bamboo Documentation

Release 1.1.1.dev150+g588a66677.d20260309

Pieter David

Mar 09, 2026

CONTENTS

The RDataFrame class provides an efficient and flexible way to process per-event information (stored in a TTree) and e.g. aggregate it into histograms.

With the typical pattern of storing object arrays as a structure of arrays (variable-sized branches with a common prefix in the names and length), the expressions that are typically needed for a complete analysis quickly become cumbersome to write (with indices to match, repeated sub-expressions etc.). As an example, imagine the expression needed to calculate the invariant mass of the two leading muons from a NanoAOD (which stores momenta with pt, eta and phi branches): one way is to construct LorentzVector objects, sum and evaluate the invariant mass. Next imagine doing the same thing with the two highest-pt jets that have a b-tag and are not within some cone of the two leptons you already selected in another way (while keeping the code maintainable enough to allow for passing jet momenta with a systematic variation applied).

Bamboo attempts to solve this problem by automatically constructing lightweight python wrappers based on the structure of the TTree, which allows constructing such expressions with high-level code, similar to the language that is commonly used to discuss and describe them. By constructing an object representation of the expression, a few powerful operations can be used to compose complex expressions. This also allows to automate the construction of derived expressions, e.g. for shape systematic variation histograms.

Building selections, plots etc. with such expressions is analysis-specific, but the mechanics of loading data samples, processing them locally or on a batch system (and merging the output of that), combining the outputs for different samples in an overview etc. is very similar over a broad range of use cases. Therefore a common implementation of these is provided, which can be used by extending a base class (to fill histograms and make stacked plots, a class needs to be written with a method that returns a list of 'plot' objects—each essentially a combination of an x-axis variable, selection, and weight to apply to every event—and a configuration file that specifies which datasets should be processed and how they should be stacked).

CHAPTER

ONE

PRESENTATIONS

Bamboo has been presented at several workshops and working meetings:

- PyHEP2019 (16 October 2019): presentation 10.5281/zenodo.3959253
- 84th ROOT Parallelism, Performance and Programming Model Meeting (8 October 2020)
- CMS Physics Workshop on Analysis Tools and Techniques (20 November 2020) presentation [CMS internal]
- HSF WLCG Virtual Workshop (24 November 2020) presentation
- HSF Data Analysis working group: Metadata discussions (16 February 2021)
- vCHEP2021 (18 May 2021) presentation
- CHEP2024 (24 October 2024) poster

A general writeup of the framework is also available in 10.1051/epjconf/202125103052 or 2103.01889, which may be cited as

```
@article{David:2021ohq,
    author = "David, Pieter",
    title = "{Readable and efficient HEP data analysis with bamboo}",
    DOI= "10.1051/epjconf/202125103052",
    journal = {EPJ Web Conf.},
    year = 2021,
    volume = 251,
    pages = "03052",
    eprint = "2103.01889",
    archivePrefix = "arXiv",
    primaryClass = "physics.data-an",
    reportNumber = "CP3-21-05",
}
```

CHAPTER

TWO

TABLE OF CONTENTS

2.1 Installation and setup

2.1.1 Dependencies and environment

Bamboo only depends on python3 (with pip/setuptools to install PyYAML and numpy if needed) and a recent version of ROOT (6.20/00 is the minimum supported version, as it introduces some compatibility features for the new PyROOT in 6.22/00).

On user interface machines (lxplus, ingrid, or any machine with cvmfs), an easy way to get such a recent version of ROOT is through a CMSSW release that depends on it, or from the SPI LCG distribution, e.g.

```
source /cvmfs/sft.cern.ch/lcg/views/LCG_105/x86_64-centos7-gcc11-opt/setup.sh
python -m venv bamboovenv
source bamboovenv/bin/activate
```

(the second command creates a virtual environment to install python packages in, after installation it is sufficient to run two other commands, to pick up the correct base system and then the installed packages).

In case of lxplus cluster, it is recommended to use

```
source /cvmfs/sft.cern.ch/lcg/views/LCG_105/x86_64-el9-gcc11-opt/setup.sh
```

Alternatively, a conda environment (e.g. with Miniconda) can be created with

```
conda config --add channels conda-forge # if not already present
conda create -n test_bamboo root pyyaml numpy cmake boost
conda activate test_bamboo
```

and bamboo installed directly there with pip, or in a virtual environment inside the conda environment (make sure to pass `--system-site-packages` to `venv` then); conda-build recipes are in the plans.

A docker image (based on repo2docker, configuration) with an up-to-date version of bamboo and plotIt is also available. It is compatible with binder, which can be used to run some examples without installing anything locally.

Some features bring in additional dependencies. Bamboo should detect if these are relied on and missing, and print a clear error message in that case. Currently, they include:

- the dasgoclient executable (and a valid grid proxy) for retrieving the list of files in samples specified with `db: das:/X/Y/Z`. Due to some interference with the setup script above, the best is to run the cms environment scripts first, and also run `voms-proxy-init` then (this can alternatively also be done from a different shell on the same machine)
- the slurm command-line tools, and CP3SlurmUtils, which can be installed using *pip* (or loaded with `module load slurm/slurm_utils` on the UCLouvain ingrid ui machines)

- machine learning libraries (libtorch, Tensorflow-C, lwtnn): see *this section* for more information
- writing out tables in LaTeX format from cutflow reports relies needs pyplotit (see below)
- Dask or pySpark for running distributed RDataFrame (see below)

2.1.2 Installation

Bamboo can (and should, in most cases) be installed in a virtual environment or conda environment (see above) with pip:

```
pip install bamboo-hep
```

Since Bamboo is still in heavy development, you may want to fetch the latest (unreleased) version using one of:

```
pip install git+https://gitlab.cern.ch/cp3-cms/bamboo.git
pip install git+ssh://git@gitlab.cern.ch:7999/cp3-cms/bamboo.git
```

It may even be useful to install from a local clone, such that you can use it to test and propose changes, using

```
git clone -o upstream https://gitlab.cern.ch/cp3-cms/bamboo.git /path/to/your/bambooclone
pip install /path/to/your/bambooclone ## e.g. ./bamboo (not bamboo - another package␣
↪with that name exists)
```

such that you can update later on with (inside `/path/to/your/bambooclone`)

```
git pull upstream master
pip install --upgrade .
```

It is also possible to install bamboo in editable mode for development; to avoid problems, this should be done in a separate virtual environment:

```
python -m venv devvenv ## deactivate first, or use a fresh shell
source devvenv/bin/activate ## deactivate first, or use a fresh shell
export SETUPTOOLS_ENABLE_FEATURES=legacy-editable
pip install -e ./bamboo
```

Note that this will store cached build outputs in the `_skbuild` directory. `python setup.py clean --all` can be used to clean this up (otherwise they will prevent updating the non-editable install). The additional environment variable is a workaround for a bug in scikit-build, see this issue.

The documentation can be built locally with `python setup.py build_sphinx`, and for running all (or some) tests the easiest is to call `pytest` directly, with the `bamboo/tests` directory to run all tests, or with a specific file to check only the tests defined there.

Note

bamboo is a shared package, so everything that is specific to a single analysis (or a few related analyses) is best stored elsewhere (e.g. in `bamboodev/myanalysis` in the example below); otherwise you will need to be very careful when updating to a newer version.

The `bambooRun` command can pick up code in different ways, so it is possible to start from a single python file, and move to a pip-installed analysis package later on when code needs to be shared between modules.

For combining the different histograms in stacks and producing pdf or png files, which is used in many analyses, the plotIt tool is used. It can be installed with cmake, e.g.

```
git clone -o upstream https://github.com/cp3-llbb/plotIt.git /path/to/your/plotitclone
mkdir build-plotit
cmake -DCMAKE_INSTALL_PREFIX=$VIRTUAL_ENV -S /path/to/your/plotitclone -B build-plotit
cmake --build build-plotit -t install -j 4
```

where `-DCMAKE_INSTALL_PREFIX=$VIRTUAL_ENV` ensures that the `plotIt` executable will be installed directly in the `bin` directory of the virtualenv (if not using a virtualenv, its path can be passed to `bambooRun` with the `--plotIt` command-line option).

plotIt is very efficient at what it does, but not so easy to adapt to producing efficiently plots, overlays of differently defined distributions etc. Therefore, a Python implementation of its main functionality was started in the pyplotit package, which can be installed with

```
pip install git+https://gitlab.cern.ch/cp3-cms/pyplotit.git
```

or editable from a local clone:

```
git clone -o upstream https://gitlab.cern.ch/cp3-cms/pyplotit.git
pip install -e pyplotit
```

pyplotit parses plotIt YAML files and implements the same grouping and stack-building logic; an easy way to get started with it is through the `iPlotIt` script, which parses a plotIt configuration file and launches an IPython shell. Currently this is used in bamboo for producing yields tables from cutflow reports. It is also very useful for writing custom postprocess functions, see *this recipe* for an example.

To use scalefactors and weights in the new CMS JSON format, the correctionlib package should be installed with

```
pip install --no-binary=correctionlib correctionlib
```

The calculator modules for *jet and MET corrections and systematic variations* were moved to a separate repository and package, such that they can also be used from other frameworks. The repository can be found at cp3-cms/CMSJMECalculators, and installed with

```
pip install git+https://gitlab.cern.ch/cp3-cms/CMSJMECalculators.git
```

For the impatient: recipes for installing and updating

Putting the above commands together, the following should give you a virtual environment with bamboo, and a clone of bamboo and plotIt in case you need to modify them, all under `bamboodev`:

Fresh install

```
mkdir bamboodev
cd bamboodev
# make a virtualenv
source /cvmfs/sft.cern.ch/lcg/views/LCG_105/x86_64-centos7-gcc11-opt/setup.sh
python -m venv bamboovenv
source bamboovenv/bin/activate
# clone and install bamboo
git clone -o upstream https://gitlab.cern.ch/cp3-cms/bamboo.git
pip install ./bamboo
# clone and install plotIt
git clone -o upstream https://github.com/cp3-llbb/plotIt.git
mkdir build-plotit
```

(continues on next page)

(continued from previous page)

```
cd build-plotit
cmake -DCMAKE_INSTALL_PREFIX=$VIRTUAL_ENV ../plotIt
make -j2 install
cd -
```

Environment setup

Once bamboo and plotIt have been installed as above, only the following two commands are needed to set up the environment in a new shell:

```
source /cvmfs/sft.cern.ch/lcg/views/LCG_105/x86_64-centos7-gcc11-opt/setup.sh
source bamboodev/bamboovenv/bin/activate
```

Update bamboo

Assuming the environment is set up as above; this can also be used to test a pull request or local modifications to the bamboo source code

```
cd bamboodev/bamboo
git checkout master
git pull upstream master
pip install --upgrade .
```

Update plotIt

Assuming the environment is set up as above; this can also be used to test a pull request or local modifications to the plotIt source code. If a plotIt build directory already exists it should have been created with the same environment, otherwise the safest solution is to remove it.

```
cd bamboodev
mkdir build-plotIt
cd build-plotit
cmake -DCMAKE_INSTALL_PREFIX=$VIRTUAL_ENV ../plotIt
make -j2 install
cd -
```

Move to a new LCG release or install an independent version

Different virtual environments can exist alongside each other, as long as for each the corresponding base LCG distribution is setup in a fresh shell. This allows to have e.g. one stable version used for analysis, and another one to test experimental changes, or check a new LCG release, without touching a known working version.

```
cd bamboodev
source /cvmfs/sft.cern.ch/lcg/views/LCG_105/x86_64-centos7-gcc11-opt/setup.sh
python -m venv bamboovenv_X
source bamboovenv_X/bin/activate
pip install ./bamboo
# install plotIt (as in "Update plotIt" above)
mkdir build-plotit
cd build-plotit
cmake -DCMAKE_INSTALL_PREFIX=$VIRTUAL_ENV ../plotIt
```

(continues on next page)

(continued from previous page)

```
make -j2 install
cd -
```

2.1.3 Test your setup

Now you can run a few simple tests on a CMS NanoAOD to see if the installation was successful. A minimal example is run by the following command:

```
bambooRun -m /path/to/your/bambooclone/examples/nanozmumu.py:NanoZMuMu /path/to/your/
↪bambooclone/examples/test1.yml -o test1
```

which will run over a single sample of ten events and fill some histograms (in fact, only one event passes the selection, so they will not look very interesting). If you have a NanoAOD file with muon triggers around, you can put its path instead of the test file in the yml file and rerun to get a nicer plot (xrootd also works, but only for this kind of tests—in any practical case the performance benefit of having the files locally is worth the cost of replicating them).

2.1.4 Getting started

The test command above shows how bamboo is typically run: using the *bambooRun* command, with a python module that specifies what to run, and an *analysis YAML file* that specifies which samples to process, and how to combine them in plots (there are several options to run a small test, or submit jobs to the batch system when processing a lot of samples).

A more realistic analysis YAML configuration file is bamboo/examples/analysis_zmm.yml, which runs on a significant fraction of the 2016 and 2017 `DoubleMuon` data and the corresponding Drell-Yan simulated samples. Since the samples are specified by their DAS path in this case, the `dasgoclient` executable and a valid grid proxy are needed for resolving those to files, and a *configuration file* that describes the local computing environment (i.e. the root path of the local CMS grid storage, or the name of the redirector in case of using xrootd); examples are included for the UCLouvain-CP3 and CERN (lxplus/lxbatch) cases.

The corresponding python module shows the typical structure of ever tighter event selections that derive from the base selection, which accepts all the events in the input, and plots that are defined based on these selection, and returned in a list from the main method (this corresponds to the pdf or png files that will be produced).

The module deals with a decorated version of the tree, which can also be inspected from an IPython shell by using the `-i` option to `bambooRun`, e.g.

```
bambooRun -i -m /path/to/your/bambooclone/examples/nanozmumu.py:NanoZMuMu /path/to/your/
↪bambooclone/examples/test1.yml
```

together with the helper methods defined on *this page*, this allows to define a wide variety of selection requirements and variables.

The *user guide* contains a much more detailed description of the different files and how they are used, and the *analysis recipes page* provides more information about the bundled helper methods for common tasks. The *API reference* describes all available user-facing methods and classes. If the builtin functionality is not sufficient, some hints on extending or modifying bamboo can be found in the *advanced topics* and the *hacking guide*.

2.1.5 Machine learning packages

In order to evaluate machine learning classifiers, bamboo needs to find the necessary C(++) libraries, both when the extension libraries are compiled and at runtime (so they need to be installed before (re)installing bamboo). libtorch is searched for in the `torch` package with `importlib`, which unfortunately does not always work due to `pip` build isolation. This can be bypassed by passing `--no-build-isolation` when installing, or by installing `bamboo-hep[torch]`, which will install it as a dependency (it is quite big, so if the former method works it should

be preferred). The `--no-build-isolation` option is a workaround: when passing CMake options to pip install (see scikit-build#479) will be possible, that will be a better solution. The minimum version required for libtorch is 1.5 (due to changes in the C++ API), which is available from LCG_99 on (contains libtorch 1.7.0). Tensorflow-C and lwtnn will be searched for (by cmake and the dynamic library loader) in the default locations, supplemented with the currently active virtual environment, if any (scripts to install them there directly are included in the bamboo source code respository, as `ext/install_tensorflow-c.sh` and `ext/install_lwtnn.sh`). ONNX Runtime should be part of recent LCG distribution. If not, it will be searched for in the standard locations. It can be added to the virtual environment by following the instruction to build from source, with the additional option `--cmake_extra_defines=CMAKE_INSTALL_PREFIX=$VIRTUAL_ENV`, after which `make install` from its `build/Linux/<config>` will install it correctly (replacing `<config>` by the CMake build type, e.g. Release or RelWithDebInfo).

Note

Installing a newer version of libtorch in a virtualenv if it is also available through the `PYTHONPATH` (e.g. in the LCG distribution) generally does not work, since virtualenv uses `PYTHONHOME`, which has lower precedence. For the pure C(++) libraries Tensorflow-C and lwtnn this could be made to work, but currently the virtual environment is only explicitly searched if they are not found otherwise. Therefore it is recommended to stick with the version provided by the LCG distribution, or set up an isolated environment with conda—see the issues #68 (for now) and #65 for more information. When a stable solution is found it will be added here.

Warning

the libtorch and Tensorflow-C builds in LCG_98python3 contain AVX2 instructions (so one of these CPU generations). See issue #68 for more a more detailed discussion, and a possible workaround.

2.1.6 Distributed RDataFrame

Through distributed ROOT::RDataFrame, bamboo can distribute the computations on a cluster managed by Dask or pySpark. While Dask, using Dask-jobqueue, can work on any existing cluster managed by SLURM or HTCondor, Spark requires a Spark scheduler to be running at your computing centre.

To install the required dependencies, run either one of:

```
pip install bamboo-hep[dask]
pip install bamboo-hep[spark]
```

2.1.7 EasyBuild-based installation at CP3

On the ingrid/manneback cluster at UCLouvain-CP3, and other environments that use EasyBuild, it is also possible to install bamboo based on the dependencies that are provided through this mechanism (potentially with some of them built as user modules). The LCG source script in the instructions above should then be replaced by e.g.

```
module load ROOT/6.22.08-foss-2019b-Python-3.7.4 CMake/3.15.3-GCCcore-8.3.0 \
   Boost/1.71.0-gompi-2019b matplotlib/3.1.1-foss-2019b-Python-3.7.4 \
   PyYAML/5.1.2-GCCcore-8.3.0 TensorFlow/2.1.0-foss-2019b-Python-3.7.4
```

2.2 User guide

This section contains some more information on doing your analysis with bamboo. It assumes you have successfully installed it following the instructions in the *previous section*.

The first thing to make sure is that bamboo can work with your trees. With CMS NanoAOD, many analysis use the same (or a very similar) tree format, which is why a set of decorators for is included in *bamboo.treedecorators.decorateNanoAOD()*; to make stacked histogram plots from them it is sufficient to make your analysis module inherit from *bamboo.analysismodules.NanoAODHistoModule* (which calls this method from its `prepareTrees()` method). Other types of trees can be included in a similar way, but a bit of development is needed to provided a more convenient way to do so (help welcome).

2.2.1 Running bambooRun

The `bambooRun` executable script can be used to run over some samples and derive other samples or histograms from them. It needs at least two arguments: a *python module*, which will tell it what to do for each event, and a *configuration file* with a list of samples to process, plot settings etc.

Typically, `bambooRun` would be invoked with

```
bambooRun -m module-specification config-file-path -o my-output
```

where `module-specification` is of the format `modulename:classname`, and `modulename` can be either a file path like `somedir/mymodule.py` or an importable module name like `myanalysispackage.mymodule`. This will construct an instance of the specified module, passing it any command-line arguments that are not used directly by `bambooRun`, and run it.

The default base module (*bamboo.analysismodules.AnalysisModule*, see *below*) provides a number of convenient command-line options (and individual modules can add more by implementing the *addArgs()* method).

- the `-h` (`--help`) switch prints the complete list of supported options and arguments, including those defined by the module if used with `bambooRun -h -m mymodule`
- the `-o` (`--output`) option can be used to specify the base directory for output files
- the `-v` (`--verbose`) switch will produce more output messages, and also print the full C++ code definitions that are passed to RDataFrame (which is very useful for debugging)
- the `-i` (`--interactive`) switch will only load one file and launch an IPython terminal, where you can have a look at its structure and test expressions
- the `--maxFiles` option can be used to specify a maximum number of files to process for each sample, e.g. `--maxFiles=1` to check that the module runs correctly in all cases before submitting to a batch system
- the `--eras` option specifies which of the eras from the configuration file to consider, and which type of plots to make. The format is `[mode][:][era1,era2,...]`, where `mode` is one of `split` (plots for each of the eras separately), `combined` (only plots for all eras combined) or `all` (both of these, this is the default).
- the `--distributed` mode specifies how the processing should run (locally, on a cluster, ...), see *below*.
- the `-t` (`--threads`) option can be used to run in multi-threaded mode, for both the local or batch mode

Computing environment configuration file

For some features such as automatically converting logical filenames from DAS to physical filenames at your local T2 storage (or falling back to xrootd), submitting to a batch cluster etc., some information about the computing resources and environment is needed. In order to avoid proliferating the command-line interface of `bambooRun`, these pieces of information are bundled in a file that can be passed in one go through the `--envConfig` option. If not specified, Bamboo will try to read `bamboo.ini` in the current directory, and then `$XDG_CONFIG_HOME/bambooorc` (which typically resolves to `~/.config/bambooorc`). Since these settings are not expected to change often or much, it is advised to

copy the closest example (e.g. `examples/ingrid.ini` or `examples/lxplus.ini`) to `~/.config/bamboorc` and edit if necessary.

2.2.2 Analysis YAML file format

The analysis configuration file should be in the YAML format. This was chosen because it can easily be parsed while also being very readable (see the YAML Wikipedia page for some examples and context) - it essentially becomes a nested dictionary, which can also contain lists.

Three top-level keys are currently required: `tree` with the name of the TTree inside the file (e.g. `tree: Events` for NanoAOD), `samples` with a list of samples to consider, and `eras`, with a list of data-taking periods and their integrated luminosity. For stacked histogram plots, a `plotIt` section should also be specified (the *bamboo.analysisutils.runPlotIt()* method will insert the `files` and `plots` sections and run plotIt with the resulting configuration; depending on the `--eras` option passed, per-era or combined plots will be produced, or both, which is the default). Each entry in the `plots` section will contain the combination of the settings explicitly passed to *make1D()*, those present in `plotDefaults`, and those specified under the `plotdefaults` block in the `plotIt` section of the analysis configuration file (in this order of precedence); if the values are callable, the result of calling them on the *Plot* is used (which may be useful to adjust e.g. the axis range to the binning; by default the binning range is used as x-axis range). The full list of plotIt configuration options can be found on this page.

Each entry in the `samples` dictionary (the keys are the names of the samples) is another dictionary. The files to be processed can be specified directly as a list under `files` (with paths relative to the location of the config file, which is useful for testing), or absolute paths/urls (e.g. xrootd). If `files` is a string, it is taken as a file with a list of such paths/urls. For actual analyses, however, samples will be retrieved from a database, e.g. DAS or SAMADhi (support for the latter still needs to be implemented). In that case, the database path or query can be specified under `db`, e.g. `db: das:/SingleMuon/Run2016E-Nano14Dec2018-v1/NANOAOD`. The results of these queries can be cached locally by adding a `dbcache` top-level configuration entry, with a directory where the text files can be stored. For each sample a file `<sample_name>.txt` will be created, with a comment that contains the `db` value used to create it, such that changes can automatically be detected and the query redone, and the list of files. To force rerunning some or all queries, the corresponding files or the whole cache directory can be moved or deleted.

Which NanoAOD samples to use?

When analysing CMS NanoAOD samples there are two options: postprocessing the centrally produced NanoAOD samples with CRAB to add corrections and systematic variations as new branches, or calculating these on demand (see the corresponding *recipes* for more details). Which solution is optimal depends on the case (it is a trade-off between file size and the time spent on calculating the variations), but the latter is the easiest to get started with: just create some Rucio rules to make the samples available locally: the transfers are usually very fast—much faster than processing all the samples with CRAB. Tip: with Rucio containers you can group datasets and manage them together. Depending on the site policies you may need to ask for quota or approval of the rules.

Tip

Samples in DAS and SAMADhi rarely change, and reading a local file is almost always faster than doing queries (and does not require a grid proxy etc.), so especially when using many samples from these databases it is recommended to cache the file lists resulting from these results, by specifying a path under `dbcache` at the top level of the configuration file (see below for an example).

For data, it is usually necessary to specify a json file to filter the good luminosity blocks (and a run range to consider from it, for efficiency). If an url is specified for the json file, the file will be downloaded automatically (and added to the input sandbox for the worker tasks, if needed).

For the formatting of the stack plots, each sample needs to be in a group (e.g. 'data' for data etc.), which will be taken

together as one contribution. The `era` key specifies which era (one of those specified in the `eras` section, see above) the sample corresponds to, and which luminosity value should be used for the normalisation.

For the normalization of simulated samples in the stacks, the number of generated events and cross-section are also needed. The latter should be specified as `cross-section` with the sample (in the same units as the `luminosity` for the corresponding `era`), the former can be computed from the input files. For this, the *bamboo.analysismodules.HistogramsModule* base class will call the `mergeCounters` method when processing the samples, and the `readCounters` method to read the values from the results file - for NanoAOD the former merges the `Runs` trees and saves the results, while the latter performs the sum of the branch with the name specified under `generated-events`.

For large samples, a `split` property can be specified, such that the input files are spread out over different batch jobs. A positive number is taken as the number of jobs to divide the inputs over, while a negative number gives the number of files per job (e.g. `split: 3` An `era` key is also foreseen (to make 2016/2017/2018/combined plots) - but it is currently ignored. will create three jobs to process the sample, while `split: -3` will result in jobs that process three files each).

