`	file with filter information (default `'NONE'`).
`checknevents`	? (default `-1`).
`treename`	name of tree for ROOT file (default `'MVATree'`).

Saves the given informations as member variables. If `readTrees` option is activated also searches for samples in different paths.

3.3 `PDFutils.py`

3.3.1 *GetMEPDFadditionalVariablesList*

`GetMEPDFadditionalVariablesList(*args)`

`csvfile`	path to csv-type file
	containing names, weights and factors for matrix element pdf variables.

Reads csv file with `ReadMEandPDFNormalizations` function, scans dictionary for double entries and returns list of extracted weight variables.

3.4 class `scriptWriter.scriptWriter`

3.4.1 *init*

`scriptWriter.scriptWriter(*args)`

`plotParaClass`	instance of `plotParallel` class, which is inherited.

Inherits `plotParallel` class.

3.4.2 *writeCC*

`scriptWriter.writeCC()`
Main function for writing and compiling the cpp program. Calls the functions `createProgram` and `compileProgram`. When an old version of the cpp program was already present in the working directory, this function also checks, whether the newly created program differs from the old one.

3.4.3 *createProgram*

`scriptWriter.createProgram()`
Main function for writing the cpp programm.
First, generates a veto list from `<pyrootdir>/data/vetolist.csv` file of variables, that should not be written to the programm automatically. This also consideres a veto list for LHEWeights which is created from the `plotParallel.MEPDFCSVFile`, a veto list for DNNInterface variables defined in `plotParallel.addInterfaces` and a veto list for data bases which is created from the `plotParallel.dataBases`.
For the cpp file, the variables in `configData.allSamples` need to be initialized, which is done with the `variablebox.Variables` class (see 3.5). For this purpose a root tree is chosen to perform variable checks.

With the variable information multiple functions of `scriptfunctions.py` (see 3.7), etc are called:

- `scriptfunctions.getHead`
- `scriptfunctions.DeclareMEPFNormFactors`
- `scriptfunctions.ADDMEandPDFNormalizationsMap`
- `scriptfunctions.InitDataBase`
- `DNNInterface.getBeforeLoopLines`
- `variablebox.Variables.initVarsProgram`
- `variablebox.Variables.initBranchAdressesProgram`
- `variablebox.Variables.setupTMVAReadersProgram`
- `scriptfunctions.initHistos`
- `scriptfunctions.startLoop`
- `scriptfunctions.initMEPDF.writeCode`
- `DNNInterface.getVariableInitInsideEventLoopLines`
- `scriptfunctions.encodeSampleSelection`
- `scriptfunctions.readOutDataBase`
- `DNNInterface.getEventLoopCodeLines`
- `variablebox.Variables.calculateVarsProgram`

- scriptfunctions.initPlots.startCat
- scriptfunctions.initPlots.initPlot
- scriptfunctions.initPlots.endCat
- scriptfunctions.endLoop
- DNNInterface.getTestCallLines
- scriptfunctions.getFoot

After successfully writing the cpp program it is saved to <workdir>/<workdirname>.cc and compiled with compileProgram.

3.4.4 *compileProgram*

scriptWriter.compileProgram()
This function generates a compile command, depending on plotParallel.addInterfaces and plotParallel.dataBases and tries to execute it on the program created in createProgram. The script aborts, if the compilation was not successfull.

3.4.5 *writeRenameScript*

scriptWriter.writeRenameScript()
This function writes a short python script at <workdir>/<workdirname>_rename.py which calls the function renameHistos.renameHistosParallel upon execution.

3.4.6 *writeRunScripts*

scriptWriter.writeRunScripts()
This function handles the writing of scripts to execute the cpp program. It creates lists for scripts, output, nentries and a dictionary samplewiseMaps for bookkeeping purposes.
It loops over all samples in configData.allSamples and over all files in those samples, counting the events in each file (writing it to nentries) and calling the function writeSingleScript to write the shell script.
If the number of events in the file exceed the limit determined by plotParallel.maxevents, the file is split into parts and separate shell scripts are written for each part.
If the analysisClass.cirun option is activated, only the first file per sample is considered, thereby reducing the number of jobs and events drastically. This is only for testing purposes of the framework, the results will not be very meaningful.
This function returns the bookkeeping lists as a dictionary, to be used by the NAF submit interfaces.

3.4.7 *writeSingleScript*

`scriptWriter.writeSingleScript(*args, **kwargs)`

`sample`	current sample for which a script is written.
`filenames`	current file in sample for which a script is written.
`nJob`	counter of jobs to determine name of written shell file.
`filterFile`	filterFile of sample.
`writeOptions`	dictionary of options for writing the shell script includes for example `'skipEvents'` for large samples.