All together a typical analysis YAML file would look like the following (but with many more sample blocks, and typically a few era blocks; the `plotIt` section is left out for brevity).

```
tree: Events
eras:
  '2016':
    luminosity: 12.34
dbcache: dascache
samples:
  SingleMuon_2016E:
    db: das:/SingleMuon/Run2016E-Nano14Dec2018-v1/NANOAOD
    run_range: [276831, 277420]
    certified_lumi_file: https://cms-service-dqm.web.cern.ch/cms-service-dqm/CAF/
↪certification/Collisions16/13TeV/ReReco/Final/Cert_271036-284044_13TeV_23Sep2016ReReco_
↪Collisions16_JSON.txt
    era: 2016
    group: data

  DY_high_2017:
    db: das:/DYJetsToLL_M-50_TuneCP5_13TeV-amcatnloFXFX-pythia8/RunIIFall17NanoAODv4-
↪PU2017_12Apr2018_Nano14Dec2018_new_pmx_102X_mc2017_realistic_v6_ext1-v1/NANOAODSIM
    era: 2017
    group: DY
    cross-section: 5765.4
    generated-events: genEventSumw
    split: 3
```

Tip

It is possible to insert the content of a configuration file into another, e.g. to separate or reuse the plot- and samples-related setings: simply use the syntax `!include file.yml` in the exact place where you would like to insert the content of `file.yml`.

2.2.3 Analysis module

For an analysis module to be run with `bambooRun`, it in principle only needs a constructor that takes an argument with command-line arguments, and a `run` method. *bamboo.analysismodules* provides a more interesting base class *AnalysisModule* that provides a lot of common functionality (most notably: parsing the analysis configuration, running sequentially or distributed (and also as worker task in the latter case), and provides *addArgs()*, *initialize()*, `processTrees()`, *postProcess()*, and `interact()`, interface member methods that should be further specified by subclasses (see the *reference documentation* for more details).

HistogramsModule does this for the stacked histogram plots, composing `processTrees()` from *prepareTree()* and *definePlots()*, while taking the JSON lumi block mask and counter merging into account. It also calls the `plotIt` executable from *postProcess()* (with the plots list and analysis configuration file, it has all required information for that). *NanoAODHistoModule* supplements this with the decorations and counter merging and reading for NanoAOD, such that all the final module needs to do is defining plots and selections, as in the example `examples.nanozmumu`. This layered structure is used such that code can be maximally reused for other types of trees.

For the code inside the module, the example is also very instructive:

```
def definePlots(self, t, noSel, sample=None, sampleCfg=None):
    from bamboo.plots import Plot, EquidistantBinning
    from bamboo import treefunctions as op

    plots = []

    twoMuSel = noSel.refine("twoMuons", cut=[ op.rng_len(t.Muon) > 1 ])
    plots.append(Plot.make1D("dimu_M", op.invariant_mass(t.Muon[0].p4, t.Muon[1].p4),␣
↪twoMuSel,
            EquidistantBinning(100, 20., 120.), title="Dimuon invariant mass", plotopts={
↪"show-overflow":False}))

    return plots
```

The key classes are defined in *bamboo.plots*: *Plot* and *Selection* (see the *reference documentation* for details). The latter represents a consistent set of selection requirements (cuts) and weight factors (e.g. to apply corrections). Selections are defined by refining a "root selection" with additional cuts and weights, and each should have a unique name (an exception is raised at construction otherwise). The root selection allows to do some customisation upfront, e.g. the applying the JSON luminosity block mask for data. A plot object refers to a selection, and specifies which variable(s) to plot, with which binning(s), labels, options etc. (the `plotOpts` dictionary is copied directly into the plot section of the plotIt configuration file).

Histograms corresponding to systematic variations (of scalefactors, collections etc.—see below) are by default generated automatically alongside the nominal one. This can however easily be disabled at the level of a *Selection* (and, consequently, all *Selection* instances deriving from it, and all *Plot* instances using it) or a single plot, by passing `autoSyst=False` to the *refine()* or *make1D()* (or related) method, respectively, when constructing them; so setting `noSel.autoSyst = False` right after retrieving the decorated tree and root selection would turn disable all automatic systematic variations.

2.2.4 Specifying cuts, weight, and variables: expressions

The first argument to the *definePlots()* method is the "decorated" tree—a proxy object from which expressions can be derived. Sticking with the NanoAOD example, `t.Muon` is another proxy object for the muon collection (similarly for the other objects), `t.Muon[0]` retrieves the leading-pt muon proxy, and `t.Muon[0].p4` its momentum fourvector. The proxies are designed to behave as much as possible as the value types they correspond to (you can get an item from a list, an attribute from an object, you can also work with numerical values, e.g. `t.Muon[0].p4.Px()+t.Muon[1].p4.Px()`) but for some more complex operations, specific functions are needed. These are as much as possible defined in the *bamboo.treefunctions* module, see *Building expressions* for an overview of all the available methods.

Ideally, the decorated tree and the *bamboo.treefunctions* module are all you ever need to import and know about the decorations. Therefore the best way to proceed now is get a decorated tree inside an IPython shell and play around. For *bamboo.analysismodules.HistogramsModule* this can always be done by passing the `--interactive` flag, with either one of (depending on if you copied the NanoAOD test file above)

```
bambooRun -m bamboo/examples/nanozmumu.py:NanoZMuMu --interactive --distributed=worker␣
↪bamboo/tests/data/DY_M50_2016.root
bambooRun -m bamboo/examples/nanozmumu.py:NanoZMuMu --interactive bamboo/examples/test_
↪nanozmm.yml [ --envConfig=bamboo/examples/ingrid.ini ] -o int1
```

The decorated tree is in the `tree` variable (the original `TChain` is in `tup`) and the *bamboo.treefunctions* module is there as `op` (the `c_...` methods construct a constant, whereas the `rng_...` methods work on a collection and return a single value, whereas the *select()* method returns a reduced collection (internally, only a list of indices to the passing objects is created, and the result is a proxy that uses this list). Some of the `rng_...` methods are extremely powerful, e.g. *rng_find()* and *rng_max_element_by()*.

Tip

In addition to the branches read from the input tree, all elements of collections have an `idx` attribute which contains their index in the *original* collection (`base`), also in case they are obtained from a subset (with *select()* or a slice), differently ordered version (with *sort()*), or systematic variation (e.g. for *jets*) of the collection. This can be especially useful to ensure that two objects are (not) identical, or when directly comparing systematic variations. Similarly, all collections, selections, slices etc. have an `idxs` attribute, with the list of indices in the original collection.

This can also be exploited to precalculate an expensive quantity for a collection of objects (with *map()*), or even to evaluate a quantity for items passing different selections (e.g. the passing and failing selections), something like `fun(passing.base[op.switch(op.rng_len(passing) > 0, passing[0].idx, failing[0].idx)])`.

The proxy classes are generated on the fly with all branches as attributes, so tab-completion can be used to have a look at what's there:

```
In [1]: tree.<TAB>
  tree.CaloMET                          tree.SoftActivityJetHT10
  tree.Electron                         tree.SoftActivityJetHT2
  tree.FatJet                           tree.SoftActivityJetHT5
  tree.Flag                             tree.SoftActivityJetNjets10
  tree.HLT                              tree.SoftActivityJetNjets2
  tree.HLTriggerFinalPath               tree.SoftActivityJetNjets5
  tree.HLTriggerFirstPath               tree.SubJet
  tree.Jet                              tree.Tau
  tree.L1Reco_step                      tree.TkMET
  tree.MET                              tree.TrigObj
  tree.Muon                             tree.deps
  tree.OtherPV                          tree.event
  tree.PV                               tree.fixedGridRhoFastjetAll
  tree.Photon                           tree.fixedGridRhoFastjetCentralCalo
  tree.PuppiMET                         tree.fixedGridRhoFastjetCentralNeutral
  tree.RawMET                           tree.luminosityBlock
  tree.SV                               tree.op
  tree.SoftActivityJet                  tree.run
  tree.SoftActivityJetHT
```

(continues on next page)

(continued from previous page)

```
In [1]: anElectron = tree.Electron[0]

In [2]: anElectron.<TAB>
   anElectron.charge                   anElectron.eInvMinusPInv            anElectron.
↪mvaSpring16HZZ_WPL
   anElectron.cleanmask                anElectron.energyErr                anElectron.
↪mvaTTH
   anElectron.convVeto                 anElectron.eta                      anElectron.op
   anElectron.cutBased                 anElectron.hoe                      anElectron.p4
   anElectron.cutBased_HEEP            anElectron.ip3d                     anElectron.
↪pdgId
   anElectron.cutBased_HLTPreSel       anElectron.isPFcand                 anElectron.
↪pfRelIso03_all
   anElectron.deltaEtaSC               anElectron.jet                      anElectron.
↪pfRelIso03_chg
   anElectron.dr03EcalRecHitSumEt      anElectron.lostHits                 anElectron.phi
   anElectron.dr03HcalDepth1TowerSumEt anElectron.mass                     anElectron.
↪photon
   anElectron.dr03TkSumPt              anElectron.miniPFRelIso_all         anElectron.pt
   anElectron.dxy                      anElectron.miniPFRelIso_chg         anElectron.r9
   anElectron.dxyErr                   anElectron.mvaSpring16GP            anElectron.
↪sieie
   anElectron.dz                       anElectron.mvaSpring16GP_WP80       anElectron.
↪sip3d
   anElectron.dzErr                    anElectron.mvaSpring16GP_WP90       anElectron.
↪tightCharge
   anElectron.eCorr                    anElectron.mvaSpring16HZZ           anElectron.
↪vidNestedWPBitmap
```

For NanoAOD the content of the branches is documented here. More information about the central NanoAOD production campaigns is provided here.

In addition to the branches present in the NanoAOD, the following attributes are added for convenience:

- `p4` if `pt`, `eta`, `phi`, and `mass` attributes are defined. `phi` and `mass` are optional, such that this also works for `TrigObj` and various kinds of MET.
- `idx` for elements of containers
- for `GenPart`: `parent`, which refers to the parent or mother particle (the presence can be tested by comparing its `idx` to `-1`), and `ancestors`, the range of all ancestors—this does check the validity, so it may be empty.
- for electrons and photons: `etaSC`, which refers to the SuperCluster (SC) pseudorapidity eta. The pseudorapity eta and SC pseudorapity eta are not exactly same. The SC eta is calculated with respect to the detector center (0,0,0), while the pseudorapidity eta is calculated with respect to the primary vertex.

2.2.5 Processing modes

The usual mode of operation is to 1. parse the analysis configuration file, 2. execute some code for every entry in each of the samples, and then 3. perform some actions on the aggregated results (e.g. produce nice-looking plots from the raw histograms). Since the second step is by far the most time-consuming, but can be performed indepently for different samples (and even entries), it is modeled as a list of tasks (which may be run in parallel), after which a postprocessing step takes the results and combines them. The latter step can also be run separately, using the results of previously run tasks, assuming these did not change.

More concretely, for e.g. histogram stack plots, the tasks produce histograms while the postprocessing step runs plotIt,

so with the `--onlypost` option the normalization, colors, labels etc. can be changed without reprocessing the samples (some tips on additional postprocessing can be found in *this recipe*).

The task processing itself can be run in three different modes, depending on the option passed to `--distributed`:

By default (`--distributed=sequential`), every sample is processed sequentially. This is useful for short tests or for small tasks that don't take very long. The execution can be made quicker by running in multithreaded mode using `-t`.

Alternatively, it is possible to process multiple tasks in parallel using `--distributed=parallel`. This will first create the processing configuration for every sample, before starting the actual processing. Again, the executing of these tasks can be made to run on multiple threads using `-t`.

In practice, you will most likely want to submit independent tasks to a computing cluster using a batch scheduler (currently HTCondor and Slurm are supported). Bamboo will submit the jobs, monitor them, and combine the results when they are finished. More information about monitoring and recovering failed batch jobs is given in *the corresponding recipe*. By default one batch job is submitted for each input sample, unless there is a `split` entry different from one for the sample, see *below* for the precise meaning.

Finally, it is possible to offload the computations to a computing cluster using modern distributed computing tools such as Dask or Spark. This means that Dask/Spark will take care of splitting the input data, launching and monitoring jobs, and retrieving the results. This mode can be activated using the `--distrdf-be` argument, and can work both with `--distributed=sequential` (in which case every sample will be processed sequentially by the whole cluster) or with `--distributed=parallel` (in which case the processing of all the input samples will happen in parallel). More information about how to configure bamboo for Dask/Spark can be found *here*.

2.2.6 Examples

Some more complete examples, based on open data RDataFrame tutorials, are available in this repository (they can be run on binder without installing anything locally).

The *recipes page* has a collection of common analysis tasks, with a recommended implementation, and pointers to the relevant helper functions; it may good to skim through to get an idea of what a typical analysis implementation will look like.

2.3 Building expressions

In order to efficiently process the input files, Bamboo builds up an object representation of the expressions (cuts, weights, plotted variables) needed to fill the histograms, and dynamically generates C++ code that is passed to RDataFrame. The expression trees are built up through proxy classes, which mimic the final type (there are e.g. integer and floating-point number proxy classes that overload the basic mathematical operators), and generate a new proxy when called. As an example: `t.Muon[0].charge` gives an integer proxy to the operation corresponding to `Muon_charge[0]`; when the addition operator is called in `t.Muon[0].charge+t.Muon[1].charge`, an integer proxy to (the object representation of) `Muon_charge[0]+Muon_charge[1]` is returned.

The proxy classes try to behave as much as possible as the objects they represent, so in most cases they can be used as if they really were a number, boolean, momentum fourvector… or a muon, electron, jet etc.—simple 'struct' types for those are generated when decorating the tree, based on the branches that are found. Some operations, however, cannot easily be implemented in this way, for instance mathematical functions and operations on containers. Therefore, the *`bamboo.treefunctions`* module provides a set of additional helper methods (such that the user does not need to know about the implementation details in the `bamboo.treeoperations` and `bamboo.treeproxies` modules). In order to keep the analysis code compact, it is recommended to import it with

```
from bamboo import treefunctions as op
```

inside every analysis module. The available functions are listed below.

2.3.1 List of functions

bamboo.treefunctions.**typeOf**(*arg*)

Get the inferred C++ type of a bamboo expression (proxy or TupleOp)

bamboo.treefunctions.**c_bool**(*arg*)

Construct a boolean constant

bamboo.treefunctions.**c_int**(*num*, *typeName='int'*, *cast=None*)

Construct an integer number constant (static_cast inserted automatically if not 'int',
a boolean can be passed to 'cast' to force or disable this)

bamboo.treefunctions.**c_float**(*num*, *typeName='double'*, *cast=None*)

Construct a floating-point number constant (static_cast inserted automatically if not 'double',
a boolean can be passed to 'cast' to force or disable this)

bamboo.treefunctions.**NOT**(*sth*)

Logical NOT

bamboo.treefunctions.**AND**(**args*)

Logical AND

bamboo.treefunctions.**OR**(**args*)

Logical OR

bamboo.treefunctions.**switch**(*test*, *trueBranch*, *falseBranch*, *checkTypes=True*)

Pick one or another value, based on a third one (ternary operator in C++)

Example

```
>>> op.switch(runOnMC, mySF, 1.) ## incomplete pseudocode
```

bamboo.treefunctions.**multiSwitch**(**args*)

Construct arbitrary-length switch (if-elif-elif-...-else sequence)

Example

```
>>> op.multiSwitch((lepton.pt > 30, 4.), (lepton.pt > 15 && op.abs(lepton.eta) < 2.
↪1, 5.), 3.)
```

is equivalent to:

```
>>> if lepton.pt > 30:
>>>     return 5.
>>> elif lepton.pt > 15 and abs(lepton.eta) < 2.1:
>>>     return 4.
>>> else:
>>>     return 3.
```

bamboo.treefunctions.**extMethod**(*name*, *returnType=None*)

Retrieve a (non-member) C(++) method

Parameters

- **name** – name of the method
- **returnType** – return type (otherwise deduced by introspection)

Returns
a method proxy, that can be called and returns a value decorated as the return type of the method

Example

```
>>> phi_0_2pi = op.extMethod("ROOT::Math::VectorUtil::Phi_0_2pi")
>>> dphi_2pi = phi_0_2pi(a.Phi()-b.Phi())
```

bamboo.treefunctions.**extVar**(*typeName*, *name*)

Use a variable or object defined outside bamboo

Parameters

- **typeName** – C++ type name
- **name** – name in the current scope

Returns
a proxy to the variable or object

bamboo.treefunctions.**construct**(*typeName*, *args*)

Construct an object

Parameters

- **typeName** – C++ type name
- **args** – constructor arguments

Returns
a proxy to the constructed object

bamboo.treefunctions.**static_cast**(*typeName*, *arg*)

Compile-time type conversion

mostly for internal use, prefer higher-level functions where possible

Parameters

- **typeName** – C++ type to cast to
- **arg** – value to cast

Returns
a proxy to the casted value

bamboo.treefunctions.**initList**(*typeName*, *valueType*, *elements*)

Construct a C++ initializer list

mostly for internal use, prefer higher-level functions where possible

Parameters

- **typeName** – C++ type to use for the proxy (note that initializer lists do not have a type)
- **valueType** – C++ type of the elements in the list
- **elements** – list elements

Returns
a proxy to the list

bamboo.treefunctions.**array**(*valueType*, **elements*)

Helper to make a constructing a std::array easier

Parameters

- **valueType** – array element C++ type
- **elements** – array elements

Returns

a proxy to the array

bamboo.treefunctions.**define**(*typeName*, *definition*, *nameHint=None*)

Define a variable as a symbol with the interpreter

Parameters

- **typeName** – result type name
- **definition** – C++ definition string, with <<name>> instead of the variable name (which will be replaced by nameHint or a unique name)
- **nameHint** – (optional) name for the variable

Caution

nameHint (if given) should be unique (for the sample), otherwise an exception will be thrown

bamboo.treefunctions.**defineOnFirstUse**(*sth*)

Construct an expression that will be precalculated (with an RDataFrame::Define node) when first used

This may be useful for expensive function calls, and should prevent double work in most cases. Sometimes it is useful to explicitly insert the Define node explicitly, in that case *bamboo.analysisutils.forceDefine()* can be used.

bamboo.treefunctions.**abs**(*sth*)

Return the absolute value

Example

```
>>> op.abs(t.Muon[0].p4.Eta())
```

bamboo.treefunctions.**sign**(*sth*)

Return the sign of a number

Example

```
>>> op.sign(t.Muon[0].p4.Eta())
```

bamboo.treefunctions.**sum**(**args*, ***kwargs*)

Return the sum of the arguments

Example

```
>>> op.sum(t.Muon[0].p4.Eta(), t.Muon[1].p4.Eta())
```

bamboo.treefunctions.**product**(**args*)

Return the product of the arguments

Example

```
>>> op.product(t.Muon[0].p4.Eta(), t.Muon[1].p4.Eta())
```

bamboo.treefunctions.**sqrt**(*sth*)

Return the square root of a number

Example

```
>>> m1, m2 = t.Muon[0].p4, t.Muon[1].p4
>>> m12dR = op.sqrt( op.pow(m1.Eta()-m2.Eta(), 2) + op.pow(m1.Phi()-m2.Phi(), 2) )
```

bamboo.treefunctions.**pow**(*base*, *exp*)

Return a power of a number

Example

```
>>> m1, m2 = t.Muon[0].p4, t.Muon[1].p4
>>> m12dR = op.sqrt( op.pow(m1.Eta()-m2.Eta(), 2) + op.pow(m1.Phi()-m2.Phi(), 2) )
```

bamboo.treefunctions.**exp**(*sth*)

Return the exponential of a number

Example

```
>>> op.exp(op.abs(t.Muon[0].p4.Eta()))
```

bamboo.treefunctions.**log**(*sth*)

Return the natural logarithm of a number

Example

```
>>> op.log(t.Muon[0].p4.Pt())
```

bamboo.treefunctions.**log10**(*sth*)

Return the base-10 logarithm of a number

Example

```
>>> op.log10(t.Muon[0].p4.Pt())
```

bamboo.treefunctions.**sin**(*sth*)

Return the sine of a number

Example

```
>>> op.sin(t.Muon[0].p4.Phi())
```

bamboo.treefunctions.**cos**(*sth*)

Return the cosine of a number

Example

```
>>> op.cos(t.Muon[0].p4.Phi())
```

bamboo.treefunctions.**tan**(*sth*)

Return the tangent of a number

Example

```
>>> op.tan(t.Muon[0].p4.Phi())
```

bamboo.treefunctions.**asin**(*sth*)

Return the arcsine of a number

Example

```
>>> op.asin(op.c_float(3.1415))
```

bamboo.treefunctions.**acos**(*sth*)

Return the arccosine of a number

Example

```
>>> op.acos(op.c_float(3.1415))
```

bamboo.treefunctions.**atan**(*sth*)

Return the arctangent of a number

Example

```
>>> op.atan(op.c_float(3.1415))
```

bamboo.treefunctions.**sinh**(*sth*)

Return the hyperbolic sine of a number

Example

```
>>> op.sinh(t.Muon[0].p4.Phi())
```

bamboo.treefunctions.**cosh**(*sth*)

Return the hyperbolic cosine of a number

Example

```
>>> op.cosh(t.Muon[0].p4.Phi())
```

bamboo.treefunctions.**tanh**(*sth*)

Return the hyperbolic tangent of a number

Example

```
>>> op.tanh(t.Muon[0].p4.Phi())
```

bamboo.treefunctions.**asinh**(*sth*)

Return the hyperbolic arcsine of a number

Example

```
>>> op.asinh(op.c_float(3.1415))
```

bamboo.treefunctions.**acosh**(*sth*)

Return the hyperbolic arccosine of a number

Example

```
>>> op.acosh(op.c_float(3.1415))
```

bamboo.treefunctions.**atanh**(*sth*)

Return the hyperbolic arctangent of a number

Example

```
>>> op.atanh(op.c_float(0.5))
```

bamboo.treefunctions.**max**(*a1*, *a2*)

Return the maximum of two numbers

Example

```
>>> op.max(op.abs(t.Muon[0].eta), op.abs(t.Muon[1].eta))
```

bamboo.treefunctions.**min**(*a1*, *a2*)

Return the minimum of two numbers

Example

```
>>> op.min(op.abs(t.Muon[0].eta), op.abs(t.Muon[1].eta))
```

bamboo.treefunctions.**in_range**(*low*, *arg*, *up*)

Check if a value is inside a range (boundaries excluded)

Example

```
>>> op.in_range(10., t.Muon[0].p4.Pt(), 20.)
```

bamboo.treefunctions.**withMass**(*arg*, *massVal*)

Construct a Lorentz vector with given mass (taking the other components from the input)

Example

```
>>> pW = withMass((j1.p4+j2.p4), 80.4)
```

bamboo.treefunctions.**invariant_mass**(**args*)

Calculate the invariant mass of the arguments

Example

```
>>> mElEl = op.invariant_mass(t.Electron[0].p4, t.Electron[1].p4)
```

Note

Unlike in the example above, *bamboo.treefunctions.combine()* should be used to make N-particle combinations in most practical cases

bamboo.treefunctions.**invariant_mass_squared**(**args*)

Calculate the squared invariant mass of the arguments using ROOT::Math::VectorUtil::InvariantMass2

Example

```
>>> m2ElEl = op.invariant_mass2(t.Electron[0].p4, t.Electron[1].p4)
```

bamboo.treefunctions.**deltaPhi**(*a1*, *a2*)

Calculate the difference in azimutal angles (using ROOT::Math::VectorUtil::DeltaPhi)

Example

```
>>> elelDphi = op.deltaPhi(t.Electron[0].p4, t.Electron[1].p4)
```

bamboo.treefunctions.**Phi_mpi_pi**(*a*)

Return an angle between -pi and pi

bamboo.treefunctions.**Phi_0_2pi**(*a*)

Return an angle between 0 and 2*pi

bamboo.treefunctions.**deltaR**(*a1*, *a2*)

Calculate the Delta R distance (using ROOT::Math::VectorUtil::DeltaR)

Example

```
>>> elelDR = op.deltaR(t.Electron[0].p4, t.Electron[1].p4)
```

bamboo.treefunctions.**rng_len**(*sth*)

Get the number of elements in a range

Parameters

rng – input range

Example

```
>>> nElectrons = op.rng_len(t.Electron)
```

bamboo.treefunctions.**rng_sum**(*rng*, *fun=<function <lambda>>*, *start=None*)

Sum the values of a function over a range

Parameters

- **rng** – input range
- **fun** – function whose value should be used (a callable that takes an element of the range and returns a number)
- **start** – initial value (0. by default)

Example

```
>>> totalMuCharge = op.rng_sum(t.Muon, lambda mu : mu.charge)
```

bamboo.treefunctions.**rng_count**(*rng*, *pred=None*)

Count the number of elements passing a selection

Parameters

- **rng** – input range
- **pred** – selection predicate (a callable that takes an element of the range and returns a boolean)

Example

```
>>> nCentralMu = op.rng_count(t.Muon, lambda mu : op.abs(mu.p4.Eta() < 2.4))
```

bamboo.treefunctions.**rng_product**(*rng*, *fun=<function <lambda>>*)

Calculate the production of a function over a range

Parameters

- **rng** – input range
- **fun** – function whose value should be used (a callable that takes an element of the range and returns a number)

Example

```
>>> overallMuChargeSign = op.rng_product(t.Muon, lambda mu : mu.charge)
```

bamboo.treefunctions.**rng_max**(*rng*, *fun=<function <lambda>>*)

Find the highest value of a function in a range

Parameters

- **rng** – input range
- **fun** – function whose value should be used (a callable that takes an element of the range and returns a number)

Example

```
>>> mostForwardMuEta = op.rng_max(t.Muon. lambda mu : op.abs(mu.p4.Eta()))
```

bamboo.treefunctions.**rng_min**(*rng*, *fun=<function <lambda>>*)

Find the lowest value of a function in a range

Parameters

- **rng** – input range
- **fun** – function whose value should be used (a callable that takes an element of the range and returns a number)

Example

```
>>> mostCentralMuEta = op.rng_min(t.Muon. lambda mu : op.abs(mu.p4.Eta()))
```

bamboo.treefunctions.**rng_max_element_index**(*rng*, *fun=<function <lambda>>*)

Find the index of the element for which the value of a function is maximal

Parameters

- **rng** – input range
- **fun** – function whose value should be used (a callable that takes an element of the range and returns a number)

Returns

the index of the maximal element in the base collection if rng is a collection, otherwise (e.g. if rng is a vector or array proxy) the index of the maximal element in rng

Example

```
>>> i_mostForwardMu = op.rng_max_element_index(t.Muon. lambda mu : op.abs(mu.p4.
↪Eta()))
```

bamboo.treefunctions.**rng_max_element_by**(*rng*, *fun=<function <lambda>>*)

Find the element for which the value of a function is maximal

Parameters

- **rng** – input range
- **fun** – function whose value should be used (a callable that takes an element of the range and returns a number)

Example

```
>>> mostForwardMu = op.rng_max_element_by(t.Muon. lambda mu : op.abs(mu.p4.Eta()))
```

bamboo.treefunctions.**rng_min_element_index**(*rng*, *fun=<function <lambda>>*)

Find the index of the element for which the value of a function is minimal

Parameters

- **rng** – input range
- **fun** – function whose value should be used (a callable that takes an element of the range and returns a number)

Returns

the index of the minimal element in the base collection if rng is a collection, otherwise (e.g. if rng is a vector or array proxy) the index of the minimal element in rng

Example

```
>>> i_mostCentralMu = op.rng_min_element_index(t.Muon. lambda mu : op.abs(mu.p4.
↪Eta()))
```

bamboo.treefunctions.**rng_min_element_by**(*rng*, *fun=<function <lambda>>*)

Find the element for which the value of a function is minimal

Parameters

- **rng** – input range
- **fun** – function whose value should be used (a callable that takes an element of the range and returns a number)

Example

```
>>> mostCentralMu = op.rng_min_element_by(t.Muon. lambda mu : op.abs(mu.p4.Eta()))
```

bamboo.treefunctions.**rng_mean**(*rng*)

Return the mean of a range

Parameters

rng – input range

Example

```
>>> pdf_mean = op.rng_mean(t.LHEPdfWeight)
```

bamboo.treefunctions.**rng_stddev**(*rng*)

Return the (sample) standard deviation of a range

Parameters
rng – input range

Example

```
>>> pdf_uncertainty = op.rng_stddev(t.LHEPdfWeight)
```

bamboo.treefunctions.**rng_any**(*rng*, *pred=<function <lambda>>*)

Test if any item in a range passes a selection

Parameters

- **rng** – input range
- **pred** – selection predicate (a callable that takes an element of the range and returns a boolean)

Example

```
>>> hasCentralMu = op.rng_any(t.Muon, lambda mu : op.abs(mu.p4.Eta()) < 2.4)
```

bamboo.treefunctions.**rng_find**(*rng*, *pred=<function <lambda>>*)

Find the first item in a range that passes a selection

Parameters

- **rng** – input range
- **pred** – selection predicate (a callable that takes an element of the range and returns a boolean)

Example

```
>>> leadCentralMu = op.rng_find(t.Muon, lambda mu : op.abs(mu.p4.Eta()) < 2.4)
```

bamboo.treefunctions.**select**(*rng*, *pred=<function <lambda>>*)

Select elements from the range that pass a cut

Parameters

- **rng** – input range
- **pred** – selection predicate (a callable that takes an element of the range and returns a boolean)

Example

```
>>> centralMuons = op.select(t.Muon, lambda mu : op.abs(mu.p4.Eta()) < 2.4)
```

bamboo.treefunctions.**sort**(*rng*, *fun=<function <lambda>>*)

Sort the range (ascendingly) by the value of a function applied on each element

Parameters

- **rng** – input range
- **fun** – function by whose value the elements should be sorted

Example

```
>>> muonsByCentrality = op.sort(t.Muon, lambda mu : op.abs(mu.p4.Eta()))
```

bamboo.treefunctions.**map**(*rng*, *fun*, *valueType=None*)

Create a list of derived values for a collection

This is useful for storing a derived quantity each item of a collection on a skim, and also for filling a histogram for each entry in a collection.

Parameters

- **rng** – input range
- **fun** – function to calculate derived values
- **valueType** – stored return type (optional, `fun(rng[i])` should be convertible to this type)

Example

```
>>> muon_absEta = op.map(t.Muon, lambda mu : op.abs(mu.p4.Eta()))
```

bamboo.treefunctions.**rng_pickRandom**(*rng*, *seed=0*)

Pick a random element from a range

Parameters

- **rng** – range to pick an element from
- **seed** – seed for the random generator

Caution

empty placeholder, to be implemented

bamboo.treefunctions.**svFitMTT**(*MET*, *lepton1*, *lepton2*, *category*)

Calculate the mass of the reconstructed tau pair using the SVfit algorithm. It employs the ClassicSVfit method.

Parameters

- **MET** – Missing transverse energy. It must include a covariance matrix.
- **lepton1** – 1st lepton (e/mu/tau) from the tau pair.
- **lepton2** – 2nd lepton (e/mu/tau) from the tau pair.
- **category** – Tau pair category. Only “1mu1tau,” “1ele1tau,” and “2tau” are supported.

Caution

This function works only if the SVfit package is installed.

bamboo.treefunctions.**svFitFastMTT**(*MET*, *lepton1*, *lepton2*, *category*)

Calculate the four-vector of the reconstructed tau pair using the SVfit algorithm. It employs the FastMTT method.

Parameters

- **MET** – Missing transverse energy. It must include a covariance matrix.
- **lepton1** – 1st lepton (e/mu/tau) from the tau pair.
- **lepton2** – 2nd lepton (e/mu/tau) from the tau pair.
- **category** – Tau pair category. Only “1mu1tau,” “1ele1tau,” and “2tau” are supported.

Caution

This function works only if the SVfit package is installed.

bamboo.treefunctions.**combine**(*rng*, *N=None*, *pred=<function <lambda>>*, *samePred=<function <lambda>>*)

Create N-particle combination from one or several ranges

Parameters

- **rng** – range (or iterable of ranges) with basic objects to combine
- **N** – number of objects to combine (at least 2), in case of multiple ranges it does not need to be given (`len(rng)` will be taken; if specified they should match)
- **pred** – selection to apply to candidates (a callable that takes the constituents and returns a boolean)
- **samePred** – additional selection for objects from the same base container (a callable that takes two objects and returns a boolean, it needs to be true for any sorted pair of objects from the same container in a candidate combination). The default avoids duplicates by keeping the indices (in the base container) sorted; `None` will not apply any selection, and consider all combinations, including those with the same object repeated.

Example

```
>>> osdimu = op.combine(t.Muon, N=2, pred=lambda mu1,mu2 : mu1.charge != mu2.charge)
>>> firstosdimu = osdimu[0]
>>> firstosdimu_Mll = op.invariant_mass(firstosdimu[0].p4, firstosdimu[1].p4)
>>> oselmu = op.combine((t.Electron, t.Muon), pred=lambda el,mu : el.charge != mu.
↪charge)
>>> trijet = op.combine(t.Jet, N=3, samePred=lambda j1,j2 : j1.pt > j2.pt)
>>> trijet = op.combine(
>>>     t.Jet, N=3, pred=lambda j1,j2,j3 : op.AND(j1.pt > j2.pt, j2.pt > j3.pt),␣
↪samePred=None)
```

Note

The default value for `samePred` undoes the sorting that may have been applied between the base container(s) and the argument(s) in `rng`. The third and fourth examples above are equivalent, and show how to get three-jet combinations, with the jets sorted by decreasing pT. The latter is more efficient since it avoids the unnecessary comparison `j1.pt > j3.pt`, which follows from the other two. In that case no other sorting should be done, otherwise combinations will only be retained if sorted by both criteria; this can be done by passing `samePred=None`.

`samePred=(lambda o1,o2 :  o1.idx != o2.idx)` can be used to get all permutations.

bamboo.treefunctions.**systematic**(*nominal*, *name=None*, ***kwargs*)

Construct an expression that will change under some systematic variations

This is useful when e.g. changing weights for some systematics. The expressions for different variations are assumed (but not checked) to be of the same type, so this should only be used for simple types (typically a number or a range of numbers); containers etc. need to be taken into account in the decorators.

Example

```
>>> psWeight = op.systematic(tree.ps_nominal, name="pdf", up=tree.ps_up, down=tree.
↪ps_down)
>>> addSys10percent = op.systematic(
>>>     op.c_float(1.), name="additionalSystematic1", up=op.c_float(1.1), down=op.c_
↪float(0.9))
>>> importantSF = op.systematic(op.c_float(1.),
        mySF_systup=op.c_float(1.1), mySF_systdown=op.c_float(0.9),
        mySF_statup=1.04, mySF_statdown=.97)
```

Parameters

- **nominal** – nominal expression
- **kwargs** – alternative expressions. "up" and "down" (any capitalization) will be prefixed with name, if given
- **name** – optional name of the systematic uncertainty source to prepend to "up" or "down"

bamboo.treefunctions.**getSystematicVariations**(*expr*)

Get the list of systematic variations affecting an expression

bamboo.treefunctions.**forSystematicVariation**(*expr*, *varName*)

Get the equivalent expression with a specific systematic uncertainty variation

Parameters

- **expr** – an expression (or proxy)
- **varName** – name of the variation (e.g. `jesTotalup`)

Returns

the expression for the chosen variation (frozen, so without variations)

class bamboo.treefunctions.**MVAEvaluator**(*evaluate*, *returnType=None*, *toArray=False*, *toVector=True*, *useSlots=False*)

Small wrapper to make sure MVA evaluation is cached

bamboo.treefunctions.**mvaEvaluator**(*fileName*, *mvaType=None*, *otherArgs=None*, *nameHint=None*)

Declare and initialize an MVA evaluator

The C++ object is defined (with *bamboo.treefunctions.define()*), and can be used as a callable to evaluate. The result of any evaluation will be cached automatically.

Currently the following formats are supported:

- .xml (`mvaType='TMVA'`) TMVA weights file, evaluated with a `TMVA::Experimental::RReader`
- .pt (`mvaType='Torch'`) pytorch script files (loaded with `torch::jit::load`).
- **.pb (`mvaType='Tensorflow'`) tensorflow graph definition (loaded with Tensorflow-C).**
  The `otherArgs` keyword argument should be `(inputNodeNames, outputNodeNames)`, where each of the two can be a single string, or an iterable of them. In the case of multiple input nodes, the input values for each should also be passed as separate arguments when evaluating (see below). Input values for multi-dimensional nodes should be flattened (row-order per node, and then the different nodes). The output will be flattened in the same way if the output node has more than one dimension, or if there are multiple output nodes.
- .json (`mvaType='lwtnn'`) lwtnn json. The `otherArgs` keyword argument should be passed the lists of input and output nodes/values, as C++ initializer list strings, e.g. `'{ { "node_0", "variable_0" }, { "node_0", "variable_1" } ... }'` and `'{ "out_0", "out_1" }'`.

- .onnx (`mvaType='ONNXRuntime'`) ONNX file, evaluated with ONNX Runtime. The `otherArgs` keyword argument should be the name of the output node (or a list of those)
- .hxx (`mvaType='SOFIE'`) ROOT SOFIE generated header file The `otherArgs` keyword argument should be the path to the `.dat` weights file (if not specified, it will taken by replacing the weight file extension from `.hxx` to `.dat`). Note: only available in ROOT>=6.26.04.
- .root (`mvaType='RBDT'`) XGBoost BDT model stored in ROOT file using `TMVA::Experimental::SaveXGBoost`, and evaluated with a `TMVA::Experimental::RBDT`. The `otherArgs` keyword argument should specify the name used to store the RBDT in the output file.

Parameters

- **`fileName`** – file with MVA weights and structure
- **`mvaType`** – type of MVA, or library used to evaluate it (Tensorflow, Torch, or lwtnn). If absent, this is guessed from the fileName extension
- **`otherArgs`** – other arguments to construct the MVA evaluator (either as a string (safest), or as an iterable)
- **`nameHint`** – name hint, see *bamboo.treefunctions.define()*

Returns

a proxy to a method that takes the inputs as arguments, and returns a `std::vector<float>` of outputs

For passing the inputs to the evaluator, there are two options

- if a list of numbers is passed, as in the example below, they will be converted to an array of `float` (with a `static_cast`). The rationale is that this is the most common simple case, which should be made as convenient as possible.
- if the MVA takes inputs in a different type than `float` or has multiple input nodes (supported for Tensorflow and ONNX Runtime), an array-like object of the correct type should be passed for each of the input nodes. No other conversions will be automatically inserted, so these should be done when constructing the inputs (e.g. with *array()* and `initList()`). This is a bit more work, but gives maximal control over the generated code.

Example

```
>>> mu = tree.Muon[0]
>>> nn1 = mvaEvaluator("nn1.pt")
>>> Plot.make1D("mu_nn1", nn1(mu.pt, mu.eta, mu.phi), hasMu)
```

Warning

By default the MVA output will be added as a column (`Define` node in the RDataFrame graph) when used, because it is almost always more efficient. In some cases, e.g. if the MVA should only be evaluated if some condition is true, this can cause problems. To avoid this, *defineOnFirstUse=False* should be passed when calling the evaluation, e.g. *nn1(mu.pt, mu.eta, mu.phi, defineOnFirstUse=False)* in the example above.

2.4 Recipes for common tasks

2.4.1 Using scalefactors

Scalefactors—CMS jargon for efficiency corrections for MC, typically binned in lepton or jet kinematic variables—can be generalized to functions that take some properties of a physics object and return a single floating-point number. The *bamboo.scalefactors* module provides support for constructing such callable objects from two different JSON formats, the CMS correctionlib format, and the one used in the CP3-llbb framework, and the CMS BTV CSV format.

CMS correctionlib JSON format

The *bamboo.scalefactors.get_correction()* method loads a `Correction` from the `CorrectionSet` stored in a JSON file, and constructs a helper object to use it in bamboo. Since corrections are usually parameterised as function of e.g. kinematic properties of a reconstructed object, callables can be passed as parameters, and the helper object called with a reconstructed object, e.g.

```
from bamboo.scalefactors import get_correction
sf = get_correction(..., params={"pt": lambda obj : obj.pt, ...})
mySel = noSel.refine(..., weight=sf(el))
```

Many of the arguments to *bamboo.scalefactors.get_correction()* are related to automatic systematic uncertainties: the name of a category axis in the `Correction`, the mapping between its categories and systematic variations in bamboo, and the name of the nominal category—more details and a complete example can be found in the reference documentation. Please note that this needs the correctionlib package to be installed, see *the installation guide* for more details.

Helper methods to configure and combine individual corrections for the purpose of applying b-tagging scale factor and uncertainties are provided, see *bamboo.scalefactors.makeBtagWeightMeth1a()* and *bamboo.scalefactors.makeBtagWeightItFit()*.

CP3-llbb JSON format

> **Warning**
>
> The CP3-llbb json format and associated Bamboo functionalities are soon going to be deprecated in favour of the central JSON format and correctionlib (see above).

The *bamboo.scalefactors* module provides support for constructing such callable objects from the JSON format used in the CP3-llbb framework, see some examples here (these JSON files are produced from the txt or ROOT files provided by the POGs using simple python scripts). Like their inputs, the JSON files contain the nominal scale factor as well as its up and down systematic variations, so the `ScaleFactor` behaves as a callable that takes a physics object and an optional `variation` keyword argument (technically, it wraps a C++ object that gets the correct value from the JSON file).

The *get_scalefactor()* method constructs such objects from a nested dictionary: the first key is a tag (as an example: "electron_2015_76", for electrons in 2015 data, analysed with a `CMSSW_7_6_X` release) and the second key is an identifier of the selection they correspond to (e.g. `id_Loose`). The value inside this dictionary can be either a single path to a JSON file, or a list of `(periods, path)` pairs, where `periods` is a list of run periods, in case scalefactors for different running periods need to be combined (the `periods` keyword argument to *get_scalefactor()* can be used to select only a certain set of these periods). The combination is done by either weighting or randomly sampling from the different periods, according to the fraction of the integrated luminosity in each (by passing `combine="weight"` or `combine="sample"`, respectively). Jet flavour tagging and dilepton (e.g. trigger) scalefactors can also be specified by passing tuples of the light, c-jet and b-jet scalefactor paths, and tuples of first-if-leading, first-if-subleading, second-if-leading, and second-if-subleading (to be reviewed for NanoAOD) scalefactor paths, respectively, instead of a single

path.

Histogram variations representing the shape systematic uncertainty due to an uncertainty on the scalefactor values can be automatically produced by passing a name to the `systName` keyword argument of the *get_scalefactor()* method.

As an example, some basic lepton ID and jet tagging scalefactors could be included in an analysis on NanoAOD by defining

```
import bamboo.scalefactors
from itertools import chain
import os.path

# scalefactor JSON files are in ScaleFactors/<era>/ alongside the module
def localize_myanalysis(aPath, era="2016legacy"):
    return os.path.join(os.path.dirname(os.path.abspath(__file__)), "ScaleFactors", era,␣
↪aPath)

# nested dictionary with path names of scalefactor JSON files
# { tag : { selection : absole-json-path } }
myScalefactors = {
    "electron_2016_94" : {
        "id_Loose"  : localize_myanalysis("Electron_EGamma_SF2D_Loose.json")
        "id_Medium" : localize_myanalysis("Electron_EGamma_SF2D_Medium.json")
        "id_Tight"  : localize_myanalysis("Electron_EGamma_SF2D_Tight.json")
    },
    "btag_2016_94" : dict((k, (tuple(localize_myanalysis(fv) for fv in v))) for k,v in␣
↪dict(
        ( "{algo}_{wp}".format(algo=algo, wp=wp),
          tuple("BTagging_{wp}_{flav}_{calib}_{algo}.json".format(wp=wp, flav=flav,␣
↪calib=calib, algo=algo)
              for (flav, calib) in (("lightjets", "incl"), ("cjets", "comb"), ("bjets",
↪"comb")))
        ) for wp in ("loose", "medium", "tight") for algo in ("DeepCSV", "DeepJet") ).
↪items())
    }

# fill in some defaults: myScalefactors and bamboo.scalefactors.binningVariables_nano
def get_scalefactor(objType, key, periods=None, combine=None, additionalVariables=None,␣
↪systName=None):
    return bamboo.scalefactors.get_scalefactor(objType, key, periods=periods,␣
↪combine=combine,
        additionalVariables=(additionalVariables if additionalVariables else dict()),
        sfLib=myScalefactors, paramDefs=bamboo.scalefactors.binningVariables_nano,␣
↪systName=systName)
```

and adding the weights to the appropriate *Selection* instances with

```
electrons = op.select(t.Electron, lambda ele : op.AND(ele.cutBased >= 2, ele.p4.Pt() >␣
↪20., op.abs(ele.p4.Eta()) < 2.5))
elLooseIDSF = get_scalefactor("lepton", ("electron_2016_94", "id_Loose"), systName="elID
↪")
hasTwoEl = noSel.refine("hasTwoEl", cut=[ op.rng_len(electrons) > 1 ],
              weight=[ elLooseIDSF(electrons[0]), elLooseIDSF(electrons[1]) ])
```

(continues on next page)

(continued from previous page)

```
jets = op.select(t.Jet, lambda j : j.p4.Pt() > 30.)
bJets = op.select(jets, lambda j : j.btagDeepFlavB > 0.2217) ## DeepFlavour loose b-tag␣
↪working point
deepFlavB_discriVar = { "BTagDiscri": lambda j : j.btagDeepFlavB }
deepBLooseSF = get_scalefactor("jet", ("btag_2016_94", "DeepJet_loose"),␣
↪additionalVariables=deepFlavB_discriVar, systName="bTag")
hasTwoElTwoB = hasTwoEl.refine("hasTwoElTwoB", cut=[ op.rng_len(bJets) > 1 ],
                  weight=[ deepBLooseSF(bJets[0]), deepBLooseSF(bJets[1]) ])
```

Note that the user is responsible for making sure that the weights are only applied to simulated events, and not to real data!

CMS BTV CSV format

> **Warning**
>
> The BTV CSV reader and associated Bamboo functionalities are soon going to be deprecated in favour of the central JSON format and correctionlib (see above).

The *bamboo.scalefactors.BtagSF* class wraps the `BTagCalibrationReader` provided by the BTV POG to read the custom CSV format for b-tagging scalefactors (more details usage instructions can be found in the reference documentation). Please note that this will only read the scalefactors, which for most methods for applying b-tagging scalefactors need to be combined with efficiency and mistag probability maps measured in simulation in the analysis phase space.

2.4.2 Pileup reweighting

> **Warning**
>
> The pileup weights maker and associated Bamboo functionalities are soon going to be deprecated in favour of the central JSON format and correctionlib (see above).

Pileup reweighting to make the pileup distribution in simulation match the one in data is very similar to applying a scalefactor, except that the efficiency correction is for the whole event or per-object—so the same code can be used. The `makePUReWeightJSON` script included in bamboo can be used to make a JSON file with weights out of a data pileup profile obtained by running `pileupcalc.py` (inside CMSSW, see the pileupcalc documentation for details), e.g. with something like

```
pileupCalc.py -i ~/Cert_271036-284044_13TeV_23Sep2016ReReco_Collisions16_JSON.txt --
↪inputLumiJSON /afs/cern.ch/cms/CAF/CMSCOMM/COMM_DQM/certification/Collisions16/13TeV/
↪PileUp/pileup_latest.txt --calcMode true --minBiasXsec 69200 --maxPileupBin 80 --
↪numPileupBins 80 ./2016PUHist_nominal.root
```

and a MC pileup profile. Data pileup distributions corresponding to the golden JSON files for Run 2 are provided by the luminosity POG, see this hypernews annoncement. The MC pileup profiles for used official CMSSW productions are currently hardcoded inside the `makePUReWeightJSON`, and can be specified by their tag or name in that list; the available tags can be listed by specifying the `--listmcprofiles` option. The full command then becomes something like

```
makePUReWeightJSON --mcprofile "Moriond17_25ns" --nominal=2016PUHist.root --
↪up=2016PUHist_up.root --down=2016PUHist_down.root --makePlot
```

To include the weight when filling plots, it is sufficient to add the weight to a selection (usually one of the topmost in the analysis, e.g. in the `prepareTree` method of the analysis module). The *bamboo.analysisutils.makePileupWeight()* method can be used to build an expression for the weight, starting from the path of the JSON file with weights from above, and an expression for the true number of interactions in the event (mean of the Poissonian used), e.g. `tree.Pileup_nTrueInt` for NanoAOD.

2.4.3 Cleaning collections

The CMS reconstruction sometimes ends up double-counting some objects. This can be because of the different quality criteria used to identify each object or because of the different performance and inner working of the reconstruction algorithms. Tau reconstruction for example operates on clusters that are usually reconstructed as jets, and on top of that it can easily pick up even isolated muons or electrons as taus (i.e. as clusters of energy with one, two, or three tracks).

It is oftentimes necessary therefore to clean a collection of objects by excluding any object that is spatially in the sample place of another object whose reconstruction we trust more.

We trust more muon and electron reconstrution than tau reconstruction, after all the quality cuts (ID efficiencies for muons and electrons are around 99.X%, whereas tau ID efficiencies are of the order of 70%. Misidentification rates are similarly quite different), and therefore we exclude from the tau collection any tau that happens to include within its reconstruction cone a muon or an electron.

Bamboo provides a handy syntax for that, resulting in something like

```
cleanedTaus = op.select(taus, lambda it : op.AND(
      op.NOT(op.rng_any(electrons, lambda ie : op.deltaR(it.p4, ie.p4) < 0.3 )),
      op.NOT(op.rng_any(muons, lambda im : op.deltaR(it.p4, im.p4) < 0.3 ))
      ))
```

In this example, we assume that the collections `taus`, `electrons`, and `muons`, have already been defined via `taus = op.select(t.Tau, lambda tau : ...)`, and we move on to use the method `op.rng_any()` to filter all taus that are within a cone of a given size (0.3, in the example) from any selected electron or muon.