This script writes the a shell script to execute the cpp programm, previously created. In the shell script various environment variables are exported for use in the cpp program.
After calling the cpp program the shell script also calls the rename script created in `writeRenameScript`.
This script also adds the name of the shell script and the name of the output root file to the bookkeeping lists `scripts` and `outputs`, which were created in `writeRunScripts`.

3.4.8 *writeHaddScript*

`scriptWriter.writeHaddScript()`
Writes a python script to `<workdir>/haddScript.py` which takes multiple arguments:

- desired output ROOT file name
- desired location of log file
- ROOT files to be added together

All arguments except the first two specify the ROOT files whose histograms are added together via the bash command `hadd` designed for ROOT.
The script writes either `OK` or `ERROR` into the specified log file, such that the success of the hadd process can be judged by the `nafInterface.haddTerminationCheck` function.

3.4.9 *writeHaddShell*

`scriptWriter.writeHaddShell(*args)`

`scriptname`	name of shell script to be written.
`haddedRootName`	desired output ROOT file name.
`haddedLogName`	desired log file name.
`sampleData`	list of ROOT files to be added together.

Writes a shell script which is supposed to be submitted to the NAF batch system.
The script calls the `<workdir>/haddScript.py` script which was created with `scriptWriter.writeH`

3.5 class `variablebox.Variables`

3.5.1 *init*

`variablebox.Variables(veto = [])`

`veto`	previouly generated vetolist of plots that should not be considered by default.

Initializes a dictionary for variables and inherits the vetolist.

3.5.2 *initVars*

`variablebox.Variables.initVars(*args)`

`expr`	list or string of expressions for creating variables.
`tree`	ROOT tree to extract variables

If `expr` is a list, the function is called iteratively per list element.
Depending on the structure of the expression, the single variables are initialized with the `initSingleVar` function, where a variable is added to the `variables` dictionary created during *init* as a `variablebox.Variable` instance (see 3.6).
Only variables that are not in the vetolist or already in the variable dictionary are added to the dictionary.

3.5.3 *initVarsProgram*

`variablebox.Variables.initVarsProgram()`
Loops over all variables in `variables` dictionary and writes output of `variablebox.Variable.initV`
to cpp file.

3.5.4 *initBranchAddressesProgram*

`variablebox.Variables.initBranchAdressesProgram()`
Loops over all variables in `variables` dictionary and writes output of `variablebox.Variable.initB`
to cpp file.

3.5.5 *setupTMVAReadersProgram*

`variablebox.Variables.setupTMVAReadersProgram()`
Loops over all variables in `variables` dictionary and writes output of `variablebox.Variable.setup`
to cpp file.

3.5.6 *calculateVarsProgram*

`variablebox.Variables.caluclateVarsProgram()`
Sorts variable dictionary such that dependencies can be caluclated in the proper order, creates list of conditions for the variables and writes code for each variable considering the conditions and also writes output of `variablebox.Variable.calculateVarProgram`
to cpp file.

3.6 class variablebox.Variable

3.6.1 *init*

variablebox.Variable(*args, **kwargs)

name	name of variable
expression	extracted expression for variable if none is given it also takes name as input
vartype	type of variable as string, e.g. 'F' for float
arraylength	flag for array variables (default None

Saves input information as member variables.

3.6.2 *initVarProgram*

variablebox.Variable.initVarProgram()
Writes code for variable depending on vartype and arraylength to cpp file.

3.6.3 *initBranchAdressProgram*

variablebox.Variable.initBranchAdressProgram()
Writes code to set ROOT.TChain branch adresses to cpp file.

3.6.4 *setupTMVAReaderProgram*

variablebox.Variable.setupTMVAReaderProgram(*args)

variables	list of variables.

writes code from variablebox.Variable.initReaderProgram, variablebox.Variable.addVariab
and variablebox.Variable.bookMVAProgram to cpp file.
Disclaimer: this is untested and might not work.

3.7 scriptfunctions.py

TODO

3.8 nafInterface.py

This file contains one function <usecase>Interface for each use case, where the NAF batch system is called, which handles the submission to the batch system and performs checks on the output with the <usecase>TerminationCheck functions.
The use cases implemented are:

- plotParallel
- haddParallel
- renameHistos
- makeDatacards
- drawParallel

3.8.1 *plotInterface*

`nafInterface.plotInterface(*args, **kwargs)`

`jobData`	dictionary containing `scripts`, `outputs`, `entries` and `maps`, created in `scriptWriter.writeRunScripts`.
`skipPlotParallel`	option to skip the submission of scripts to the batch system. (default `False`).
`maxTries`	maximum number of resubmits after failing (default `10`).
`nTries`	counter for current try (default `0`).