2.4.4 Jet and MET systematics

For propagating uncertainties related to the jet energy calibration, and the difference in jet energy resolution between data and simulation, each jet in the reconstructed jet collection should be modified, the collection sorted, and any derived quantity re-evaluated.

How to do this depends on the input trees: in production NanoAOD the modified momenta need to be calculated using the jet energy correction parameters; it is also possible to add them when post-processing with the jetmetUncertainties module of the NanoAODTools package. In the latter case the NanoAOD decoration method will pick up the modified branches if an appropriate `NanoSystematicVarSpec` entry (e.g. `nanoReadJetMETVar`) is added to the `systVariations` attribute of the *NanoAODDescription* that is passed to the *prepareTree()* (or *decorateNanoAOD()*) method.

To calculate the variations on the fly, two things are needed: when decorating the tree, some redirections should be set up to pick up the variations from a calculator module, and then this module needs to be configured with the correct JEC and resolution parameters. The first step can be done by adding `nanoJetMETCalc` to the `systVariations` attribute of the *NanoAODDescription* that is passed to the *prepareTree()* method (which will pass this to the *decorateNanoAOD()* method); these will also make sure that all these variations are propagated to the missing transverse momentum. Next, a calculator must be added and configured. This can be done with the *bamboo.analysisutils.configureJets()* and *bamboo.analysisutils.configureType1MET()* methods, which provide a convenient way to correct the jet

resolution in MC, apply a different JEC, and add variations due to different sources of uncertainty in the jet energy scale, for the jet collection and MET, respectively (the arguments should be the same in most cases).

Note

The jet and MET calculators were moved to a separate package. Since these calculators are C++ classes with an RDF-friendly interface and minimal dependencies, they are not only useful from bamboo, but also from other (RDF-based or similar) frameworks. Therefore, they were moved to a separate repository cp3-cms/CMSJMECalculators. They can be installed with e.g. `pip install git+https://gitlab.cern.ch/cp3-cms/CMSJMECalculators.git`.

As an example, the relevant code of a NanoAOD analysis module could look like this to apply a newer JEC to 2016 data and perform smearing, add uncertainties to 2016 MC, and the same for the MET:

```
class MyAnalysisModule(NanoAODHistoModule):
    def prepareTree(self, tree, sample=None, sampleCfg=None):
        tree,noSel,be,lumiArgs = super(MyAnalysisModule, self).prepareTree(tree,␣
↪sample=sample, sampleCfg=sampleCfg
          , NanoAODDescription.get("v5", year="2016", isMC=self.isMC(sample),␣
↪systVariations=[nanoJetMETCalc]))
        from bamboo.analysisutils import configureJets, configureType1MET
        era = sampleCfg["era"]
        if era == "2016":
            if self.isMC(sample): # can be inferred from sample name
                configureJets(tree._Jet, "AK4PFchs",
                    jsonFile="jet_jerc_2016.json.gz",
                    jec="Summer16_07Aug2017_V20_MC",
                    smear="Summer16_25nsV1_MC",
                    jsonFileSmearingTool="jer_smear.json.gz",
                    jesUncertaintySources=["Total"],
                    isMC=self.isMC(sample), backend=be)
                configureType1MET(tree._MET,
                    jsonFile="jet_jerc_2016.json.gz",
                    jec="Summer16_07Aug2017_V20_MC",
                    smear="Summer16_25nsV1_MC",
                    jsonFileSmearingTool="jer_smear.json.gz",
                    jesUncertaintySources=["Total"],
                    isMC=self.isMC(sample), backend=be)
            else:
                if "2016G" in sample or "2016H" in sample:
                    configureJets(tree._Jet, "AK4PFchs",
                        jsonFile="jet_jerc_2016.json.gz",
                        jec="Summer16_07Aug2017GH_V11_DATA",
                        isMC=self.isMC(sample), backend=be)
                    configureType1MET(tree._MET,
                        jsonFile="jet_jerc_2016.json.gz",
                        jec="Summer16_07Aug2017GH_V11_DATA",
                        isMC=self.isMC(sample), backend=be)
                elif ...: ## other 2016 periods
                    pass

        return tree,noSel,be,lumiArgs
```

Both with variations read from a postprocessed NanoAOD and calculated on the fly, the different jet collections are available from `t._Jet`, e.g., `t._Jet["nom"]` (postprocessed) or `t._Jet["nominal"]` (calculated), `t._Jet["jerup"]`, `t._Jet["jerdown"]`, `t._Jet["jesTotalUp"]`, `t._Jet["jesTotalDown"]`, etc., depending on the configured variations. When accessing these directly, `t._Jet[variation][j.idx]` should be used to retrieve the entry corresponding to a specific jet `j`, if the latter is obtained from a selected and/or sorted version of the original collection—`object.idx` is always the index in the collection as found in the tree).

`t.Jet` will be changed for one of the above for each systematic variation, if it affects a plot, in case automatically producing the systematic variations is enabled (the collections from `t._Jet` will not be changed). The automatic calculation of systematic variations can be disabled globally or on a per-selection basis (see above), and for on-the-fly calculation also by passing `enableSystematics=[]` to *bamboo.analysisutils.configureJets()*). The jet collection as stored on the input file, finally, can be retrieved as `t._Jet.orig`.

Important

Sorting the jets No sorting is done as part of the above procedure, so if relevant this should be added by the user (e.g., using `jets = op.sort(t.Jet, lambda j : -j.pt)` for sorting by decreasing transverse momentum). In a previous version of the code, this was included, but since some selection is usually applied on the jets anyway, it is simpler (and more efficient) to perform the sorting then.

Note

Isn't it slow to calculate jet corrections on the fly? It does take a bit of time, but the calculation is done in one C++ module, which should not be executed more than once per event (see the explanation of the *bamboo.analysisutils.forceDefine()* method in the *section above*). In most realistic cases, the bottleneck is in reading and decompressing the input files, so the performance hit from the jet corrections should usually be acceptable.

2.4.5 Lepton momentum scale correction

The momentum of muons and electrons also requires correction. For data, a scale correction is applied. In simulation, additional smearing is introduced to match the resolution observed in data. These corrections improve the agreement between data and simulation, particularly in the description of the Z boson peak.

To calculate the correction on the fly, two things are needed: when decorating the tree, some redirections should be set up to pick up the variations from a calculator module, and then this module needs to be configured with the correct input parameters. The first step can be done by adding `nanoMuonCalc` and `nanoElectronCalc` to the `systVariations` attribute of the *NanoAODDescription* that is passed to the *prepareTree()* method (which will pass this to the *decorateNanoAOD()* method). Next, a calculator must be added and configured. This can be done with the *bamboo.analysisutils.configureMuons()* and *bamboo.analysisutils.configureElectrons()* methods, which provide a convenient way to correct the muon and electron momentum in data and MC, and add variations due to different sources of uncertainty in the lepton momentum scale correction, for the muon and electron collection, respectively.

```
class MyAnalysisModule(NanoAODHistoModule):
    def prepareTree(self, tree, sample=None, sampleCfg=None):
        tree,noSel,be,lumiArgs = NanoAODHistoModule.prepareTree(self, tree,␣
↪sample=sample, sampleCfg=sampleCfg, calcToAdd=["nMuon", "nElectron"])
        from bamboo.analysisutils import configureMuons, configureElectrons
        isMC = self.isMC(sample)

        configureElectrons(tree._Electron, paramsFile="electronSS_EtDependent_2022EE.
↪json.gz",
```

(continues on next page)

(continued from previous page)

```
                            scale="EGMScale_Compound_Ele_2022postEE",
                            smearing="EGMSmearAndSyst_ElePTsplit_2022postEE",
                            jsonFileRandomGenerator="randomNumbers.json.gz",
                            addSystematics=True if isMC else False, isMC=isMC, backend=be)

        configureMuons(tree._Muon, paramsFile="MES_Summer_2022EE.json.gz",
                       jsonFileRandomGenerator="randomNumbers.json.gz",
                       addSystematics=True if isMC else False, isMC=isMC, backend=be)

        return tree,noSel,be,lumiArgs
```

2.4.6 Rochester correction for muons

The so-called Rochester correction removes a bias in the muon momentum, and improves the agreement between data and simulation in the description of the Z boson peak. As for the jet correction and variations described in the previous section, this can either be done during postprocessing, with the muonScaleResProducer module of the NanoAODTools package, or on the fly. To adjust the decorators, a suitable *NanoSystematicVarSpec* instance to read the corrected values, or `nanoRochesterCalc` to use the calculated values, should be added to the `systVariations` attribute of the *NanoAODDescription* that is passed to the *prepareTree()* (or *decorateNanoAOD()*) method.

The on the fly calculator should be added and configured with the *bamboo.analysisutils.configureRochesterCorrection()* method, as in the example below. `tree._Muon` keeps track of everything related to the calculator; the uncorrected muon collection can be found in `tree._Muon.orig`, and the corrected muons are in `tree.Muon`.

```
class MyAnalysisModule(NanoAODHistoModule):
    def prepareTree(self, tree, sample=None, sampleCfg=None):
        tree,noSel,be,lumiArgs = NanoAODHistoModule.prepareTree(self, tree,␣
↪sample=sample, sampleCfg=sampleCfg, calcToAdd=["nMuon"])
        from bamboo.analysisutils import configureRochesterCorrection
        era = sampleCfg["era"]
        if era == "2016":
            configureRochesterCorrection(tree._Muon, "RoccoR2016.txt", isMC=self.
↪isMC(sample), backend=be)
    return tree,noSel,be,lumiArgs
```

Caution

The Rochester correction for muons are available only for Run 2 analyses. For Run 3 analyses, the *bamboo.analysisutils.configureMuons()* method (*see section above*) should be used.

2.4.7 Energy correction for taus

Similar to muons and electrons, the energy of taus also requires correction. This can be done on the fly. To adjust the decorators, `nanoTauESCalc` to use the calculated values, should be added to the `systVariations` attribute of the *NanoAODDescription* that is passed to the *prepareTree()* (or *decorateNanoAOD()*) method.

The on-the-fly calculator should be added and configured with the *bamboo.analysisutils.configureTaus()* method, as in the example below. `tree._Tau` keeps track of everything related to the calculator; the uncorrected tau collection can be found in `tree._Tau.orig`, and the corrected taus are in `tree.Tau`.

```
class MyAnalysisModule(NanoAODHistoModule):
    def prepareTree(self, tree, sample=None, sampleCfg=None):
        tree,noSel,be,lumiArgs = NanoAODHistoModule.prepareTree(self, tree,␣
↪sample=sample, sampleCfg=sampleCfg, calcToAdd=["nTau"])
        from bamboo.analysisutils import configureTaus
        if self.isMC(sample):
            configureTaus(tree._Tau, "tau_DeepTau2018v2p5_2022EE.json.gz",
                          tauIdAlgo="DeepTau2018v2p5", tauCorr="tau_energy_scale",
                          tauWP="Loose", tauWPvsE="VVLoose", addSystematics=True,
                          isMC=self.isMC(sample),backend=be)
        return tree,noSel,be,lumiArgs
```

2.4.8 Correlating systematic variations

To understand how systematic variations are implemented in bamboo, and how to take advantage of that to correlate e.g. a b-tagging scalefactor variation with a jet and MET kinematic variation, it is useful to remember that your code creates *expressions* that are converted to C++ code, and imagine a variable with a systematic uncertainty as a single nominal value with a dictionary of alternative values: the keys of this dictionary are the variation names, e.g. `elIDup` or `jerdown`. This is also how they are represented in the expression objects tree. When creating a plot or selection, the variable(s), weight(s), and cut(s) are scanned for such nodes with systematic variations, and additional RDataFrame nodes are created for all the variations.

There are two interesting consequences of this dictionary with variations: all variations are equal, i.e. there is no concept of "uncertainty X with e.g. up and down variations"—although this is very common in practice, and trivial to reconstruct from the dictionary where needed—and all expression nodes with the same variation change together. The latter is necessary in many cases, e.g. when passing the MET and some jet pTs to a multivariate classifier, both should pass the `jerdown` variation to get the corresponding variation of the classifier output. It also provides a very transparent way to correlate variations: if the name is the same, they will be simultaneously varied—so it is enough that a b-tagging scalefactor variation is called `jesAbsup` to be varied together with that variation of the jet pt's; turning that around: to be varied independently, the names must be made different (this is why `up` and `down` by themselves as variation names lead to an error message being printed).

2.4.9 Splitting a sample into sub-components

It is frequently necessary to split a single Monte-Carlo sample into different processes, depending on generator-level information, or simply to add some cuts at generator level (e.g. to stitch binned samples together). This can be achieved by duplicating that sample in the analysis configuration file for as many splits as needed, and putting (any) additional information into that sample's entry, e.g. as:

```
ttbb:
  db: das:/TTToSemiLeptonic_TuneCP5_13TeV-powheg-pythia8/RunIIAutumn18NanoAODv5-
↪Nano1June2019_102X_upgrade2018_realistic_v19-v1/NANOAODSIM
  era: 2018
  group: ttbb
  subprocess: ttbb
  cross-section: 366.
  generated-events: genEventSumw

ttjj:
  db: das:/TTToSemiLeptonic_TuneCP5_13TeV-powheg-pythia8/RunIIAutumn18NanoAODv5-
↪Nano1June2019_102X_upgrade2018_realistic_v19-v1/NANOAODSIM
  era: 2018
  group: ttjj
```

(continues on next page)

(continued from previous page)

```
  subprocess: ttjj
  cross-section: 366.
  generated-events: genEventSumw
```

That information can then be retrieved in the analysis module through the `sampleCfg` keyword argument, to add additional cuts to the selection when preparing the tree:

```
def prepareTree(self, tree, sample=None, sampleCfg=None):
    tree,noSel,be,lumiArgs = super(MyAnalysisModule, self).prepareTree(tree,␣
↪sample=sample, sampleCfg=sampleCfg)

    if "subprocess" in sampleCfg:
         subProc = sampleCfg["subprocess"]
         if subProc == "ttbb":
             noSel = noSel.refine(subProc, cut=(tree.genTtbarId % 100) >= 52)
         elif subProc == "ttjj":
             noSel = noSel.refine(subProc, cut=(tree.genTtbarId % 100) < 41)

    return tree,noSel,be,lumiArgs
```

2.4.10 Adding command-line arguments

The base *analysis module*, *bamboo.analysismodules.AnalysisModule*, calls the *addArgs()* method (the default implementation does nothing) when constructing the command-line arguments parser (using the argparse module). Analysis modules can reimplement this method to specify more arguments, e.g.

```
class MyModule(...):

    def addArgs(self, parser):
        super(MyModule, self).addArgs(parser)
        parser.add_argument("--whichPlots", type=str,
                            default="control",
                            help="Set of plots to produce")
```

The parsed arguments are available under the `args` member variable, e.g. `self.args.whichPlots` in the example above. The complete list of command-line options (including those specified in the analysis module) can be printed with `bambooRun -h -m myModule.py.MyModule`. In fact, the parser argument is an argument group, so they are listed separately from those in the base class. This is also used to copy all user-defined arguments to the commands that are passed to the worker tasks, when running in distributed mode.

2.4.11 Editing the analysis configuration

Similarly to the above, it is possible to modify the analysis configuration (loaded from the YAML file) from a module before the configuration is used to create jobs (in distributed mode), run on any file (in sequential mode), or run plotIt (in the postprocessing step). This allows e.g. to change the samples that are going to be used, change the list of systematics, etc., without having to manually edit the YAML file or maintaining separate files. Below is an example of how this works:

```
class MyModule(...):

    def customizeAnalysisCfg(self, analysisCfg):
        for smp in list(analysisCfg["samples"]):
```

(continues on next page)

(continued from previous page)

```
        if not analysisCfg["samples"][smp].get("is_signal", False):
            del analysisCfg["samples"][smp]
```

2.4.12 Evaluate an MVA classifier

Several external libraries can be used to evaluate the response of MVA classifiers inside expressions. For convenience, a uniform interface is defined that uses a vector of floats as input and output, with implementations available for PyTorch, TensorFlow, lwtnn, TMVA, ONNX Runtime, and SOFIE. That works as follows (see the documentation for the *bamboo.treefunctions.mvaEvaluator()* method for a detailed description, additional options may be needed, depending on the type):

```
mu = tree.Muon[0]
nn1 = mvaEvaluator("nn1.pt", mvaType="Torch")
Plot.make1D("mu_nn1", nn1(mu.pt, mu.eta, mu.phi), hasMu)
```

For TensorFlow, PyTorch, and ONNX Runtime multiple inputs (and inputs with different types) are also supported. In that case, no automatic conversion is performed, so the inputs should be passed in the correct format (most of the time the number of inputs per node is known, so arrays constructed with *bamboo.treefunctions.array()* are a good choice).

> **Warning**
>
> Especially for PyTorch and TensorFlow, setting up an installation where the necessary C(++) libraries are correctly identified, and compatible with the CPU capabilities, is not always trivial. See *this section* in the installation guide for more information.

Skims for training a classifier can also be produced straightforwardly with bamboo.

Obtaining a classifier in the right format

All MVA inference is done through the C(++) APIs provided by the different machine learning and inference libraries, which means that the model should be stored in the appropriate format (often with some conversion step).

ONNX and lwtnn are formats for the exchange and inference of neural networks, so they need converters from the model building and/or training framework. Converting Keras models to lwtnn is described on the lwtnn wiki. PyTorch comes with ONNX export included. Most Keras models can also easily be exported to ONNX with keras2onnx.

The PyTorch evaluator uses TorchScript, this tutorial is a very good starting point if your model is trained with PyTorch.

TMVA uses an XML format which probably also just works. The TMVA reader will work with multi-threading, but the reader class uses locking because the internal TMVA classes are not thread-safe, so performance will be degraded if aggressive multi-threading is used and the TMVA evaluation dominates the CPU usage.

For Keras models TensorFlow is the most natural fit. Please note that the frozen graph is needed, see e.g. keras_to_tensorflow, this detailed explanation, and this script for an example of how to do so.

SOFIE allows one to evaluate models in ONNX, PyTorch or Keras format, provided they have been first converted into a header and weight files using the helpers available in ROOT (see the ROOT documentation and tutorials for how to convert models). While a limited set of models are supported (only a few types of layers are implemented in SOFIE), if the conversion is possible, model evaluation in SOFIE has the potential to be significantly faster than using the ONNX, TensorFlow or PyTorch APIs. Note that SOFIE is only supported in ROOT >= 6.26.04 but it is not enabled by default, so you'll need to make sure that your ROOT build has SOFIE enabled.

Testing the evaluation outside RDataFrame

MVA inference with all the libraries described above is done by creating an instance of an evaluator class, which provides a similar RDataFrame-friendly interface: the filename of the saved model and additional options are passed to the constructor, and an evaluate method that takes the input values and returns the MVA outputs is called from inside the RDataFrame graph. It is straightforward to do the same from PyROOT: for each framework there is a method in the *bamboo.root* to load the necessary shared libraries and the evaluator class. After calling this method, an evaluator can be instantiated and tested with some simple arguments. This is done in the bamboo tests, so these can serve as an example (links for the the relevant fragments: test_tensorflow, test_lwtnn, test_libtorch; TMVA is directly included in ROOT, so it is sufficient to retrieve the `TMVA::Experimental::RReader` class).

2.4.13 Make combined plots for different selections

It is rather common to define categories with e.g. different lepton flavours and selections, but then make plots with the entries from these (disjoint) sets of events combined. Given the structure of the RDataFrame graph and the *Selection* tree, the most convenient way to achieve this is by defining the histograms for each category, and make a merged histogram later on. The *SummedPlot* class does exactly this, and since it presents the same interface to the analysis module as a regular *Plot*, it can simply be added to the same plot list (to produce only the combined plot and not those for the individual contributions, it is sufficient to not add the latter to the plot list), e.g.

```
from bamboo.plots import Plot, SummedPlot, EquidistantBinning
mjj_mumu = Plot.make1D("Mjj_MuMu", op.invariant_mass(jets[0].p4, jets[1].p4),
                       sel_mumu, EquidistantBinning(50, 20., 120.))
mjj_elel = Plot.make1D("Mjj_ElEl", op.invariant_mass(jets[0].p4, jets[1].p4),
                       sel_elel, EquidistantBinning(50, 20., 120.))
mjj_sum = SummedPlot("Mjj", [mjj_mumu, mjj_elel], title="m(jj)")
plots += [ mjj_mumu, mjj_elel, mjj_sum ] # produce all plots
```

The other plot properties of a *SummedPlot* (titles, labels etc.) can be specified with keyword arguments to the constructor; otherwise they are taken from the first component plot.

> **Note**
>
> *SummedPlot* simply adds up the histograms, it is up to the user to make sure an event can only enter one of the categories, if this is what it is used for.

2.4.14 Producing skimmed trees

The *bamboo.plots.Skim* class allows to define skimmed trees to save in the output file. Since this uses the `Snapshot` method from RDataFrame, there will be an entry for each event that passes the selection, so in some cases (e.g. MVA training) additional manipulations may need to be done on these outputs. A second limitation is that, as for plots, a skim is attached to a selection, which means that if different categories need to be combined, multiple skims should be defined, and the stored products merged—but multiple skims can now be produced in the same job, thanks to the lazy Snapshot calls. The main advantage over the *SkimmerModule* (which still exists for backwards compatibility) is that the same module can produce plots and skims, with the same selections and definitions (in practice a *command-line option* would usually be added to select some products), e.g.

```
from bamboo.plots import Plot, Skim

twoMuSel = noSel.refine("twoMuons", cut=[ op.rng_len(muons) > 1 ])
mll = op.invariant_mass(muons[0].p4, muons[1].p4)
if self.args.makeSkim:
    plots.append(Skim("dimuSkim", {
```

(continues on next page)

(continued from previous page)

```
        "run": None,  # copy from input file
        "luminosityBlock": None,
        "event": None,
        "dimu_M": mll,
        "mu1_pt": muons[0].pt,
        "mu2_pt": muons[1].pt,
        }, twoMuSel))
else:
    plots.append(Plot.make1D("dimu_M", mll, twoMuSel,
                 EquidistantBinning(100, 20., 120.)))
```

2.4.15 Postprocessing beyond plotIt

The *HistogramsModule* postprocessing method calls plotIt to make the usual data and simulation stack plots (for the different eras that are considered), and prints the counter values of cut flow reports, but since all possible (meta-)information is available there, as well as the filled histograms, it can be useful to do any further processing there (e.g. running fits to the distributions, dividing histograms to obtain scale factors or fake rates, exporting counts and histograms to a different format).

For many simple cases, it should be sufficient to override the *postProcess()* method, first call the base class method, and then do any additional processing. If the base class method is not called, the plot list should be constructed by calling the *getPlotList()* method.

Most of the other code, e.g. to generate the plotIt YAML configuration file, is factored out in helper methods to allow reuse from user-defined additions—see the *bamboo.analysisutils.writePlotIt()* and *bamboo.analysisutils.printCutFlowReports()* methods, and their implementation.

Note

getPlotList(), when called without a specified file and sample, will read a so-called skeleton file *for an arbitrary sample* (essentially an empty tree with the same format as the input—typically for the first sample encountered) from the results directory and calls the *definePlots()* method with that to obtain the list of defined plots. This is also done when running with the `--onlypost` option, and works as expected when the same plots are defined for all samples. If this assumption does not hold, some customisation of the *definePlots()* method will be necessary.

It is also possible to skip the writing of a plotIt YAML file, and directly load the configuration as it would be parsed by plotIt with its partial python reimplementation pyplotit, which makes it easy to access the scaled grouped and stacked histograms.

As an example, a simple visualisation of 2D histograms could be obtained with

```
def postProcess(self, taskList, config=None, workdir=None, resultsdir=None):
    super(MyModule, self).postProcess(taskList, config=config, workdir=workdir,␣
↪resultsdir=resultsdir)
    from bamboo.plots import Plot, DerivedPlot
    plotList_2D = [ ap for ap in self.plotList if ( isinstance(ap, Plot) or␣
↪isinstance(ap, DerivedPlot) ) and len(ap.binnings) == 2 ]
    from bamboo.analysisutils import loadPlotIt
    p_config, samples, plots_2D, systematics, legend = loadPlotIt(config, plotList_2D,␣
↪eras=self.args.eras[1], workdir=workdir, resultsdir=resultsdir, readCounters=self.
↪readCounters, vetoFileAttributes=self.__class__.CustomSampleAttributes,␣
↪plotDefaults=self.plotDefaults)
```

(continues on next page)

(continued from previous page)

```
    from plotit.plotit import Stack
    from bamboo.root import gbl
    for plot in plots_2D:
        obsStack = Stack(smp.getHist(plot) for smp in samples if smp.cfg.type == "DATA")
        expStack = Stack(smp.getHist(plot) for smp in samples if smp.cfg.type == "MC")
        cv = gbl.TCanvas(f"c{plot.name}")
        cv.Divide(2)
        cv.cd(1)
        expStack.obj.Draw("COLZ")
        cv.cd(2)
        obsStack.obj.Draw("COLZ")
        cv.Update()
        cv.SaveAs(os.path.join(resultsdir, f"{plot.name}.png"))
```

2.4.16 Data-driven backgrounds and subprocesses

In many analyses, some backgrounds are estimated from a data control region, with some per-event weight that depends on the physics objects found etc. This can be largely automated: besides the main *Selection*, one or more instances with alternative cuts (the control region instead of the signal region) and weights (the mis-ID, fake, or transfer factors). That is exactly what is done by the *SelectionWithDataDriven* class: its *create()* method is like *bamboo.plots.Selection.refine()*, but with alternative cuts and weights to construct the correctly reweighted control region besides the signal region. Since it supports the same interface as *Selection*, further selections can be applied to both regions at the same time, and every *Plot* will declare the histograms for both. The additional code for configuring which data-driven contributions to use, and to make sure that histograms for backgrounds end up in a separate file (such that they can transparently be used e.g. in plotIt), the analysis module should inherit from *DataDrivenBackgroundHistogramsModule* (or *DataDrivenBackgroundAnalysisModule* if the histogram-specific functionality is not required).

Data-driven contributions should be declared in the YAML configuration file with the lists of samples or groups from which the background estimate should be obtained, those that are replaced by it, e.g.

```
datadriven:
  chargeMisID:
    uses: [ data ]
    replaces: [ DY ]
  nonprompt:
    uses: [ data ]
    replaces: [ TTbar ]
```

The `--datadriven` command-line argument can then be used to specify which of these should be used (`all`, `none`, or an explicit list). Several can be specified in the same run: different sets will then be produced. The parsed versions are available as the `datadrivenScenarios` attribute of the module (and the contributions as `datadrivenContributions`). The third argument passed to the *create()* method should correspond to one of the contribution names in the YAML file, e.g. (continuing the example above):

```
hasSameSignElEl = SelectionWithDataDriven.create(hasElElZ, "hasSSDiElZ", "chargeMisID",
    cut=(diel[0].Charge == diel[1].Charge),
    ddCut=(diel[0].Charge != diel[1].Charge),
    ddWeight=p_chargeMisID(diel[0])+p_chargeMisID(diel[1]),
    enable=any("chargeMisID" in self.datadrivenContributions and self.
↪datadrivenContributions["chargeMisID"].usesSample(sample, sampleCfg))
    )
```

The generation of modified sample configuration dictionaries in the plotIt YAML file can be customised by replacing the corresponding entry in the `datadrivenContributions` dictionary with a variation of a *`DataDrivenContribution`* instance.

A very similar problem is the splitting of a sample into different contributions based on some generator-level quantities, e.g. the number of (heavy-flavour) partons in the matrix element. In this case, splitting the RDF graph early on, such that each event is processed by a nearly identical branch of it, would not be very efficient. The *`bamboo.plots.LateSplittingSelection`* class, a variation of *`bamboo.plots.SelectionWithDataDriven`*, may help in such cases: it will branch the RDF graph only when attaching plots to a selection, so it can be constructed earlier, but the RDF graph branching will be minimal. By default the combined plot is also saved because it helps avoid duplication in the graph, but this may be disabled by passing `keepInclusive=False` to the *`create()`* method. To make sure the resulting histograms are saved, an analysis module that makes use of *`SelectionWithDataDriven`* should inherit from `bamboo.analysismodules.HistogramsModuleWithSub`; since the use case is rather specific, no customisation to the postprocessing method is done, but in most cases it should be straightforward to manipulate the `samples` dictionary in the configuration before calling the superclass' postprocessing method, see e.g. *this recipe*.

2.4.17 Dealing with (failed) batch jobs

When splitting the work over a set of batch jobs using the `--distributed=driver` option (see the *bambooRun options* reference), some may fail for various reasons: CPU time or memory limits that are too tight, environment or hardware issues on the worker node, or bugs in the analysis or bamboo code. The monitoring loop will check the status of the running jobs every two minutes, print information when some fail, merge outputs if all jobs for a sample complete, and finally run the postprocessing when all samples are processed, or exit when no running jobs remain. Currently (improvements and additions are being discussed in issue #87) resubmission of the failed jobs and monitoring of the recovery jobs, after identifying the reason why they fail, needs to be done using the tools provided by the batch system (`sbatch --array=X,Y,Z ...` for slurm; for HTCondor a helper script `bambooHTCondorResubmit` is provided that takes a very similar set of options—the commands are also printed by the monitoring loop).

When the outputs for all jobs that initially failed have been produced, `bambooRun` can be used with the `--distributed=finalize` option (and otherwise all the same options as for the original submission) to do any remaining merging of outputs, and run the postprocessing step. If some outputs are missing it will suggest a resubmission command and exit. This only looks at the output files that are found to decide what still needs to be done, so if a file in the `results/` subdirectory of the output is present, it will assume that is valid—this can be exploited in two ways: if anything goes wrong in the merging, removing the `results/<<sample>>.root` and running with `--distributed=finalize` will try that again (similarly, removing a corrupt job output file will add it to the resubmission command), and if a sample is processed with a different splitting it is sufficient to put the merged output file in the `results/` directory.

Note

Understanding why batch jobs fail is not always easy, and the specifics depend on the batch system and the environment Bamboo collects all possible log files (standard output and error, submission log) in the `batch/logs` directory, and per-job inputs and output in `batch/input` and `batch/output`, respectively.

In principle the worker jobs run in the same environment as where they are submitted, and typically take all software is installed from CVMFS, so most problems with batch jobs are related to data access, e.g. overloaded storage or permissions to access some resources. When reading files through XRootD a grid proxy is needed, at CERN the easiest is to create it in an AFS directory and pass that to the job.

2.4.18 Reproducible analysis: keep track of the version that produced some results

While bamboo does not by default force you to adopt a specific workflow, it can help with adopting some best practices for reproducible analysis. The most important thing is to keep the analysis code under version control: git is widely used for this (see the Pro Git book for an introduction). The idea is to keep the analysis code and configurations in a

separate directory, which is tracked by git, from the bamboo outputs (plots, results etc.)—this can also be a subdirectory that is ignored by git, if you prefer.

`bambooRun` will write a file with the git version of the repository where the module and configuration file are found to the output directory: the `version.yml` file. This will also contain the list of command-line arguments that were passed, and the bamboo version. In order for this to work, the analysis repository must at least have all local changes committed, but it is even better to create a tag for versions that are used to produce results, and push it to GitHub or GitLab (see e.g. this overview; it is also worth noting that tags in git can be *annotated*, which means that they can have a descriptive message, just like a commit). If the `--git-policy` switch, or the `policy` key in the `git` section in the `~/.config/bamboorc` file, gets a different value than the default (`testing`), `bambooRun` will check that the analysis code is committed, tagged, or tagged and pushed, based on the specified value (`committed`, `tagged`, and `pushed`, respectively). It is recommended to use at least `committed` (which will print warnings if the commit has not been pushed, or is not tagged).

Tip: use git worktrees

An interesting solution to have several checkouts of the same repository, e.g. to run jobs with one version of the analysis code, and edit it at the same time, are git worktrees (see git-worktree manual page for a reference, or this article for some examples). They may also help with making sure that everything is committed and tracked by git: if you use the main clone to edit the code, and checkout a commit or tag in a worktree to produce plots on the full dataset, committing all necessary files is the best way to keep them synchronized (the "production" directory should not contain any untracked files then).

Git worktrees were introduced in version 2.5, so it will not work with older versions. The LCG distribution includes git since LCG_99, so if you use that method of installing bamboo it will be included automatically.

Tip: make a python package out of your analysis

For small analyses and projects, all that is needed are a YAML configuration file and a python module, or a few of each. When code needs to be shared between different modules, a simple solution is to make it a python package: move the shared modules to a subdirectory, called e.g. `myanalysis`, add an empty `__init__.py` to it, and write a `setup.py` file (still required for editable installs) like this one:

```
from setuptools import setup, find_packages

setup(
    name="myexperiment-myanalysis",
    description="Hunt for new physics (implemented with bamboo)",
    url="https://gitlab.cern.ch/.../...",
    author="...",
    author_email="...",

    packages=find_packages("."),

    setup_requires=["setuptools_scm"],
    use_scm_version=True
)
```

It can then be installed in the virtual environment with

```
pip install -e .
```

and the shared modules imported as `myanalysis.mymodule`. The `-e` flag actually puts only a link in the virtual environment, such that any changes in the source files are immediately available, without updating the installed version (then it does not spoil the change tracking above).

More information on packaging python packages can be found in the PyPA packaging tutorial, the setuptools documentation, the PyPA setuptools guide and the Scikit-HEP packaging guidelines. For packages that include C++ components scikit-build allows to combine setuptools and CMake (it is also used by bamboo and correctionlib).

2.4.19 SVfit for the reconstruction of the Higgs mass in $H \to \tau\tau$ events

The Higgs mass in events with Higgs bosons decaying into a pair of τ leptons can be reconstructed using the SVfit algorithm. The algorithm is based on matrix element techniques and typically achieves a relative resolution on the Higgs boson mass of 15–20%. It utilizes information about the missing transverse energy and the kinematic properties of leptons as input.

The SVfit algorithm is not implemented within bamboo framework itself, but an interface is provided to enable users to use SVfit in bamboo.

Firstly, SVfit needs to be installed (outside of the bamboo package).

```
git clone git@github.com:cp3-llbb/ClassicSVfit.git ClassicSVfit -b fastMTT_cmake_build
mkdir build-ClassicSVfit-FastMTT/
cd build-ClassicSVfit-FastMTT/
cmake cmake -DCMAKE_INSTALL_PREFIX=$VIRTUAL_ENV ../ClassicSVfit/
make -j2
```

After installation, the on the fly calculator should be added and configured using the *bamboo.analysisutils.configureSVfitCalculator()* method, as shown in the example below. Users can then utilize the *bamboo.treefunctions.svFitMTT()* and *bamboo.treefunctions.svFitFastMTT()* functions for the reconstruction of the Higgs mass.

```
class MyAnalysisModule(NanoAODHistoModule):
    def prepareTree(self, tree, sample=None, sampleCfg=None):
        tree,noSel,be,lumiArgs = NanoAODHistoModule.prepareTree(self, tree,␣
↪sample=sample, sampleCfg=sampleCfg)
        from bamboo.analysisutils import configureSVfitCalculator
        configureSVfitCalculator(pathToSVfit=" ", backend=be)
        return tree,noSel,be,lumiArgs
```

2.5 Advanced topics

2.5.1 Loading (and using) C++ modules

The *bamboo.root* module defines a few thin wrappers to load additional libraries or headers with the ROOT interpreter (PyROOT makes them directly accessible through the global namespace, which can be imported as `gbl` from *bamboo.root*. The *bamboo.root.loadDependency()* allows to load a combination of headers and libraries; it is best called through the backend's `addDependency()` method, which will also correctly register all components for the compilation of standalone executables, when using the compiled backend (see below). As an example, the library that contains the dictionaries for the classes used in Delphes output trees can be added as follows:

```
be.addDependency(libraries="Delphes")
```

In this specific case, `bamboo.root.addLibrary("libDelphes")` should also be added to the analysis module's *initialize()* method, to make sure the library is added before any file is opened. For a module that is used in calculating expressions, it is sufficient to load it only from the *prepareTree()* or `processTrees()` method (repeated loads should not cause problems).

C++ methods can be used directly from an expression. Non-member methods that are known by the interpreter (e.g.

because the corresponding header has been added with *bamboo.root.loadHeader()*), can be retrieved with *bamboo.treefunctions.extMethod()*, which returns a decorated version of the method.

It is often useful to define a class that stores some parameters, and then call a member method with event quantities to obtain a derived quantity (this is also the mechanism used for most of the built-in corrections). In order to use such a class, its header (and shared library, if necessary) should be loaded as above, and an instance defined with *bamboo.treefunctions.define()*, e.g.

```
myCalc = op.define("MyCalculatorType", 'const auto <<name>> = MyCalculatorType("test");')
myCorrection = myCalc.evaluate(tree.Muon[0].pt, tree.Muon[1].pt)
```

Warning

With implicit multi-threading enabled, only thread-safe methods may be called in this way (e.g. const member methods, without global or member variables used for caching).

Note

The usual logic to avoid redefinition of these variables is applied. In cases like above where all parameters are supplied at once, this will work as expected. If the calculator is further configured by calling member methods (it can be accessed directly through PyROOT), it is safer to create a unique instance for each sample, e.g. by adding a comment that contains the sample name at the end of the declaration (an optional `nameHint` argument can be given to make the generated code more readable, but this will be ignored in case the declaration string is the same).

2.5.2 Distributed RDataFrame

This adds support for DistRDF, to distribute the computations dynamically to a cluster without having to manage the jobs in bamboo manually. The backend handling the distribution is either Dask (distributed) (typically with dask-jobqueue) or Spark. Through dask-jobqueue, "regular" batch systems based on Slurm or HTCondor are supported. The long-term goal is to deprecate the batch job management (`driver` mode) in Bamboo entirely, and rely only on DistRDF. This section describes how to configure bamboo to use these systems efficiently.

Warning

These features should still be considered experimental. More feedback is welcome on how to fine-tune these systems in practice.

To use DistRDF, call `bambooRun` with the additional `--distrdf-be <BE>` argument, where `<BE>` can be one of:

- `dask_local`: local-machine multiprocessing, useful for testing but functionally equivalent to *–threads*
- `dask_scheduler`: connect to an existing Dask scheduler for your cluster
- `dask_slurm` or `dask_condor`: Dask handles job submission to a Slurm or HTCondor cluster
- `spark`: connect to an existing Spark cluster (or spawn a locally running cluster)

DistRDF works with both default (sequential) processing mode, where one sample is processed after the other, and with `--distributed parallel`, where the graphs for all samples are first built, before processing all samples in parallel. Obviously, `--distributed driver` does not make sense here.

To configure the distributed backend, add a corresponding section in the environment file, e.g.:

```
[dask_slurm]
adapt_max = 100 ; submit max. 100 jobs
```

Have a look at the example at examples/distributed.ini for more configuration options.

For dask-jobqueue, you will in addition need to configure the submission to the Slurm/HTCondor job scheduler using a *jobqueue.yml* file placed into *~/.config/dask/*. Have a look at the documentation of dask-jobqueue and at the example at examples/dask_jobqueue.yaml.

2.5.3 Ordering selections and plots efficiently

Internally, Bamboo uses the RDataFrame class to process the input samples and produce histograms or skimmed trees—in fact no python code is run while looping over the events: Bamboo builds up a computation graph when *Selection* and *Plot* objects are defined by the analysis module's *definePlots()* method, RDataFrame compiles the expressions for the cuts and variables, and the input files and events are only looped over once, when the histograms are retrieved and stored.

In practice, however, there are not only `Filter` (*Selection*) and `Fill` (*Plot*) nodes in the computation graph, but also `Define` nodes that calculate a quantity based on other columns and make the result available for downstream nodes to use directly. This is computationally more efficient if the calculation is complex enough. Bamboo tries to make a good guess at which (sub-)expressions are worth pre-calculating (typically operations that require looping over a collection), but the order in which plots and selections are defined may still help to avoid inserting the same operation twice in the computation graph.

The main feature to be aware of is that RDataFrame makes a node in the computation graph for every `Define` operation, and the defined column can only be used from nodes downstream of that. Logically, however, all defined columns needed for plots or sub-selections of one selection will need to be evaluated for all events passing this selection, and the most efficient is to do this only once, so ideally all definitions should be inserted right after the `Filter` operation of the selection, and before any of the `Fill` and subsequent `Filter` nodes. This is a bit of a simplification: it is possible to imagine cases where it can be better to have a column only defined for the sub-nodes that actually use it, but then it is hard to know in all possible cases where exactly to insert the definitions, so the current implementation opts for a simple and predictable solution: on-demand definitions of sub-expressions are done when *Plot* and *Selection* objects are constructed, and they update the computation graph node that other nodes that derive from the same selection will be based on. A direct consequence of this is that it is usually more efficient to first define plots for a stage of the selection, and only then define refined selections based on it—otherwise the sub-selection will be based on the node without the columns defined for the plots and, in the common case where the same plots are made at different stages of the selection, recreate nodes with the same definitions in its branch of the graph. As an illustration, the pseudocode equivalent of these two cases is

```
## define first sub-selection then plots
## some_calculation(other_columns) is done twice
if selectionA:
    if selectionB:
       myColumn1 = some_calculation(other_columns)
       myPlot1B = makePlot(myColumn1)
    myColumn2 = some_calculation(other_columns)
    myPlot1A = makePlot(myColumn2)

## define first plots then sub-selection
## some_calculation(other_columns) is only done once
if selectionA:

    myColumn1 = some_calculation(other_columns)
    myPlot1A = makePlot(myColumn1)
```

(continues on next page)

(continued from previous page)

```
    if selectionB:
       myPlot1B = makePlot(myColumn1)
```

This is why it is advisable to reserve the *definePlots()* method of the analysis module for defining event and object container selections, and define helper methods that declare the plots for a given selection—with a parameter that is inserted in the plot name to make sure they are unique, if used to define the same plots for different selection stages, e.g.

```
def makeDileptonPlots(self, sel, leptons, uname):
    from bamboo.plots import Plot, EquidistantBinning
    from bamboo import treefunctions as op
    plots = [
         Plot.make1D("{0}_llM".format(uname),
            op.invariant_mass(leptons[0].p4, leptons[1].p4), sel,
            EquidistantBinning(100, 20., 120.),
            title="Dilepton invariant mass",
            plotopts={"show-overflow":False}
            )
    ]
    return plots

def definePlots(self, t, noSel, sample=None, sampleCfg=None):
    from bamboo import treefunctions as op

    plots = []

    muons = op.select(t.Muon, lambda mu : op.AND(mu.p4.Pt() > 20., op.abs(mu.p4.Eta() <␣
↪2.4)))

    twoMuSel = noSel.refine("twoMuons", cut=[ op.rng_len(muons) > 1 ])

    plots += self.makeDileptonPlots(twoMuSel, muons, "DiMu")

    jets = op.select(t.Jet, lambda j : j.p4.Pt() > 30.)

    twoMuTwoJetSel = twoMuSel.refine("twoMuonsTwoJets", cut=[ op.rng_len(jets) > 1 ])

    plots += self.makeDileptonPlots(twoMuTwoJetSel, muons, "DiMu2j")

    return plots
```

Finally, there are some cases where the safest is to force the inclusion of a calculation at a certain stage, for instance when performing expensive function calls, since the default strategy is not to precalculate these because there are many more function calls that are not costly. A prime example of this is the calculation of modified jet collections with e.g. an alternative JEC applied, which is done in a separate C++ module (see below), and is probably the slowest operation in most analysis tasks. The definition can be added explicitly under a selection by calling the *bamboo.analysisutils.forceDefine()* method, e.g. with

```
for calcProd in t._Jet.calcProds:
    forceDefine(calcProd, mySelection)
```

2.5.4 Different backends

Different approaches for converting plots and selections to an RDataFrame graph have been implemented, each with some advantages and disadvantages. The `--backend` option of the `bambooRun` command allows to select one. The default approach is to generate the necessary helper methods and define JITted nodes whenever a plot or selection is added. This has the advantage that if there is a problem that can be detected at this stage, it is easy to trace back.

The lazy backend (`--backend==lazy`) instead waits for all plots and selections to be defined before constructing the graph. In principle, that ensures the order from the previous section, so depending on the case it will generate a more efficient graph. This uses the same JITted `Define` and `Filter` nodes as the default.

The experimental compiled backend takes things a step further, by generating the full C++ source for a standalone executable, essentially eliminating the need for any type inference at runtime. The structure of the graph will be identical to the one from the lazy backend, but with compiled instead of JITted nodes.

Note

Event processing with such an executable will be faster—in many cases by a significant amount—but the time needed for compilation, and especially the memory usage during compilation and linking, may be non-negligible. A few dynamic features (e.g. progress printing) are currently disabled as well. More efficiently using the compiled backend is closely tied to reusing the same executable across batch jobs and for different samples, and compiling different executables in parallel. These are planned, but they need some restructuring of the code, which will be done in steps as they will also allow for some other new features to be added, e.g. only producing a subset of the histograms, and using distributed RDF.

The compiled backend takes a few additional options, which can be passed by setting its attributes (a reference to the backend should be returned by the *`prepareTree()`* method). These are:

- `cmakeConfigOptions`: options passed to CMake when configuring the build, in addition to the path and the variables needed to pick up the bamboo C++ extensions (default: `["-DCMAKE_BUILD_TYPE=Release"]`, which implies `-O3`)
- `cmakeBuildOptions`: options passed to `cmake --build` when compiling the executable

In case anything goes wrong during the configuration or build, the temporary directory will be saved, and the errors and path printed in the log file.

2.6 Under the hood

This page collects some useful information for debugging and developing bamboo.

2.6.1 Debugging problems

Despite a number of internal checks, bamboo may not work correctly in some cases. If you encounter a problem, the following list of tips may help to find some clues about what is going wrong:

- does the error message (python exception, or ROOT printout) provide any hint? (for batch jobs: check the log file, its name should be printed)
- try to rerun on (one of) the offending sample(s), with debug printout turned on (by passing the `-v` or `--verbose` option, for failed batch jobs the main program prints the command to reproduce)
- if the problem occurs only for one or some samples: is there anything special in the analysis module for this sample, or in its tree format? The interactive mode to explore the decorated tree can be very useful to understand problems with expressions.

- in case of a segmentation violation while processing the events: check if you are not accessing any items from a container that are not guaranteed to exist (i.e. if you plot properties of the 2nd highest-pt jet in the event, the event selection should require at least two jets; with combinations or selections of containers this may not always be easy to find). The *bamboo.analysisutils.addPrintout()* function may help to insert printout statements in the RDataFrame graph, see its description for an example.
- check the open issues to see if your problem has already been reported, or is a known limitation, and, if not, ask for help on mattermost or directly create a new issue

2.6.2 Different components and their interactions

Expressions: proxies and operations

The code to define expressions (cuts, weight factors, variables) in bamboo is designed to look as if the per-event variables (or columns of the RDataFrame) are manipulated directly, but what actually happens is that a python object representation is constructed. The classes used for this are defined in the `bamboo.treeoperations` module, and inherit from `TupleOp`. There are currently about 25 concrete implementations.

These classes contain the minimal needed information to obtain the value they represent (e.g. names of columns to retrieve, methods to call), but generally no complete type information or convenience methods to use them. They are used by almost all other bamboo code, but not meant to be directly manipulated by the user code—this is what the proxy classes are for.

The main restriction on `TupleOp` classes is that, once constructed, the operation part of an expression should not be modified. More specifically: not after they have been passed to any backend code (so directly after construction, e.g. by cloning, should be safe, but since sub-expressions may be passed on-demand, one should not make any assumptions in other cases). This allows to cache the hash of an operation, and thus very fast lookup of expressions in sets and dictionaries, which the backend uses extensively.

The proxy classes wrap one or more operations, and behave as the resulting value. In some cases the correspondence is trivial, e.g. a branch with a single floating-point number is retrieved with a `GetColumn` operation, and wrapped with a `FloatProxy`, which overloads operators for basic math, but a proxy can also represent an object or concept that does not correspond to a C++ type stored on the input tree, e.g. an electron (the collection of values with the same index in all `Electron_*[nElectron]` branches), or a subset of the collection of electrons, whose associated operation would be a list of indices, with the proxy holding a reference to the original collection proxy.

All proxy classes (currently about 25) are defined in the `bamboo.treeproxies` module, and inherit from the `TupleBaseProxy` base class, which means they need to have an associated type, and hold a reference to a parent operation. Operations only refer to other operations and constants, not to proxies, so when an action (overloaded operator, member method, or a function from *bamboo.treefunctions*) is performed on a proxy, a new proxy is returned that wraps the resulting operation.

In principle proxies are only there for the user code: starting from the input tree proxy, expressions are generated and passed to the backend, which strips off the proxy, and generates executable code from the operation (possibly retaining the result type from the proxy, if relevant for the produced output, e.g. when producing a skimmed tree). There are therefore few constraints on how the proxy classes work, as long as the result of any action on them produces a valid operation with the expected meaning.

Tree decorations

All user-defined expressions start from the decorated input tree, which can, following the previous subsection, be seen as a tree proxy. In fact, this is exactly what the tree decoration method does: it generates the necessary ad-hoc types that inherit from the building block proxy classes from `bamboo.treeproxies`, and also have all the attributes corresponding to the branches of the input tree. Technically, this is done with the type builtin, and a few descriptor classes.

Much of the information needed for this can be obtained by introspecting the tree, but some details, e.g. about systematics to enable, may need to be supplied by the user.

Selections, plots, and the RDataFrame

The main thing to know about the RDataFrame in bamboo is that partial results are declared upon construction of *Plot* and *Selection* objects. The backend keeps a shadow graph of selections (with their alternatives under systematic variations, if needed), and, for each of these, a list of the operations that have been defined as a new column.