The functions `nafSubmit.submitArrayToNAF` or `nafSubmit.submitToNAF` are called, depending on `nTries`.
After submission of the scripts with the `nafSubmit` functions, the job status is monitored with `nafSubmit.monitorJobStatus`.
Upon termination of all jobs, the output is checked with `plotTerminationCheck`. Scripts that did not pass the check are resubmitted to the batch system, by iteratively calling the `plotInterface` function.
If the `nTries` counter exceeds the `maxTries` threshold, the program is terminated.
If the `skipPlotParallel` option was activated, it skips directly to `plotTerminationCheck`, to check whether the `plotParallel` output is complete.

3.8.2 *plotTerminationCheck*

`nafInterface.plotTerminationCheck(*args)`

`jobData`	dictionary containing `scripts`, `outputs`, `entries` and `maps`, created in `scriptWriter.writeRunScripts`.

Loops over the jobs in `jobData` and crosschecks the number of entries in `<output>.cutflow.txt` (which is created by the cpp program) with the number of entries in `<entries>`.
If the number does not match or the cutflow file does not exist, the job is added to a resubmission list, which is returned at the end of the function.

3.8.3 *haddInterface*

`nafInterface.haddInterface(*args, **kwargs)`

`jobsToSubmit`	list of shell scripts to be submitted to the batch system.
`outfilesFromSubmit`	list of output ROOT files.
`maxTries`	maximum number of resubmits after failing (default `10`).
`nTries`	counter for current try (default `0`).

The functions `nafSubmit.submitArrayToNAF` or `nafSubmit.submitToNAF` are called, depending on `nTries`.
After submission of the scripts with the `nafSubmit` functions, the job status is monitored with `nafSubmit.monitorJobStatus`.
Upon termination of all jobs, the output is checked with `haddTerminationCheck`. Scripts that did non pass the check are resubmitted to the batch system, by iteratively calling the `haddInterface` function.
If the `nTries` counter exceeds the `maxTries` threshold, the program is terminated.

3.8.4 *haddTerminationCheck*

`nafInterface.haddTerminationCheck(*args)`

`outputScripts`	list of shell scripts that were submitted to the batch system.
`outputFiles`	list of output ROOT files.

The log files which were created during the run of `<workdir>/haddPara.py` are checked. These contain either `OK` or `ERROR`, depending on the success of the hadding process.
If the status is not `OK` or the log file is missing, the job is added to a resubmission list, whcih is returned at the end of the function.

3.8.5 *renameInterface*

`nafInterface.renameInterface(*args, **kwargs)`

`jobsToSubmit`	list of shell scripts to be submitted to the batch system.
`outfilesFromSubmit`	list of output ROOT files.
`maxTries`	maximum number of resubmits after failing (default `10`).
`nTries`	counter for current try (default `0`).

The functions `nafSubmit.submitArrayToNAF` or `nafSubmit.submitToNAF` are called, depending on `nTries`.
After submission of the scripts with the `nafSubmit` functions, the job status is monitored with `nafSubmit.monitorJobStatus`.
Upon termination of all jobs, the output is checked with `renameTerminationCheck`. Scripts that did not pass the check are resubmitted to the batch system, by iteratively calling the `renameInterface` function.
If the `nTries` counter exceeds the `maxTries` treshold, the program is terminated.

3.8.6 *renameTerminationCheck*

`nafInterface.renameTerminationCheck(*args)`

`shellScripts`	list of shell scripts that were submitted to the batch system.
`outputFiles`	list of output ROOT files.

The ROOT files which were created during the batch jobs are checked. If they do not exist, the responsible job is added to a resubmission list, which is returned at the end of the function.

3.8.7 *datacardInterface*

`nafInterface.datacardInterface(*args)`

`jobsToSubmit`	list of shell scripts to be submitted to the batch system.
`datacardFiles`	list of datacards to be created.
`maxTries`	maximum number of resubmits after failing (default `10`).
`nTries`	counter for current try (default `0`).

The functions `nafSubmit.submitArrayToNAF` or `nafSubmit.submitToNAF` are called, depending on `nTries`.