When an operation is converted to a C++ expression string, a reference to the selection node where it is needed is passed, such that sub-expressions can be defined on-demand (as explained in *this section*, if a precalculated column is needed for a selection, it may be beneficial to declare that earlier rather than later). This makes the verbose output a bit harder to read (to avoid re-declaring the same function, argument names are also replaced), but ensures the correct order of definition and reasonable efficiency. Currently, all operations that take range arguments, and those that are explicitly marked, are precalculated. Function calls, notably, are not, since most are cheap to evaluate—this is why expensive function calls sometimes should be explicitly requested to be precalculated for a specific selection with *bamboo.analysisutils.forceDefine()*.

Organisationally, the bookkeeping code, and all the code that accesses the interpreter and RDataFrame directly, is kept in `bamboo.dataframebackend`, while the conversion of a `TupleOp` is done by its `get_cppStr()` method (many of these are trivial, but for range-based operations, which define a helper function, they get a bit more involved).

2.6.3 Running the tests, or adding test cases

The test suite consists of two parts: the standard tests, which are run for every opened merge request, and push to a pull request or the master branch, and a set of regression tests that perform a bin-by-bin comparison of the histograms produced with a simple plotter over a small dataset. The former are closer to unit tests, and limited integration tests, so they check test some components in isolation, and sequences of basic operations, like constructing a few *Selection* and *Plot* objects.

All the tests can easily be run with pytest, the standard tests with

```
pytest <bamboo_clone>/tests
```

and the additional regression tests with

```
pytest <bamboo_clone>/tests/test_plotswithreference.py --plots-reference=/home/ucl/cp3/
↪pdavid/scratch/bamboo_test_reference
```

where the directory above is one set of reference histograms at the UCLouvain T2 grid site; details on producing such a set are given below. These are not fully integrated with GitLab CI yet because they require access to CMS NanoAOD files. More generally, passing a specific file to pytest will make it run only the tests defined in that file.

> **Note**
>
> Tests are not only useful when developing new code. They can also be very helpful in understanding some unexpected or buggy behaviour, and pytest makes it very easy to run the tests, and add more: just add a method starting with `test_` in one of the test files, with an assertion to check if the tests pass, or add a file with a name starting with `test_` to the `tests` directory and define your test cases there. Contributing tests is one of the easiest ways to get to know the internals and help with bamboo development, so additional tests are always welcome.

The regression tests will by default use a temporary directory, so the output is automatically removed when the test run finishes. This can be changed by passing a directory to the `--plots-output` argument. To turn such an output directory into a new reference directory, two files should be added, `test_zmm_ondemand.yml` and `test_zmm_postproc.yml`, which are the configuration files for the on-demand and post-processed runs, respectively. In fact the only output files that are used are the histogram files in the respective `results` directories, so the rest of the output directories can, but needs not, be removed.

The T2_BE_UCL test configs use a single file of data, DoubleMuon for 2016 and DoubleEG for 2017, and 100k events from a Drell-Yan simulation sample for each of the two years, but any similar configuration should work. The post-processing must add the full set of jet and MET kinematic variations.

The bin-by-bin comparison may also be useful for other contexts, so it is made available as a command-line script in `<bamboo_clone>/tests/diffHistsAndFiles.py`. Full documentation is available through the `--help` command, but generally it takes two directories with histograms, and will compare all histograms in ROOT files present in both (if some ROOT files are present in one but not the other directory, that will also be considered a failure).

2.7 API Reference

This page lists the classes and methods that are necessary for building an analysis, but are *not* related to expressions (see *Building expressions* for a description of those)—the aim is to provide a set of easy-to-use classes and methods.

2.7.1 Plots and selections

The *bamboo.plots* module provides high-level classes to represent and manipulate selections and plots.

class bamboo.plots.**CategorizedSelection**(*parent=None*, *categories=None*, *name=None*)

Helper class to represent a group of similar selections on different categories

The interface is similar, but not identical to that of *Selection* (constructing *Plot* objects is done through the *makePlots()* method, which takes additional arguments). Each category selection can have a candidate, typically the object or group of object that differs between the categories. The axis variables can then either be expressions, or callables that will be passed this per-category object.

Example

```
>>> muonSel = noSel.refine("hasMuon", cut=(
>>>     op.rng_len(muons) > 0, op.OR(op.rng_len(electrons) == 0,
>>>     muons[0].pt > electrons[0].pt)))
>>> electronSel = noSel.refine("hasElectron", cut=(
>>>     op.rng_len(electrons) > 0, op.OR(op.rng_len(muons) == 0,
>>>     electrons[0].pt > muons[0].pt)))
>>> oneLeptonSel = CategorizedSelection(categories={
...     "Mu" : (muonSel, muons[0]),
...     "El" : (electronSel, electrons[0])
...     })
>>> oneLep2JSel = onLeptonSel.refine("hasLep2J", cut=(op.rng_len(jets) >= 2))
>>> plots += oneLep2JSel.makePlots("J1PT", jets[0].pt, EqB(50, 0., 150.))
>>> plots += oneLep2JSel.makePlots("LJ1Mass",
...     (lambda l : op.invariant_mass(jets[0].p4, l.p4)), EqB(50, 0., 200.))
```

__init__(*parent=None*, *categories=None*, *name=None*)

Construct a group of related selections

Parameters

- **name** – name (optional)
- **parent** – parent CategorizedSelection (optional)
- **categories** – dictionary of a *Selection* and candidate (any python object) per category (key is category name), see the *addCategory()* method below

addCategory(*catName*, *selection*, *candidate=None*)

Add a category

Parameters

- **catName** – category name
- **selection** – *Selection* for this category
- **candidate** – any python object with event-level quantities specific to this category

makePlots(*name*, *axisVariables*, *binnings*, *construct=None*, *savePerCategory=True*, *saveCombined=True*, *combinedPlotType=<class 'bamboo.plots.SummedPlot'>*, ***kwargs*)

Make a plot for all categories, and/or a combined one

Parameters

- **name** – plot name (per-category plot names will be `"{name}_{category}"`)
- **axisVariables** – one or more axis variables
- **binnings** – as many binnings as variables
- **construct** – plot factory method, by default the `make{N}D` method of *Plot* (with N the number of axis variables)
- **savePerCategory** – save the individual plots (enabled by default)
- **saveCombine** – save the combined plot (enabled by default)
- **combinedPlotType** – combined plot type, *SummedPlot* by default

Returns
a list of plots

refine(*name*, *cut=None*, *weight=None*, *autoSyst=True*)

Equivalent of *refine()*, but for all categories at a time

Parameters

- **name** – common part of the name for the new category selections (individual names will be `"{name}_{category}"`)
- **cut** – cut(s) to add. If callable, the category's candidate will be passed
- **weight** – weight(s) to add. If callable, the category's candidate will be passed
- **autoSyst** – automatically add systematic variations (True by default - set to False to turn off; note that this would also turn off automatic systematic variations for any selections and plots that derive from the one created by this method)

Returns
the new *CategorizedSelection*

class bamboo.plots.**CutFlowReport**(*name*, *selections=None*, *recursive=False*, *titles=None*, *autoSyst=False*, *cfres=None*, *printInLog=False*)

Collect and print yields at different selection stages, and cut efficiencies

The simplest way to use this, just to get an overview of the number of events passing each selection stage in the log file, is by adding a `CutFlowReport("yields", selections=<list of selections>, recursive=True, printInLog=True)` to the list of plots. `recursive=True` will add all parent selections recursively, so only the final selection categories need to be passed to the `selections` keyword argument.

It is also possible to output a LaTeX yields table, and specify exactly which selections and row or column headers are used. Then the *CutFlowReport* should be constructed like this:

```
yields = CutFlowReport("yields")
plots.append(yields)
yields.add(<selection1-or-list-of-selections1>, title=title1)
yields.add(<selection2-or-list-of-selections2>, title=title2)
...
```

Each `yields.add` call will then add one entry in the yields table, with the yield the one of the corresponding selection, or the sum over the list (e.g. different categories that should be taken together); the other dimension are the samples (or sample groups). The sample (group) titles and formatting of the table can be customised in the same way as in plotIt, see *printCutFlowReports()* for a detailed description of the different options.

__init__(*name*, *selections=None*, *recursive=False*, *titles=None*, *autoSyst=False*, *cfres=None*, *printInLog=False*)

Constructor. `name` is mandatory, all other are optional; for full control the *add()* should be used to add entries.

Using the constructor with a list of *Selection* instances passed to the `selections` keyword argument, and `recursive=True, printInLog=True` is the easiest way to get debugging printout of the numbers of passing events.

add(*selections*, *title=None*)

Add an entry to the yields table, with a title (optional)

produceResults(*bareResults*, *fbe*, *key=None*)

Main interface method, called by the backend

Parameters

- **bareResults** – iterable of histograms for this plot produced by the backend
- **fbe** – reference to the backend
- **key** – key under which the backend stores the results (if any)

Returns

an iterable with ROOT objects to save to the output file

readFromResults(*resultsFile*)

Reconstruct the *CutFlowReport*, reading counters from a results file

class bamboo.plots.**DerivedPlot**(*name*, *dependencies*, ***kwargs*)

Base class for a plot with results based on other plots' results

The `dependencies` attribute that lists the *Plot*-like objects this one depends on (which may be used e.g. to order operations). The other necessary properties (binnings, titles, labels, etc.) are taken from the keyword arguments to the constructor, or the first dependency. The *produceResults()* method, which is called by the backend to retrieve the derived results, should be overridden with the desired calculation.

Typical use cases are summed histograms, background subtraction, etc. (the results are combined for different subjobs with hadd, so derived quantities that require the full statistics should be calculated from the postprocessing step; alternative or additional systematic variations calculated from the existing ones can be added by subclassing *Plot*).

__init__(*name*, *dependencies*, ***kwargs*)

collectDependencyResults(*fbe*, *key=None*)

helper method: collect all results of the dependencies

Returns

`[ (nominalResult, {"variation" : variationResult}) ]`

produceResults(*bareResults*, *fbe*, *key=None*)

Main interface method, called by the backend

Parameters

- **bareResults** – iterable of histograms for this plot produced by the backend (none)
- **fbe** – reference to the backend, can be used to retrieve the histograms for the dependencies, e.g. with *collectDependencyResults()*
- **key** – key under which the backend stores the results (if any)

Returns

an iterable with ROOT objects to save to the output file

class bamboo.plots.**EquidistantBinning**(*N*, *mn*, *mx*)

Equidistant binning

__init__(*N*, *mn*, *mx*)

Parameters

- **N** – number of bins
- **mn** – minimum axis value
- **mx** – maximum axis value

class bamboo.plots.**FactoryBackend**

Interface for factory backend (to separate Plots classes from ROOT::RDataFrame part)

__init__()

buildGraph(*plotList*)

Called after adding all products, but before retrieving the results

classmethod create(*tree*, *nThreads=None*)

Factory method, should return a pair of the backend and root selection

define(*op*, *selection*)

explicitly define column for expression

class bamboo.plots.**LateSplittingSelection**(*parent*, *name*, *cuts=None*, *weights=None*, *autoSyst=True*, *keepInclusive=None*)

A drop-in replacement for *Selection* to efficiently split a sample

The concept is quite similar to *SelectionWithDataDriven*, but with very different performance trade-offs: the former creates two parallel branches of the RDF graph, each for their own set of events (with a typically small performance overhead due to dupliation), whereas this is for cases where all events should be processed identically until they are filled into histograms (e.g. separating subprocesses based on MC truth). It is worth defining columns with these categories early on, such that the splitting does not need to do it many times for different selections and categories.

__init__(*parent*, *name*, *cuts=None*, *weights=None*, *autoSyst=True*, *keepInclusive=None*)

Constructor. Prefer using *refine()* instead

(except for the 'root' selection)

Parameters

- **parent** – backend or parent selection
- **name** – (unique) name of the selection

- **cuts** – iterable of selection criterion expressions (optional)
- **weights** – iterable of weight factors (optional)

static create(*parent*, *name*, *splitCuts=None*, *keepInclusive=True*, *cut=None*, *weight=None*, *autoSyst=True*)

Create a selection that will lazily split into categories

Parameters

- **name** – name of the new selection (after applying the cut and weight, as in *bamboo.plots.Selection.refine()*)
- **splitCuts** – dictionary of regions, the values should be the cuts that define the region
- **keepInclusive** – also produce the plots without splitting
- **cut** – common selection
- **weight** – common weight
- **autoSyst** – automatically propagate systematic uncertainties

initSub()

Initialize related selections, should be called before registering non-plot products
(anything not going through registerPlot)

class bamboo.plots.**Plot**(*name*, *variables*, *selection*, *binnings*, *weight=None*, *title=''*, *axisTitles=()*, *axisBinLabels=()*, *plotopts=None*, *autoSyst=True*, *key=None*)

A *Plot* object contains all information needed to produce a histogram: the variable(s) to plot, binnings and options (axis titles, optionally some style information), and a reference to a *Selection* (which holds all cuts and weights to apply for the plot).

Note

All *Plot* (and *Selection*) instances need to have a unique name. This name is used to construct output filenames, and internally to define DataFrame columns with readable names. The constructor will raise an exception if an existing name is used.

__init__(*name*, *variables*, *selection*, *binnings*, *weight=None*, *title=''*, *axisTitles=()*, *axisBinLabels=()*, *plotopts=None*, *autoSyst=True*, *key=None*)

Generic constructor. Please use the static *make1D()*, *make2D()* and *make3D()* methods, which provide a more convenient interface to construct histograms (filling in some defaults requires knowing the dimensionality).

clone(*name=None*, *variables=None*, *selection=None*, *binnings=None*, *weight=None*, *title=None*, *axisTitles=None*, *axisBinLabels=None*, *plotopts=None*, *autoSyst=True*, *key=None*)

Helper method: create a copy with optional re-setting of attributes

classmethod make1D(*name*, *variable*, *selection*, *binning*, ***kwargs*)

Construct a 1-dimensional histogram plot

Parameters

- **name** – unique plot name
- **variable** – x-axis variable expression
- **selection** – the *Selection* with cuts and weights to apply

- **binning** – x-axis binning
- **weight** – per-entry weight (optional, multiplied with the selection weight)
- **title** – plot title
- **xTitle** – x-axis title (optional, taken from plot title by default)
- **xBinLabels** – x-axis bin labels (optional)
- **plotopts** – dictionary of options to pass directly to plotIt (optional)
- **autoSyst** – automatically add systematic variations (True by default - set to False to turn off)

Returns
the new *Plot* instance with a 1-dimensional histogram

Example

```
>>> hasTwoEl = noSel.refine(cut=(op.rng_len(t.Electron) >= 2))
>>> mElElPlot = Plot.make1D(
>>>     "mElEl", op.invariant_mass(t.Electron[0].p4, t.Electron[1].p4),␣
↪hasTwoEl,
>>>     EquidistantBinning(80, 50., 130.), title="Invariant mass of the leading-
↪PT electrons")
```

classmethod make2D(*name*, *variables*, *selection*, *binnings*, ***kwargs*)

Construct a 2-dimensional histogram plot

Parameters

- **name** – unique plot name
- **variables** – x- and y-axis variable expression (iterable, e.g. tuple or list)
- **selection** – the *Selection* with cuts and weights to apply
- **binnings** – x- and y-axis binnings (iterable, e.g. tuple or list)
- **weight** – per-entry weight (optional, multiplied with the selection weight)
- **title** – plot title
- **xTitle** – x-axis title (optional, empty by default)
- **yTitle** – y-axis title (optional, empty by default)
- **xBinLabels** – x-axis bin labels (optional)
- **yBinLabels** – y-axis bin labels (optional)
- **plotopts** – dictionary of options to pass directly to plotIt (optional)
- **autoSyst** – automatically add systematic variations (True by default - set to False to turn off)

Returns
the new *Plot* instance with a 2-dimensional histogram

classmethod make3D(*name*, *variables*, *selection*, *binnings*, ***kwargs*)

Construct a 3-dimensional histogram

Parameters

- **name** – unique plot name

- **variables** – x-, y- and z-axis variable expression (iterable, e.g. tuple or list)
- **selection** – the *Selection* with cuts and weights to apply
- **binnings** – x-, y-, and z-axis binnings (iterable, e.g. tuple or list)
- **weight** – per-entry weight (optional, multiplied with the selection weight)
- **title** – plot title
- **xTitle** – x-axis title (optional, empty by default)
- **yTitle** – y-axis title (optional, empty by default)
- **zTitle** – z-axis title (optional, empty by default)
- **xBinLabels** – x-axis bin labels (optional)
- **yBinLabels** – y-axis bin labels (optional)
- **zBinLabels** – z-axis bin labels (optional)
- **plotopts** – dictionary of options to pass directly to plotIt (optional)
- **autoSyst** – automatically add systematic variations (True by default - set to False to turn off)

Returns
the new *Plot* instance with a 3-dimensional histogram

produceResults(*bareResults*, *fbe*, *key=None*)

Trivial implementation of *produceResults()*

Subclasses can e.g. calculate additional systematic variation histograms from the existing ones

Parameters

- **bareResults** – list of nominal and systematic variation histograms for this *Plot*
- **fbe** – reference to the backend
- **key** – key under which the backend stores the results (if any)

Returns
bareResults

class bamboo.plots.**Product**(*name*, *key=None*)

Interface for output products (plots, counters etc.)

__init__(*name*, *key=None*)

produceResults(*bareResults*, *fbe*, *key=None*)

Main interface method, called by the backend

Parameters

- **bareResults** – iterable of histograms for this plot produced by the backend
- **fbe** – reference to the backend
- **key** – key under which the backend stores the results (if any)

Returns
an iterable with ROOT objects to save to the output file

class bamboo.plots.**Selection**(*parent*, *name*, *cuts=None*, *weights=None*, *autoSyst=True*)

A *Selection* object groups a set of selection criteria (cuts) and weight factors that belong to a specific stage of the selection and analysis. Selections should be constructed by calling the *refine()* method on a 'root' selection (which may include overall selections and weights, e.g. a lumi mask for data and pileup reweighting for MC).

> **Note**
>
> All *Selection* (and *Plot*) instances need to have a unique name. This name is used internally to define DataFrame columns with readable names. The constructor will raise an exception if an existing name is used.

__init__(*parent*, *name*, *cuts=None*, *weights=None*, *autoSyst=True*)

Constructor. Prefer using *refine()* instead
(except for the 'root' selection)

Parameters

- **parent** – backend or parent selection
- **name** – (unique) name of the selection
- **cuts** – iterable of selection criterion expressions (optional)
- **weights** – iterable of weight factors (optional)

refine(*name*, *cut=None*, *weight=None*, *autoSyst=True*)

Create a new selection by adding cuts and/or weight factors

Parameters

- **name** – unique name of the new selection
- **cut** – expression (or list of expressions) with additional selection criteria (combined using logical AND)
- **weight** – expression (or list of expressions) with additional weight factors
- **autoSyst** – automatically add systematic variations (True by default - set to False to turn off; note that this would also turn off automatic systematic variations for any selections and plots that derive from the one created by this method)

Returns
the new *Selection*

class bamboo.plots.**SelectionWithDataDriven**(*parent*, *name*, *cuts=None*, *weights=None*, *autoSyst=True*, *sub=None*)

A main *Selection* with the corresponding "shadow"
Selection instances for evaluating data-driven backgrounds (alternative cuts and/or weights)

static create(*parent*, *name*, *ddSuffix*, *cut=None*, *weight=None*, *autoSyst=True*, *ddCut=None*, *ddWeight=None*, *ddAutoSyst=True*, *enable=True*)

Create a selection with a data-driven shadow selection

Drop-in replacement for a *bamboo.plots.Selection.refine()* call: the main selection is made from the parent with `cut` and `weight`, the shadow selection is made from the parent with `ddCut` and `ddWeight`. With `enable=False` no shadow selection is made (this may help to avoid duplication in the calling code).

class bamboo.plots.**SelectionWithMultipleDataDriven**(*parent*, *name*, *cuts=None*, *weights=None*, *autoSyst=True*, *sub=None*)

A main *Selection* with the corresponding "shadow"

Selection instances for evaluating data-driven backgrounds (alternative cuts and/or weights)

static create(*parent*, *name*, *ddSuffixes*, *cut=None*, *weight=None*, *autoSyst=True*, *ddCuts=None*, *ddWeights=None*, *ddAutoSyst=True*, *enables=True*)

Create a selection with a data-driven shadow selections

Drop-in replacement for a *bamboo.plots.Selection.refine()* call: the main selection is made from the parent with `cut` and `weight`, the shadow selections are made from the parent with `ddCuts` and `ddWeights`. With `enables=[False]` no shadow selections are made (this may help to avoid duplication in the calling code).

This class is derived from a *bamboo.plots.SelectionWithDataDriven()* class and it works in a similar manner. In contrast to the *bamboo.plots.SelectionWithDataDriven()* class, this class can be used for the selection with multiple data-driven backgrounds at the same time. The `ddCuts`, `ddWeights`, `ddSuffixes` and `enables` need to be lists of ddCuts, ddWeights, ddSuffixes and enables, respectively.

class bamboo.plots.**SelectionWithSub**(*parent*, *name*, *cuts=None*, *weights=None*, *autoSyst=True*, *sub=None*)

A common base class for *Selection* subclasses

with related/alternative/sub-*Selection* instances attached

A dictionary of additional selections is kept in the `sub` attribute (could be `None` to disable).

__init__(*parent*, *name*, *cuts=None*, *weights=None*, *autoSyst=True*, *sub=None*)

Constructor. Prefer using *refine()* instead

(except for the 'root' selection)

Parameters

- **parent** – backend or parent selection
- **name** – (unique) name of the selection
- **cuts** – iterable of selection criterion expressions (optional)
- **weights** – iterable of weight factors (optional)

static getSubsForPlot(*p*, *requireActive=False*, *silent=False*)

Helper method: gather the sub-selections for which a plot is produced

initSub()

Initialize related selections

(no-op by default, subclasses can request to call this to enable some functionality)

refine(*name*, *cut=None*, *weight=None*, *autoSyst=True*)

Create a new selection by adding cuts and/or weight factors

Parameters

- **name** – unique name of the new selection
- **cut** – expression (or list of expressions) with additional selection criteria (combined using logical AND)
- **weight** – expression (or list of expressions) with additional weight factors

- **autoSyst** – automatically add systematic variations (True by default - set to False to turn off; note that this would also turn off automatic systematic variations for any selections and plots that derive from the one created by this method)

Returns
the new *Selection*

class bamboo.plots.**Skim**(*name*, *branches*, *selection*, *keepOriginal=None*, *maxSelected=-1*, *compressionLevel=1*, *compressionAlgo=None*, *treeName=None*, *key=None*)

Save selected branches for events that pass the selection to a skimmed tree

__init__(*name*, *branches*, *selection*, *keepOriginal=None*, *maxSelected=-1*, *compressionLevel=1*, *compressionAlgo=None*, *treeName=None*, *key=None*)

Skim constructor

Parameters

- **name** – name of the skim (also default name of the TTree)
- **branches** – dictionary of branches to keep (name and definition for new branches, or name and `None` for specific branches from the input tree)
- **selection** – *Selection* of events to save
- **keepOriginal** – list of branch names to keep, `bamboo.plots.Skim.KeepRegex` instances with patterns of branch names to keep, or `bamboo.plots.Skim.KeepAll` to keep all branches from the input tree
- **maxSelected** – maximal number of events to keep (default: no limit)
- **compressionLevel** – compression level of output file (default: 1)
- **compressionAlgo** – compression algorithm of output file (default: ZLIB)

Example

```
>>> plots.append(Skim("dimuSkim", {
>>>     "run": None,  # copy from input
>>>     "luminosityBlock": None,
>>>     "event": None,
>>>     "dimu_m": op.invariant_mass(muons[0].p4, muons[1].p4),
>>>     "mu1_pt": muons[0].pt,
>>>     "mu2_pt": muons[1].pt,
>>>     }, twoMuSel,
>>>     keepOriginal=[
>>>         Skim.KeepRegex("PV_.*"),
>>>         "nOtherPV",
>>>         Skim.KeepRegex("OtherPV_.*")
>>>         ])
```

produceResults(*bareResults*, *fbe*, *key=None*)

Main interface method, called by the backend

Parameters

- **bareResults** – iterable of histograms for this plot produced by the backend
- **fbe** – reference to the backend
- **key** – key under which the backend stores the results (if any)

Returns
an iterable with ROOT objects to save to the output file

class bamboo.plots.**SummedPlot**(*name*, *termPlots*, ***kwargs*)

A *DerivedPlot* implementation that sums histograms

__init__(*name*, *termPlots*, ***kwargs*)

produceResults(*bareResults*, *fbe*, *key=None*)

Main interface method, called by the backend

Parameters

- **bareResults** – iterable of histograms for this plot produced by the backend (none)
- **fbe** – reference to the backend, can be used to retrieve the histograms for the dependencies, e.g. with *collectDependencyResults()*
- **key** – key under which the backend stores the results (if any)

Returns
an iterable with ROOT objects to save to the output file

class bamboo.plots.**VariableBinning**(*binEdges*)

Variable-sized binning

__init__(*binEdges*)

Parameters
binEdges – iterable with the edges. There will be `len(binEges)-1` bins

2.7.2 Analysis modules

Minimally, `bambooRun` needs a class with a constructor that takes a single argument (the list of command-line arguments that it does not recognize as its own), and a `run` method that takes no arguments. *bamboo.analysismodules* provides more interesting base classes, starting from *AnalysisModule*, which implements a large part of the common functionality for loading samples and distributing worker tasks. *HistogramsModule* specializes this further for modules that output stack histograms, and *NanoAODHistoModule* supplements this with loading the decorations for NanoAOD, and merging of the counters for generator weights etc.

Note

When defining a base class that should also be usable for other things than only making plots or only making skims (e.g. both of these) it should not inherit from *HistogramsModule* or *SkimmerModule* (but the concrete classes should); otherwise a concrete class may end up inheriting from both (at which point the method resolution order will decide whether it behaves as a skimmer or a plotter, and the result may not be obvious).

A typical case should look like this:

```
class MyBaseClass(NanoAODModule):
    ...  # define addArgs, prepareTree etc.
class MyPlotter(MyBaseClass, HistogramsModule):
    ...
class MySkimmer(MyBaseClass, SkimmerModule):
    ...
```

class bamboo.analysismodules.**AnalysisModule**(*args*)

Base analysis module

Adds common infrastructure for parsing analysis config files and running on a batch system, with customization points for concrete classes to implement

__init__(*args*)

Constructor

set up argument parsing, calling *addArgs()* and *initialize()*

Parameters
args – list of command-line arguments that are not parsed by bambooRun

addArgs(*parser*)

Hook for adding module-specific argument parsing (receives an argument group),
parsed arguments are available in self.args afterwards

customizeAnalysisCfg(*analysisCfg*)

Hook to modify the analysis configuration before jobs are created
(only called in driver or non-distributed mode)

getATree(*fileName=None*, *sampleName=None*, *config=None*)

Retrieve a representative TTree, e.g. for defining the plots or interactive inspection,
and a dictionary with metadatas

getTasks(*analysisCfg*, *resolveFiles=None*, ***extraOpts*)

Get tasks from analysis configs (and args), called in for driver or sequential mode

Returns
a list of SampleTask instances

initialize()

Hook for module-specific initialization (called from the constructor after parsing arguments)

postProcess(*taskList*, *config=None*, *workdir=None*, *resultsdir=None*)

Do postprocessing on the results of the tasks, if needed

should be implemented by concrete modules

Parameters

- **taskList** – (inputs, output), kwargs for the tasks (list, string, and dictionary)
- **config** – parsed analysis configuration file
- **workdir** – working directory for the current run
- **resultsdir** – path with the results files

run()

Main method

Depending on the arguments passed, this will:

- if -i or --interactive, call interact() (which could do some initialization and start an IPython shell)
- if --distributed=worker call processTrees() with the appropriate input, output, treename, lumi mask and run range

- if `--distributed=driver` or not given (sequential mode): parse the analysis configuration file, construct the tasks with *getTasks()*, run them (on a batch cluster or in the same process with `processTrees()`), and finally call *postProcess()* with the results.

class bamboo.analysismodules.**DataDrivenBackgroundAnalysisModule**(*args*)

`AnalysisModule` with support for data-driven backgrounds

A number of contributions can be defined, each based on a list of samples or groups needed to evaluate the contribution (typically just data) and a list of samples or groups that should be left out when making the plot with data-driven contributions. The contributions should be defined in the analysis YAML file, with a block `datadriven` (at the top level) that could look as follows:

```
datadriven:
  chargeMisID:
    uses: [ data ]
    replaces: [ DY ]
  nonprompt:
    uses: [ data ]
    replaces: [ TTbar ]
```

The `--datadriven` command-line switch then allows to specify a scenario for data-driven backgrounds, i.e. a list of data-driven contributions to include (`all` and `none` are also possible, the latter is the default setting). The parsed contributions are available as `self.datadrivenContributions`, and the scenarios (each list is a list of contributions) as `self.datadrivenScenarios`.

addArgs(*parser*)

Hook for adding module-specific argument parsing (receives an argument group),
parsed arguments are available in `self.args` afterwards

initialize()

Hook for module-specific initialization (called from the constructor after parsing arguments)

class bamboo.analysismodules.**DataDrivenBackgroundHistogramsModule**(*args*)

`HistogramsModule` with support for data-driven backgrounds

see the `DataDrivenBackgroundAnalysisModule` class for more details about configuring data-driven backgrounds, and the *SelectionWithDataDriven* class for ensuring the necessary histograms are filled correctly. `HistogramsModule` writes the histograms for the data-driven contributions to different files. This one runs `plotIt` for the different scenarios.

postProcess(*taskList*, *config=None*, *workdir=None*, *resultsdir=None*)

Do postprocessing on the results of the tasks, if needed

should be implemented by concrete modules

Parameters

- **taskList** – `(inputs, output), kwargs` for the tasks (list, string, and dictionary)
- **config** – parsed analysis configuration file
- **workdir** – working directory for the current run
- **resultsdir** – path with the results files

class bamboo.analysismodules.**DataDrivenContribution**(*name*, *config*)

Configuration helper class for data-driven contributions

An instance is constructed for each contribution in any of the scenarios by the *bamboo.analysismodules.DataDrivenBackgroundAnalysisModule.initialize()* method, with the name and configuration dictionary found in YAML file. The *usesSample()*:, *replacesSample()*: and *modifiedSampleConfig()*: methods can be customised for other things than using the data samples to estimate a background contribution.

__init__(*name*, *config*)

modifiedSampleConfig(*sampleName*, *sampleConfig*, *lumi=None*)

Construct the sample configuration for the reweighted counterpart of a sample

The default implementation assumes a data sample and turns it into a MC sample (the luminosity is set as `generated-events` to avoid changing the normalisation).

replacesSample(*sampleName*, *sampleConfig*)

Check if this contribution replaces a sample (name or group in 'replaces')

usesSample(*sampleName*, *sampleConfig*)

Check if this contribution uses a sample (name or group in 'uses')

class bamboo.analysismodules.**HistogramsModule**(*args*)

Base histogram analysis module

__init__(*args*)

Constructor

Defines a `plotList` member variable, which will store a list of plots (the result of *definePlots()*, which will be called after *prepareTree()*). The *postProcess()* method specifies what to do with the results.

addArgs(*parser*)

Hook for adding module-specific argument parsing (receives an argument group),
parsed arguments are available in `self.args` afterwards

definePlots(*tree*, *noSel*, *sample=None*, *sampleCfg=None*)

Main method: define plots on the trees (for a give systematic variation)

should be implemented by concrete modules, and return a list of *bamboo.plots.Plot* objects. The structure (name, binning) of the histograms should not depend on the sample, era, and the list should be the same for all values (the weights and systematic variations associated with weights or collections may differ for data and different MC samples, so the actual set of histograms will not be identical).

Parameters

- **tree** – decorated tree
- **noSel** – base selection
- **sample** – sample name (as in the samples section of the analysis configuration file)
- **sampleCfg** – that sample's entry in the configuration file

getPlotList(*fileHint=None*, *sampleHint=None*, *resultsdir=None*, *config=None*)

Helper method for postprocessing: construct the plot list

The path (and sample name) of an input file can be specified, otherwise the results directory is searched for a skeleton tree. Please note that in the latter case, the skeleton file is arbitrary (in practice it probably corresponds to the first sample encountered when running in sequential or `--distributed=driver` mode), so if the postprocessing depends on things that are different between samples, one needs to be extra careful to avoid surprises.

Parameters

- **fileHint** – name of an input file for one of the samples
- **sampleHint** – sample name for the input file passed in `fileHint`
- **resultsdir** – directory with the produced results files (mandatory if no `fileHint` and `sampleHint` are passed)
- **config** – analysis config (to override the default - optional)

initialize()

makeBackendAndPlotList(*inputFiles*, *tree=None*, *certifiedLumiFile=None*, *runRange=None*, *sample=None*, *sampleCfg=None*, *inputFileLists=None*, *backend=None*, *distRDFargs=None*)

Prepare and plotList definition (internal helper)

Parameters

- **inputFiles** – input file names
- **tree** – key name of the tree inside the files
- **certifiedLumiFile** – lumi mask json file name
- **runRange** – run range to consider (for efficiency of the lumi mask)
- **sample** – sample name (key in the samples block of the configuration file)
- **sampleCfg** – that sample's entry in the configuration file
- **inputFileLists** – names of files with the input files (optional, to avoid rewriting if this already exists)
- **backend** – type of backend (lazy/default, debug, compiled, distributed)
- **distRDFargs** – parameters for distributed backend

Returns

the backend and plot list (which can be *None* if run in onlyprepare" mode)

mergeCounters(*outF*, *infileNames*, *sample=None*)

Merge counters

should be implemented by concrete modules

Parameters

- **outF** – output file (TFile pointer)
- **infileNames** – input file names
- **sample** – sample name

postProcess(*taskList*, *config=None*, *workdir=None*, *resultsdir=None*)

Postprocess: run plotIt

The list of plots is created if needed (from a representative file, this enables rerunning the postprocessing step on the results files), and then plotIt is executed

prepareTree(*tree*, *sample=None*, *sampleCfg=None*, *backend=None*)

Create decorated tree, selection root (noSel), backend, and (run,LS) expressions

should be implemented by concrete modules

Parameters

- **tree** – decorated tree

- **sample** – sample name (as in the samples section of the analysis configuration file)
- **sampleCfg** – that sample's entry in the configuration file
- **backend** – type of backend (lazy/default, debug, compiled, distributed)

readCounters(*resultsFile*)

Read counters from results file

should be implemented by concrete modules, and return a dictionary with counter names and the corresponding sums

Parameters
resultsFile – TFile pointer to the results file

class bamboo.analysismodules.**NanoAODHistoModule**(*args*)

A *HistogramsModule* for NanoAOD,
with decorations and merging of counters from *NanoAODModule*

__init__(*args*)

Constructor

set up argument parsing, calling *addArgs()* and *initialize()*

Parameters
args – list of command-line arguments that are not parsed by bambooRun

class bamboo.analysismodules.**NanoAODModule**(*args*)

A *AnalysisModule* extension for NanoAOD,
adding decorations and merging of the counters

mergeCounters(*outF*, *infileNames*, *sample=None*)

Merge the Runs trees

prepareTree(*tree*, *sample=None*, *sampleCfg=None*, *description=None*, *backend=None*)

Add NanoAOD decorations, and create an RDataFrame backend

In addition to the arguments needed for the base class prepareTree`()` method, a description of the tree, and settings for reading systematic variations or corrections from alternative branches, or calculating these on the fly, should be passed, such that the decorations can be constructed accordingly.

Parameters
description – description of the tree format, and configuration for reading or calculating systematic variations and corrections, a *NanoAODDescription* instance (see also *bamboo.treedecorators.NanoAODDescription.get()*)

readCounters(*resultsFile*)

Sum over each leaf of the (merged) Runs tree (except run)

class bamboo.analysismodules.**NanoAODSkimmerModule**(*args*)

A *SkimmerModule* for NanoAOD,
with decorations and merging of counters from *NanoAODModule*

__init__(*args*)

Constructor

set up argument parsing, calling *addArgs()* and *initialize()*

Parameters
args – list of command-line arguments that are not parsed by bambooRun

class bamboo.analysismodules.**SkimmerModule**(*args*)

Base skimmer module

Left for backwards-compatibility, please use a *HistogramsModule* that defines one or more *bamboo.plots.Skim* products instead.

addArgs(*parser*)

Hook for adding module-specific argument parsing (receives an argument group),
parsed arguments are available in `self.args` afterwards

definePlots(*tree*, *noSel*, *sample=None*, *sampleCfg=None*)

Main method: define plots on the trees (for a give systematic variation)

should be implemented by concrete modules, and return a list of *bamboo.plots.Plot* objects. The structure (name, binning) of the histograms should not depend on the sample, era, and the list should be the same for all values (the weights and systematic variations associated with weights or collections may differ for data and different MC samples, so the actual set of histograms will not be identical).

Parameters

- **tree** – decorated tree
- **noSel** – base selection
- **sample** – sample name (as in the samples section of the analysis configuration file)
- **sampleCfg** – that sample's entry in the configuration file

defineSkimSelection(*tree*, *noSel*, *sample=None*, *sampleCfg=None*)

Main method: define a selection for the skim

should be implemented by concrete modules, and return a *bamboo.plots.Selection* object

Parameters

- **tree** – decorated tree
- **noSel** – base selection
- **sample** – sample name (as in the samples section of the analysis configuration file)
- **sampleCfg** – that sample's entry in the configuration file

Returns
the skim *bamboo.plots.Selection*, and a map `{ name: expression }` of branches to store (to store all the branches of the original tree in addition, pass –keepOriginalBranches to bambooRun; individual branches can be added with an entry `name: None` entry)

2.7.3 Tree decorator customisation

Expressions are constructed by executing python code on decorated versions of decorated trees. The *bamboo.treedecorators* module contains helper methods to do so for commonly used formats, e.g. *decorateNanoAOD()* for CMS NanoAOD.

class bamboo.treedecorators.**NanoSystematicVarSpec**(*nomName=None*, *origName=None*, *exclVars=None*, *isCalc=False*)

Interface for classes that specify how to incorporate systematics
or on-the-fly corrections in the decorated tree

See *NanoAODDescription* and *decorateNanoAOD()*

appliesTo(*name*)

Return true if this systematic variation requires action
for this variable, group, or collection

changesTo(*name*)

Return the new name(s) for a collection or group (assuming appliesTo(name) is True)

getVarName(*branchName*, *collgrpname=None*)

Get the variable name and variation corresponding to an
(unprefixed, in case of groups or collections) branch name

nomName(*name*)

Nominal systematic variation name for a group/collection

exclVars(*name*)

Systematic variations to exclude for a group/collection

class bamboo.treedecorators.**ReadVariableVarWithSuffix**(*commonName*, *sep='_'*, *nomName='nominal'*, *exclVars=None*)

Read variations of a single branch from branches with the same name with a suffix

appliesTo(*name*)

True if name starts with the prefix

getVarName(*branchName*, *collgrpname=None*)

Split into prefix and variation (if present, else nominal)

class bamboo.treedecorators.**ReadJetMETVar**(*jetsName*, *metName*, *jetsNomName='nom'*, *jetsOrigName='raw'*, *metNomName=''*, *metOrigName='raw'*, *jetsExclVars=None*, *metExclVars=None*, *bTaggers=None*, *bTagWPs=None*)

Read jet and MET kinematic variations from different branches for automatic systematic uncertainties

Parameters

- **jetsName** – jet collection prefix (e.g. `"Jet"`)
- **metName** – MET prefix (e.g. `"MET"`)
- **jetsNomName** – name of the nominal jet variation (`"nom"` by default)
- **jetsOrigName** – name of the original jet variation (`"raw"` by default)
- **metNomName** – name of the nominal jet variation (`"nom"` by default)
- **metOrigName** – name of the original jet variation (`"raw"` by default)
- **jetsExclVars** – jet variations that are present but should be ignored (if not specified, only `jetsOrigName` is taken, so if specified this should usually be added explicitly)
- **metExclVars** – MET variations that are present but should be ignored (if not specified, only `metOrigName` is taken, so if specified this should usually be added explicitly)
- **bTaggers** – list of b-tagging algorithms, for scale factors stored in a branch
- **bTagWPs** – list of b-tagging working points, for scale factors stored in a branch (`shape` should be included here, if wanted)

> **Note**
>
> The implementation of automatic systematic variations treats "xyzup" and "xyzdown" independently (since this is the most flexible). If a source of systematic uncertainty should be excluded, both the "up" and "down" variation should then be added to the list of variations to exclude (`jetsExclVars` or `metExclVars`).

appliesTo(*name*)

Return true if this systematic variation requires action
for this variable, group, or collection

nomName(*name*)

Nominal systematic variation name for a group/collection

exclVars(*name*)

Systematic variations to exclude for a group/collection

getVarName(*nm*, *collgrpname=None*)

Get the variable name and variation corresponding to an
(unprefixed, in case of groups or collections) branch name

class bamboo.treedecorators.**NanoReadRochesterVar**(*systName=None*)

Read precalculated Rochester correction variations

Parameters
systName – name of the systematic uncertainty, if variations should be enabled

appliesTo(*name*)

Return true if this systematic variation requires action
for this variable, group, or collection

getVarName(*nm*, *collgrpname=None*)

Get the variable name and variation corresponding to an
(unprefixed, in case of groups or collections) branch name

class bamboo.treedecorators.**CalcCollectionsGroups**(*nomName='nominal'*, *origName='raw'*, *exclVars=None*, *changes=None*, ***colsAndAttrs*)

NanoSystematicVarSpec **for on-the-fly corrections**
and systematic variation calculation

appliesTo(*name*)

Return true if this systematic variation requires action
for this variable, group, or collection

changesTo(*name*)

Return the new name(s) for a collection or group (assuming appliesTo(name) is True)

getVarName(*nm*, *collgrpname=None*)

Get the variable name and variation corresponding to an
(unprefixed, in case of groups or collections) branch name

class bamboo.treedecorators.**NanoAODDescription**(*groups=None*, *collections=None*, *systVariations=None*)

Description of the expected NanoAOD structure, and configuration for systematics and corrections

Essentially, a collection of three containers:

- `collections` a list of collections (by the name of the length leaf)
- `groups` a list of non-collection groups (by prefix, e.g. `HLT_`)
- **`systVariations` a list of *`NanoSystematicVarSpec`* instances,**
  to configure reading systematics variations from branches, or calculating them on the fly

The recommended way to obtain a configuration is from the factory method *`get()`*

static get(*tag*, *year='2016'*, *isMC=False*, *addGroups=None*, *removeGroups=None*, *addCollections=None*, *removeCollections=None*, *systVariations=None*)

Create a suitable NanoAODDescription instance based on a production version

A production version is defined by a tag, data-taking year, and a flag to distinguish data from simulation. Any number of groups or collections can be added or removed from this. The `systVariations` option

Example

```
>>> decorateNanoAOD(tree, NanoAODDescription.get(
>>>     "v5", year="2016", isMC=True,
>>>     systVariations=[nanoRochesterCalc, nanoJetMETCalc]))
>>> decorateNanoAOD(tree, NanoAODDescription.get(
>>>     "v5", year="2017", isMC=True,
>>>     systVariations=[nanoPUWeightVar, nanoReadJetMETVar_METFixEE2017]))
```

Parameters

- **`tag`** – production version (e.g. "v5")
- **`year`** – data-taking year
- **`isMC`** – simulation or not
- **`addGroups`** – (optional) list of groups of leaves to add (e.g. `["L1_", "HLT_"]`, if not present)
- **`removeGroups`** – (optional) list of groups of leaves to remove (e.g. `["L1_"]`, if skimmed)
- **`addCollections`** – (optional) list of containers to add (e.g. `["nMyJets"]`)
- **`removeCollections`** – (optional) list of containers to remove (e.g. `["nPhoton", "nTau"]`)
- **`systVariations`** – list of correction or systematic variation on-the-fly calculators or configurations to add (*`NanoSystematicVarSpec`* instances)

See also *`decorateNanoAOD()`*

bamboo.treedecorators.**decorateNanoAOD**(*aTree*, *description=None*)

Decorate a CMS NanoAOD Events tree

Variation branches following the NanoAODTools conventions (e.g. Jet_pt_nom) are automatically used (but calculators for the same collection take precendence, if requested).

Parameters

- **`aTree`** – TTree to decorate

- **description** – description of the tree format, and configuration for reading or calculating systematic variations and corrections, a *NanoAODDescription* instance (see also *NanoAODDescription.get()*)

bamboo.treedecorators.**decorateCMSPhase2SimTree**(*aTree*, *isMC=True*)

Decorate a flat tree as used for CMS Phase2 physics studies

2.7.4 Helper functions

The *bamboo.analysisutils* module bundles a number of more specific helper methods that use the tree decorators and integrate with other components, connect to external services, or are factored out of the classes in *bamboo.analysismodules* to facilitate reuse.

class bamboo.analysisutils.**YMLIncludeLoader**(*stream*)

Custom yaml loading to support including config files.
Use *!include (file)* to insert content of *file* at that position.

bamboo.analysisutils.**addLumiMask**(*sel*, *jsonName*, *runRange=None*, *runAndLS=None*, *name='goodlumis'*)

Refine selection with a luminosity block filter

Typically applied directly to the root selection (for data). runAndLS should be a tuple of expressions with the run number and luminosity block ID. The run range is used to limit the part of the JSON file to consider, see the LumiMask helper class for details.

bamboo.analysisutils.**addPrintout**(*selection*, *funName*, **args*)

Call a method with debugging printout, as part of the RDataFrame graph

This method is only meant to work with the default backend, since it works by inserting a `Filter` node that lets all events pass.

Parameters

- **selection** – selection for which to add the printout. The function call will be added to the RDataFrame graph in its current state, so if a plot causes a problem this method should be called before defining it.
- **funName** – name of a C++ method to call. This method should always return `true`, and can take any number of arguments.
- **args** – arguments to pass to the function

The following example would print the entry and event number for each event that passes some selection.

Example

```
>>> from bamboo.root import gbl
>>> gbl.gInterpreter.Declare("""
... bool bamboo_printEntry(long entry, long event) {
...   std::cout << "Processing entry #" << entry << ": event " << event <<␣
↪std::endl;
... }""")
>>> addPrintout(sel, "bamboo_printEntry", op.extVar("ULong_t", "rdfentry_"), t.
↪event)
```

bamboo.analysisutils.**configureElectrons**(*variProxy*, *paramsFile*, *scale=''*, *smearing=''*, *jsonFileRandomGenerator=''*, *randomDistr='stdnormal'*, *addSystematics=False*, *enableSystematics=None*, *isMC=True*, *backend=None*)

Apply the energy correction for electrons (Run3 only)

Parameters

- **variProxy** – tau variatons proxy, e.g. `tree._Electron` for NanoAOD
- **paramsFile** – path of the json file with correction parameters
- **scale** – name of the scale correction, e.g. `EGMScale_Compound_Ele_2022postEE`
- **smearing** – name of the smearing correction, e.g. `EGMSmearAndSyst_ElePTsplit_2022postEE`
- **jsonFileRandomGenerator** – json file with random number generator (e.g. https://gitlab.cern.ch/cms-analysis/general/bamboo/-/blob/master/ tests/data/randomNumbers.json.gz)
- **randomDistr** – random distribution, default is `stdnormal`
- **addSystematics** – add systematic uncertaintis
- **enableSystematics** – filter systematics variations to enable (collection of names or callable that takes the variation name; default: all that are available for MC, none for data)
- **isMC** – MC or not
- **backend** – backend pointer (returned from *prepareTree()*)

bamboo.analysisutils.**configureJets**(*variProxy*, *jsonFile*, *jetType*, *jec=None*, *jecLevels='default'*, *smear=None*, *useGenMatch=True*, *genMatchDR=0.2*, *genMatchDPt=3.0*, *jesUncertaintySources=None*, *smearingToolName='JERSmear'*, *jsonFileSmearingTool=None*, *splitJER=False*, *addHEM2018Issue=False*, *enableSystematics=None*, *jetAlgoSubjet='AK4PFPuppi'*, *jecSubjet=None*, *jsonFileSubjet=None*, *jecLevelSubjet='default'*, *subjets=None*, *isMC=False*, *jecJetType=None*, *backend=None*)

Reapply JEC, set up jet smearing, or prepare JER/JES uncertainties collections

Parameters

- **variProxy** – jet variations proxy, e.g. `tree._Jet`
- **jsonFile** – json file with JEC corrections
- **jetType** – jet type, e.g. AK4PFchs
- **smear** – tag of resolution (and scalefactors) to use for smearing (no smearing is done if unspecified)
- **jec** – tag of the new JEC to apply, or for the JES uncertainties (pass an empty list to jecLevels to produce only the latter without reapplying the JEC)
- **jecLevels** – list of JEC levels to apply (if left out the recommendations are used: L1FastJet, L2Relative, L3Absolute, and also L2L3Residual for data)
- **jesUncertaintySources** – list of jet energy scale uncertainty sources (see the JECUncertaintySources twiki),
- **enableSystematics** – filter systematics variations to enable (collection of names or callable that takes the variation name; default: all that are available for MC, none for data)
- **useGenMatch** – use matching to generator-level jets for resolution smearing
- **genMatchDR** – DeltaR for generator-level jet matching (half the cone size is recommended, default is 0.2)
- **genMatchDPt** – maximal relative PT difference (in units of the resolution) between reco and gen jet

- **jsonFileSmearingTool** – json file with JER smearing tool (e.g. /cvmfs/cms.cern.ch/rsync/cms-nanoAOD/jsonpog-integration/POG/JME/jer_smear.json.gz)
- **smearingToolName** – name of the JER smearing tool, default is `JERSmear`
- **splitJER** – vary the JER uncertainty independently in six kinematic bins (see the JER uncertainty twiki)
- **addHEM2018Issue** – add a JES uncertainty for the HEM issue in 2018 (see this hypernews post)
- **subjets** – subjets proxy (`tree.SubJet`)
- **isMC** – MC or not
- **jecJetType** – jet type to use for JEC payload lookup (optional). Defaults to `jetType` if not specified. Useful when applying JEC of a different jet collection (e.g. AK4PFPuppi JEC applied to AK8PFPuppi jets).
- **backend** – backend pointer (returned from *prepareTree()*)

bamboo.analysisutils.**configureMuons**(*variProxy*, *paramsFile*, *jsonFileRandomGenerator*, *randomDistr='stdflat'*, *addSystematics=False*, *enableSystematics=None*, *isMC=True*, *backend=None*)