After submission of the scripts with the `nafSubmit` functions, the job status is monitored with `nafSubmit.monitorJobStatus`.
Upon termination of all jobs, the output is checked with `datacardTerminationCheck`.
Scripts that did not pass the check are resubmitted to the batch system, by iteratively calling the `datacardInterface` function.
If the `nTries` counter exceeds the `maxTries` treshold, the program is terminated.

3.8.8 *datacardTerminationCheck*

`nafInterface.datacardTerminationCheck(*args)`

`shellScripts`	list of shell scripts that were submitted to the batch system.
`datacardFiles`	list of datacards that should have been created.

The datacards which were created during the batch jobs are checked. If they do not exist, the responsible job is added to a resubmission list, which is returned at the end of the function.

3.8.9 *drawInterface*

`nafInterface.drawInterface(*args)`

`jobsToSubmit`	list of shell scripts to be submitted to the batch system.
`outputPlots`	list of discriminator plots for which the output plots should be created.
`nTries`	counter for current try (default `0`).

The functions `nafSubmit.submitArrayToNAF` or `nafSubmit.submitToNAF` are called, depending on `nTries`.
After submission of the scripts with the `nafSubmit` functions, the job status is monitored with `nafSubmit.monitorJobStatus`.
Upon termination of all jobs, the output is checked with `drawTerminationCheck`.
Scripts that did not pass the check are resubmitted to the batch system, by iteratively calling the `drawInterface` function.

3.8.10 *drawTerminationCheck*

`nafInterface.drawTerminationCheck(*args)`

`jobsToSubmit`	list of shell scripts that were submitted to the batch system.
`outputPlots`	list of discriminator plots for which the output plots should have been created.

This is not implemented yet.

3.9 `nafSubmit.py`

The functions in this file handle the actual submission of jobs to the NAF batch system.

3.9.1 *submitToNAF*

`nafSubmit.submitToNAF(*args, **kwargs)`

`scripts`	list of shell scripts to be submitted.
`holdIDs`	list of IDs of jobs that need to be finished before queueing the new scripts (default `None`).
`submitOptions`	dictionary of submitOptions (see 3.9.4).

Calls `writeSubmitCode` to write submit code for the HTC system for every script and submits those scripts with the `condorSubmit` function.
If the `holdIDs` option was used, it also writes a release script with `setupRelease` which releases the other scripts to the batch system, as soon as the conditions are fulfilled.
It returns the list of `jobIDs` of the submitted jobs.

3.9.2 *submitArrayToNAF*

`nafSubmit.submitArrayToNAF(*args, **kwargs)`

`scripts`	list of shell scripts to be submitted.
`arrayName`	desired name of the array file (default `''`).
`holdIDs`	list of IDs of jobs that need to be finished before queueing the new scripts (default `None`).
`submitOptions`	dictionary of submitOptions (see 3.9.4).

Calls `writeArrayCode` to write the shell script that handles the submission of the scripts as an array and writes a submit code for that array script with `writeSubmitCode` and submits the script with the `condorSubmit` function.
If the `holdIDs` option was used, it also writes a release script with `setupRelease` which releases the other scripts to the batch system, as soon as the conditions are fulfilled.
It returns the list of `jobIDs` of the submitted jobs.

3.9.3 *writeArrayCode*

`nafSubmit.writeArrayCode(*args)`

`scripts`	list of scripts to be concatenated as an array.
`arrayName`	desired name of the array file.

When submitting multiple scripts, one can use the array functionality, where instead of submitting all scripts independently, all scripts are submitted at once with an array script that manages the submit with a task ID which is being iterated.
This function writes the array script for this purpose and returns the path of the file.

Table 4: **submit options.**

option	default	
`RequestMemory`	`1000M`	requested amount of memory for slot.
`RequestDisk`	`1000M`	requested amount of disk for slot.
`+RequestRuntime`	`4800`	requested runtime in seconds.
`PeriodicHold`	`1000`	put job in hold status after `x` seconds.
`PeriodicRelease`	`5`	release job from hold status after `x` seconds.

3.9.4 *writeSubmitCode*

`nafSubmit.writeSubmitCode(*args, **kwargs)`

`script`	path to shell script for which the code is written.
`logdir`	path to directory for logfiles.
`hold`	option to initiate job in hold state (default `False`).
`isArray`	indicator, wheter the shell script is for an array job (default `False`).
`nScripts`	number of jobs, if the shell script is an array job (defualt `0`).
`options`	dictionary of options for submit (see table 4).

Writes submit file according to the chosen options and returns path to submit file which is used by the `condorSubmit` function to submit the job.

3.9.5 *setupRelease*

`nafSubmit.setupRelease(*args)`

`oldJIDs`	list of jobs that need to be finished before new jobs are queued.
`newJIDs`	list of jobs that wait on old jobs to be finished before being queued.

Writes a shell script which monitors the jobs with `oldJIDs`. It releases the jobs with `newJIDs` when the old ones are finished and submits the shell script with a submit file.
Returns the jobID of the job.

3.9.6 *condorSubmit*

`nafSubmit.condorSubmit(*args)`

`submitPath`	path to the submit file that is supposed to be submitted.

Submits the submit file to the NAF HTC batch system and returns its jobID.

3.9.7 *monitorJobStatus*

`nafSubmit.monitorJobStatus(**kwargs)`

`jobIDs`	list of jobs that are monitored (default `None`).

Periodically check the status of all jobs specified by `jobIDs`. This functions runs until all jobs are terminated, i.e. are neither in hold-, idle- or run- state.
If no jobID is specified, all jobs of the user are monitored.
This function cannot check, whether the job has terminated successfully or terminated with an error.

4 Configs

4.1 plots.py

4.1.1 *getDiscriminatorPlots*

`getDiscriminatorPlots(*args)`

`data`	initialized instance of `catData` (see 2.2.2).
`discrname`	name of the discriminator, usually defined in `analysisClass`.

Loads configs for different types of plots via `add_<type>` functions into `catData`. This includes:

- `categories`: category of plot
- `discrs`: discriminator of plot (sometimes depends on `finaldiscr`)
- `nhistobins`: number of bins for plot
- `minxvals`: minimal x-axis value for plot
- `maxxvals`: maximal x-axis value for plot

Afterwards, `plotPreselections` and `binlabels` are created from `categories` with the `genPlotInput` function.
Finally, a list of `plotClasses.Plot`-type instances (see **??**) is created from that data and returned.

Alternatively, the list of `plotClasses.Plot`-type instances can also be created directly in `add_<type>` functions, without using the `catData` class.

4.1.2 *evtYieldCategories*

`evtYieldCategories()`
Returns a list of categories, which is used for calculating the event yields during the `makeEventYields` step.

4.2 addVariables.py

4.2.1 *getAddVars*

`getAddVars()`
Contains hard coded list of additional variables, which is returned.

4.3 samples.py

4.3.1 *getSamples*

`getSamples(*args)`

`pltcfg`	plotconfig instance, initialized in `analysisConfig`

Loads list of `plotClasses.Sample`-type instances which contain signal and background samples. These samples are used for generating plots with `genPlots`.

4.3.2 *getControlSamples*

`getControlSamples(*args)`

`pltcfg`	plotconfig instance, initialized in `analysisConfig`

Loads list of `plotClasses.Sample`-type instances which contain control samples. These samples are used for generating plots with `genPlots`.

4.3.3 *getSystSamples*

`getSystSamples(*args)`

`pltcfg`	plotconfig instance, initialized in `analysisConfig`
`analysis`	`analysisConfig` instance
`samples`	list of samples from `getSamples` function

Loads list of `plotClasses.Sample`-type instances which contain systematic samples. To determine, which systematics are used, the list created in `getSamples` is used together with multiple lists of systematic names in plotconfig.

4.3.4 *getAllSamples*

`getAllSamples(*args)`

`pltcfg`	plotconfig instance, initialized in `analysisConfig`
`analysis`	`analysisConfig` instance
`samples`	list of samples from `getSamples` function

Concatenates the lists of `plotClasses.Sample`-type instances created with `getSamples`, `getControlSamples` and `getSystSamples`.
This list of samples is used to create the cpp program in `plotParallel`.

4.3.5 *getAllSystNames*

`getAllSystNames(*args)`

`pltcfg`	plotconfig instance, initialized in `analysisConfig`

Loads multiple lists of systematic names from plotconfig.

4.3.6 *getOtherSystNames*

`getOtherSystNames(*args)`

`pltcfg`	plotconfig instance, initialized in `analysisConfig`

Loads multiple lists of systematic names from plotconfig.

4.3.7 *getWeightSystNames*

`getWeightSystNames(*args)`

`pltcfg`	plotconfig instance, initialized in `analysisConfig`

Loads list of names of weight systematics.
This is used for example in `plotParallel` and `renameHistos`.

4.3.8 *getSystWeights*

`getSystWeights(*args)`

`pltcfg`	plotconfig instance, initialized in `analysisConfig`

Loads list of systematic weights from plotconfig.
This is used for example in `plotParallel` and `renameHistos`.