Apply the energy correction for muons (Run3 only)

Parameters

- **variProxy** – tau variatons proxy, e.g. `tree._Muon` for NanoAOD
- **paramsFile** – path of the json file with correction parameters
- **jsonFileRandomGenerator** – json file with random number generator (e.g. https://gitlab.cern.ch/cms-analysis/general/bamboo/-/blob/master/ tests/data/randomNumbers.json.gz)
- **randomDistr** – random distribution, default is `stdflat`
- **addSystematics** – add systematic uncertaintis
- **enableSystematics** – filter systematics variations to enable (collection of names or callable that takes the variation name; default: all that are available for MC, none for data)
- **isMC** – MC or not
- **backend** – backend pointer (returned from *prepareTree()*)

bamboo.analysisutils.**configureRochesterCorrection**(*variProxy*, *paramsFile*, *isMC=False*, *backend=None*, *uName=''*)

Apply the Rochester correction for muons

Parameters

- **variProxy** – muon variatons proxy, e.g. `tree._Muon` for NanoAOD
- **paramsFile** – path of the text file with correction parameters
- **isMC** – MC or not
- **backend** – backend pointer (returned from *prepareTree()*)
- **uName** – [deprecated, ignored] unique name for the correction calculator (sample name is a safe choice)

bamboo.analysisutils.**configureSVfitCalculator**(*pathToSVfit=''*, *backend=None*)

Configure SVfit

Parameters

- **pathToSVfit** – path to your SVfit installation
- **backend** – backend pointer (returned from *prepareTree()*)

bamboo.analysisutils.**configureTaus**(*variProxy*, *paramsFile*, *tauIdAlgo*, *tauCorr='tau_energy_scale'*, *tauWP=None*, *tauWPvsE=None*, *addSystematics=False*, *splitSystematics=False*, *enableSystematics=None*, *isMC=False*, *backend=None*)

Apply the energy correction for taus

Parameters

- **variProxy** – tau variatons proxy, e.g. `tree._Tau` for NanoAOD
- **paramsFile** – path of the json file with correction parameters
- **tauIdAlgo** – name of the algorithm for the tau identification, e.g. "DeepTau2017v2p1"
- **tauCorr** – name of the correction, e.g. "tau_energy_scale"
- **tauWP** – working point for DeepTau2018v2p5VSjet
- **tauWPvsE** – working point for DeepTau2018v2p5VSe
- **addSystematics** – add systematic uncertainties
- **splitSystematics** – split systematic uncertainties (If splitSystematics is set to true, the TES systematic uncertainty is split between genuine taus and e->tau fakes.)
- **enableSystematics** – filter systematics variations to enable (collection of names or callable that takes the variation name; default: all that are available for MC, none for data)
- **isMC** – MC or not
- **backend** – backend pointer (returned from *prepareTree()*)

bamboo.analysisutils.**configureType1MET**(*variProxy*, *jsonFile*, *jec=None*, *jecLevels='default'*, *smear=None*, *isT1Smear=False*, *useGenMatch=True*, *genMatchDR=0.2*, *genMatchDPt=3.0*, *jesUncertaintySources=None*, *smearingToolName='JERSmear'*, *jsonFileSmearingTool=None*, *splitJER=False*, *isXYCorrected=False*, *eraForXYCorrection=None*, *jsonFileXYCorrection=None*, *addHEM2018Issue=False*, *enableSystematics=None*, *isMC=False*, *backend=None*)

Reapply JEC, set up jet smearing, or prepare JER/JES uncertainties collections

Parameters

- **variProxy** – MET variations proxy, e.g. `tree._MET`
- **jsonFile** – json file with JEC corrections
- **smear** – tag of resolution (and scalefactors) to use for smearing (no smearing is done if unspecified)
- **isT1Smear** – T1Smear (smeared as nominal, all variations with respect to that) if True, otherwise T1 (JES variations with respect to the unsmeared MET, jerup and jerdown variations are nominally smeared)
- **jec** – tag of the new JEC to apply, or for the JES uncertainties

- **jesUncertaintySources** – list of jet energy scale uncertainty sources (see the JECUncertaintySources twiki),
- **enableSystematics** – filter systematics variations to enable (collection of names or callable that takes the variation name; default: all that are available for MC, none for data)
- **useGenMatch** – use matching to generator-level jets for resolution smearing
- **genMatchDR** – DeltaR for generator-level jet matching (half the cone size is recommended, default is 0.2)
- **genMatchDPt** – maximal relative PT difference (in units of the resolution) between reco and gen jet
- **jsonFileSmearingTool** – json file with JER smearing tool (e.g. /cvmfs/cms.cern.ch/rsync/cms-nanoAOD/jsonpog-integration/POG/JME/jer_smear.json.gz)
- **smearingToolName** – name of the JER smearing tool, default is `JERSmear`
- **splitJER** – vary the JER uncertainty independently in six kinematic bins (see the JER uncertainty twiki)
- **addHEM2018Issue** – add a JES uncertainty for the HEM issue in 2018 (see this hypernews post)
- **isXYCorrected** – apply XY correction to MET
- **eraForXYCorrection** – era for the XY Correction, e.g 2022, 2022EE, 2023, or 2023BPix
- **jsonFileXYCorrection** – json file with XY corrections
- **isMC** – MC or not
- **be** – backend pointer (returned from *prepareTree()*)

bamboo.analysisutils.**forceDefine**(*arg*, *selection*, *includeSub=True*)

Force the definition of an expression as a column at a selection stage

Use only for really computation-intensive operations that need to be precalculated

Parameters

- **arg** – expression to define as a column
- **selection** – *Selection* for which the expression should be defined
- **includeSub** – also precalculate for data-driven background 'shadow' selections (*bamboo.plots.SelectionWithSub* 'sub'-selections)

bamboo.analysisutils.**getAFileFromAnySample**(*samples*, *resolveFiles=None*, *cfgDir='.'*)

Helper method: get a file from any sample (minimizing the risk of errors)

Tries to find any samples with: - a list of files - a cache file - a SAMADhi path - a DAS path

If successful, a single read / query is sufficient to retrieve a file

bamboo.analysisutils.**loadPlotIt**(*config*, *plotList*, *eras=None*, *workdir='.'*, *resultsdir='.'*, *readCounters=<function <lambda>>*, *vetoFileAttributes=None*, *plotDefaults=None*)

Load the plotit configuration with the plotIt python library

The plotIt YAML file writing and parsing is skipped in this case (to write the file, the *writePlotIt()* method should be used, with the same arguments).

Parameters

- **config** – parsed analysis configuration. Only the `configuration` (if present) and `eras` sections (to get the luminosities) are read.
- **plotList** – list of plots to convert (`name` and `plotopts`, combined with the default style)
- **eras** – list of eras to consider (`None` for all that are in the config)
- **workdir** – output directory
- **resultsdir** – directory with output ROOT files with histograms
- **readCounters** – method to read the sum of event weights from an output file
- **vetoFileAttributes** – list of per-sample keys that should be ignored (those specific to the bamboo part, e.g. job splitting and DAS paths)
- **plotDefaults** – plot defaults to add (added to those from `config["plotIt"]["plotdefaults"]`, with higher precedence if present in both)

bamboo.analysisutils.**makeMultiPrimaryDatasetTriggerSelection**(*sampleName*, *datasetsAndTriggers*)

Construct a selection that prevents processing multiple times (from different primary datasets)

If an event is passes triggers for different primary datasets, it will be taken from the first of those (i.e. the selection will be 'passes one of the triggers that select it for this primary dataset, and not for any of those that come before in the input dictionary).

Parameters

- **sampleName** – sample name
- **datasetsAndTriggers** – a dictionary `{primary-dataset, set-of-triggers}`, where the key is either a callable that takes a sample name and returns true in case it originates from the corresponding primary datasets, or a string that is the first part of the sample name in that case. The value (second item) can be a single expression (e.g. a trigger flag, or an OR of them), or a list of those (in which case an OR-expression is constructed from them).

Returns

an expression to filter the events in the sample with given name

Example

```
>>> if not self.isMC(sample):
>>>     trigSel = noSel.refine("trigAndPrimaryDataset",
>>>         cut=makeMultiPrimaryDatasetTriggerSelection(sample, {
>>>               "DoubleMuon" : [t.HLT.Mu17_TrkIsoVVL_Mu8_TrkIsoVVL,
>>>                               t.HLT.Mu17_TrkIsoVVL_TkMu8_TrkIsoVVL],
>>>               "DoubleEG"   : t.HLT.Ele23_Ele12_CaloIdL_TrackIdL_IsoVL_DZ,
>>>               "MuonEG"     : [t.HLT.Mu23_TrkIsoVVL_Ele12_CaloIdL_TrackIdL_IsoVL,
>>>                               t.HLT.Mu8_TrkIsoVVL_Ele23_CaloIdL_TrackIdL_IsoVL ]
>>>               }))
```

bamboo.analysisutils.**makePileupWeight**(*puWeights*, *numTrueInteractions*, *systName=None*, *nameHint=None*, *sel=None*, *defineOnFirstUse=True*)

Construct a pileup weight for MC, based on the weights in a JSON file

Parameters

- **puWeights** – path of the JSON file with weights (binned in NumTrueInteractions) for cp3-llbb JSON, or tuple of JSON path and correction name (correctionlib JSON)
- **numTrueInteractions** – expression to get the number of true interactions (Poissonian expectation value for an event)

- **systName** – name of the associated systematic nuisance parameter
- **sel** – a selection in the current graph (only used to retrieve a pointer to the backend)

bamboo.analysisutils.**printCutFlowReports**(*config*, *reportList*, *workdir='.'*, *resultsdir='.'*, *suffix=None*, *readCounters=<function <lambda>>*, *eras=('all', None)*, *verbose=False*)

Print yields to the log file, and write a LaTeX yields table for each

Samples can be grouped (only for the LaTeX table) by specifying the `yields-group` key (overriding the regular `groups` used for plots). The sample (or group) name to use in this table should be specified through the `yields-title` sample key.

In addition, the following options in the `plotIt` section of the YAML configuration file influence the layout of the LaTeX yields table:

- `yields-table-stretch`: `\arraystretch` value, 1.15 by default
- **`yields-table-align`: orientation, `h` (default), samples in rows,** or `v`, samples in columns
- `yields-table-text-align`: alignment of text in table cells (default: `c`)
- **`yields-table-numerical-precision-yields`: number of digits after** the decimal point for yields (default: 1)
- **`yields-table-numerical-precision-ratio`: number of digits after** the decimal point for ratios (default: 2)

bamboo.analysisutils.**readEnvConfig**(*explName=None*)

Read computing environment config file (batch system, storage site etc.)

For using a batch cluster, the [batch] section should have a 'backend' key, and there should be a section with the name of the backend (slurm, htcondor...), see bamboo.batch_<backend> for details. The storage site information needed to resolve the PFNs for datasets retrieved from DAS should be specified under the [das] section (sitename and storageroot).

bamboo.analysisutils.**runPlotIt**(*cfgName*, *workdir='.'*, *plotsdir='plots'*, *plotIt='plotIt'*, *eras=('all', None)*, *verbose=False*)

Run plotIt

Parameters

- **cfgName** – plotIt YAML config file name
- **workdir** – working directory (also the starting point for finding the histograms files, `--i` option)
- **plotsdir** – name of the plots directory inside workdir (`plots`, by default)
- **plotIt** – path of the `plotIt` executable
- **eras** – `(mode, eras)`, mode being one of `"split"`, `"combined"`, or `"all"` (both of the former), and eras a list of era names, or `None` for all
- **verbose** – print the plotIt command being run

bamboo.analysisutils.**splitVariation**(*variProxy*, *variation*, *regions*, *nomName='nom'*)

Split a systematic variation between (kinematic) regions (to decorrelate the nuisance parameter)

Parameters

- **variProxy** – jet variations proxy, e.g. `tree._Jet`

- **variation** – name of the variation that should be split (e.g. "jer")
- **regions** – map of region names and selections (for non-collection objects: boolean expression, for collection objects: a callable that returns a boolean for an item from the collection)
- **nomName** – name of the nominal variation ("nom" for postprocessed, "nominal" for calculator)

Example

```
>>> splitVariation(tree._Jet, "jer", {"forward" : lambda j : j.eta > 0.,
>>>                                   "backward" : lambda j : j.eta < 0.})
```

bamboo.analysisutils.**writePlotIt**(*config*, *plotList*, *outName*, *eras=None*, *workdir='.'*, *resultsdir='.'*, *readCounters=<function <lambda>>*, *vetoFileAttributes=None*, *plotDefaults=None*)

Combine creation and saving of a plotIt config file

for convenience inside a *HistogramsModule*, the individual parts are also available in *bamboo.analysisutils*.

Parameters

- **config** – parsed analysis configuration. Only the `configuration` (if present) and `eras` sections (to get the luminosities) are read.
- **plotList** – list of plots to convert (`name` and `plotopts`, combined with the default style)
- **outName** – output YAML config file name
- **eras** – valid era list
- **workdir** – output directory
- **resultsdir** – directory with output ROOT files with histograms
- **readCounters** – method to read the sum of event weights from an output file
- **vetoFileAttributes** – list of per-sample keys that should be ignored (those specific to the bamboo part, e.g. job splitting and DAS paths)
- **plotDefaults** – plot defaults to add (added to those from `config["plotIt"]["plotdefaults"]`, with higher precedence if present in both)

Scale factors

The *bamboo.scalefactors* module contains helper methods for configuring scale factors, fake rates etc.

The basic configuration parameter is the JSON file path for a set of scalefactors. There two basic types are

- lepton scale factors (dependent on a number of object variables, e.g. pt and eta),
- jet (b-tagging) scale factors (grouped set for different flavours, for convenience)

Different values (depending on the data-taking period) can be taken into account by weighting or by randomly sampling.

class bamboo.scalefactors.**BtagSF**(*taggerName*, *csvFileName*, *wp=None*, *sysType='central'*, *otherSysTypes=None*, *measurementType=None*, *getters=None*, *jesTranslate=None*, *sel=None*, *uName=None*)

Helper for b- and c-tagging scalefactors using the BTV POG reader

__call__(*jet*, *nomVar=None*, *systVars=None*)

Evaluate the scalefactor for a jet

Please note that this only gives the applicable scalefactor: to obtain the event weight one of the recipes in the POG twiki should be used.

By default the nominal and systematic variations are taken from the *bamboo.scalefactors.BtagSF* instance, but they can be overriden with the `nomVar` and `systVars` keyword arguments. Please note that when using split uncertainties (e.g. for the reshaping method) some uncertainties only apply to specific jet flavours (e.g. c-jets) and the csv file contains zeroes for the other flavours. Then the user code should check the jet flavours, and call this method with the appropriate list of variations for each.

__init__(*taggerName*, *csvFileName*, *wp=None*, *sysType='central'*, *otherSysTypes=None*, *measurementType=None*, *getters=None*, *jesTranslate=None*, *sel=None*, *uName=None*)

Declare a BTagCalibration (if needed) and BTagCalibrationReader (unique, based on `uName`), and decorate for evaluation

> **Warning**
>
> This function is deprecated. Use correctionlib and helpers in *scalefactors* instead.

Parameters

- **taggerName** – first argument for `BTagCalibration`
- **csvFileName** – name of the CSV file with scalefactors
- **wp** – working point (used as `BTagEntry::OP_{wp.upper()}`)
- **sysType** – nominal value systematic type (`"central"`, by default)
- **otherSysTypes** – other systematic types to load in the reader
- **measurementType** – dictionary with measurement type per true flavour (B, C, and UDSG), or a string if the same for all (if not specified, `"comb"` will be used for b- and c-jets, and `incl` for light-flavour jets)
- **getters** – dictionary of methods to get the kinematics and classifier for a jet (the keys `Pt`, `Eta`, `JetFlavour`, and `Discri` are used. For the former three, the defaults are for NanoAOD)
- **jesTranslate** – translation function for JEC systematic variations, from the names in the CSV file to those used for the jets (the default should work for on-the-fly corrections)
- **sel** – a selection in the current graph
- **uName** – unique name, to declare the reader (e.g. sample name)

bamboo.scalefactors.**get_bTagSF_fixWP**(*json_path*, *tagger*, *wp*, *flavour*, *sel*, *jet_pt_variation=None*, *light_method='incl'*, *heavy_method='comb'*, *syst_prefix='btagSF_fixWP_'*, *decorr_wps=False*, *decorr_eras=False*, *era=None*, *full_scheme=False*, *syst_mapping=None*, *defineOnFirstUse=True*)

Build correction evaluator for fixed working point b-tagging scale factors

Loads the b-tagging scale factors as correction object from the JSON file, configures the systematic variations, and returns a callable that can be evaluated on a jet to return the scale factor.

Parameters

- **json_path** – JSON file path
- **tagger** – name of the tagger inside the JSON (not the same as in the event!)
- **wp** – working point of the tagger (*"L"*, *"M"*, *"T"*)
- **flavour** – hadron flavour of the jet (*0*, *4*, *5*)
- **sel** – a selection in the current graph (only used to retrieve a pointer to the backend)
- **jet_pt_variation** – if specified, only use that specific systematic variation (e.g. the *nominal*) of the jet pt to evaluate the scale factors. By default, the scale factors are evaluated for each variation.
- **light_method** – B-tagging measurement method for light-flavour jets (*"incl"* or *"light"*). Name depends on the naming convention (Run2 vs Run3).
- **heavy_method** – B-tagging measurement method for heavy-flavour jets (*"comb"* or *"mujets"*).
- **syst_prefix** – Prefix to prepend to the name of all resulting the b-tagging systematic variations. Variations for light or heavy jets will be prefixed resp. by *{syst_prefix}light* or *{syst_prefix}heavy* (unless the full scheme is used).
- **decorr_wps** – If *True*, insert the working point into the systematic name for the uncorrelated/statistical component. Otherwise, all working points will be taken as fully correlated when using several in the analysis.
- **decorr_eras** – If *True*, use the scale factor uncertainties split into "uncorrelated" and "correlated" parts, and insert the *era* name into the variation names. If *False*, only use the total scale factor uncertainties (not split).
- **era** – Name of era, used in the name of systematic variations if one of *decorr_eras* or *full_scheme* is *True*.
- **full_scheme** – If *True*, use split uncertainty sources as specified in the *full_scheme* argument
- **syst_mapping** – Dictionary used to list the systematics to consider, and to map the naming of the full-scheme b-tagging uncertainties to variations defined elsewhere in the analysis, for varying them together when needed (see example below).
- **defineOnFirstUse** – see description in *get_correction()*

Returns

a callable that takes a jet and returns the correction (with systematic variations as configured here) obtained by evaluating the b-tagging scale factors on the jet

Example

```
>>> btvSF_b = get_bTagSF_fixWP("btv.json", "deepJet", "M", 5, sel, syst_prefix=
↪"btagSF_",
>>>                            era="2018UL", full_scheme=True,
>>>                            syst_mapping={
>>>                                "pileup": "pileup",
>>>                                "type3": None,
>>>                                "jer0": "jer",
>>>                                "jer1": "jer"
>>>                             })
```

Will result in the following systematic uncertainties:

- *btagSF_statistic_2018UL{up/down}*: mapped to *{up/down}_statistic* in the JSON
- ***btagSF_pileup{up/down}*: mapped to *{up/down}_pileup* in the JSON, and correlated** with the *pileup{up/down}* variations in the analysis
- *btagSF_type3{up/down}*: mapped to *{up/down}_type3* in the JSON
- ***btagSF_jer0{up/down}*: mapped to *{up/down}_jer* in the JSON, and correlated** with the *jer0{up/down}* variations in the analysis
- ***btagSF_jer1{up/down}*: mapped to *{up/down}_jer* in the JSON, and correlated** with the *jer1{up/down}* variations in the analysis

```
>>> btvSF_b = get_bTagSF_fixWP("btv.json", "deepJet", "M", 5, sel, syst_prefix=
↪"btagSF_",
>>>                             era="2018UL", decorr_wps=True, decorr_eras=True)
```

Will result in the following systematic uncertainties:

- *btagSF_heavy_M_2018UL{up/down}*: mapped to *{up/down}_uncorrelated* in the JSON
- *btagSF_heavy{up/down}*: mapped to *{up/down}_correlated* in the JSON

bamboo.scalefactors.**get_bTagSF_itFit**(*json_path*, *tagger_json*, *tagger_jet*, *flavour*, *sel*, *jet_pt_variation=None*, *syst_prefix='btagSF_shape_'*, *decorr_eras=False*, *era=None*, *syst_mapping=None*, *defineOnFirstUse=True*)

Build correction evaluator for continuous (iterativeFit) b-tagging scale factors

Loads the b-tagging scale factors as correction object from the JSON file, configures the systematic variations, and returns a callable that can be evaluated on a jet to return the scale factor.

Parameters

- **json_path** – JSON file path
- **tagger_json** – name of the tagger inside the JSON
- **tagger_jet** – name of the tagger in the tree
- **flavour** – hadron flavour of the jet (*0*, *4*, *5*)
- **sel** – a selection in the current graph (only used to retrieve a pointer to the backend)
- **jet_pt_variation** – see description in *get_bTagSF_fixWP()*
- **syst_prefix** – Prefix to prepend to the name of all resulting the b-tagging systematic variations.
- **decorr_eras** – If *True*, insert the era into the variation name for statistical uncertainties
- **era** – Name of era, used in the name of systematic variations if *decorr_eras* is *True*
- **syst_mapping** – see description in *get_bTagSF_fixWP()*, with the difference that here the "basic" (non-JES-related) variations are already included no matter what.
- **defineOnFirstUse** – see description in *get_correction()*

Returns

a callable that takes a jet and returns the correction (with systematic variations as configured here) obtained by evaluating the b-tagging scale factors on the jet

Example

```
>>> btvSF_b = get_bTagSF_itFit("btv.json", "deepJet", "btagDeepFlavB", 5, sel, syst_
↪prefix="btagSF_",
>>>                             decorr_eras=True, era="2018UL",
>>>                             syst_mapping={"jesTotal": "jes"})
```

Will result in the following systematic uncertainties:

- *btagSF_hfstats1_2018UL{up/down}*: mapped to *{up/down}_hfstats1* in the JSON
- *btagSF_hfstats2_2018UL{up/down}*: mapped to *{up/down}_hfstats2* in the JSON
- *btagSF_lfstats1_2018UL{up/down}*: mapped to *{up/down}_lfstats1* in the JSON
- *btagSF_lfstats2_2018UL{up/down}*: mapped to *{up/down}_lfstats2* in the JSON
- *btagSF_hf{up/down}*: mapped to *{up/down}_hf* in the JSON
- *btagSF_lf{up/down}*: mapped to *{up/down}_lf* in the JSON
- ***btagSF_jesTotal{up/down}*: mapped to *{up/down}_jes* in the JSON, and correlated** with the *jesTotal{up/down}* variations in the analysis

(for c jets, the *hf* and *lf* variations are absent and replaced by *cferr1* and *cferr2*)

bamboo.scalefactors.**get_correction**(*path*, *correction*, *params=None*, *systParam=None*, *systNomName='nominal'*, *systVariations=None*, *systName=None*, *defineOnFirstUse=True*, *sel=None*)

Load a correction from a CMS JSON file

The JSON file is parsed with correctionlib. The contents and structure of a JSON file can be checked with the `correction` script, e.g. `correction summary sf.json`

Parameters

- **path** – JSON file path
- **correction** – name of the correction inside `CorrectionSet` in the JSON file
- **params** – parameter definitions, a dictionary of values or functions
- **systParam** – name of the parameter (category axis) to use for systematic variations
- **systNomName** – name of the systematic category axis to use as nominal
- **systVariations** – systematic variations list or `{variation: name_in_json}`
- **systName** – systematic uncertainty name (to prepend to names, if 'up' and 'down')
- **defineOnFirstUse** – wrap with *defineOnFirstUse()* (to define as a column and reuse afterwards), this is enabled by default since it is usually more efficient
- **sel** – a selection in the current graph (only used to retrieve a pointer to the backend)

Returns

a callable that takes `(object)` and returns the correction (with systematic variations, if present and unless a specific variation is requested) obtained by evaluating the remaining parameters with the object

Example

```
>>> elIDSF = get_correction("EGM_POG_SF_UL.json", "electron_cutbased_looseID",
>>>     params={"pt": lambda el : el.pt, "eta": lambda el : el.eta},
>>>     systParam="weight", systNomName="nominal", systName="elID", systVariations=(
↪"up", "down")
>>>     )
>>> looseEl = op.select(t.Electron, lambda el : el.looseId)
>>> withDiEl = noSel.refine("withDiEl",
>>>     cut=(op.rng_len(looseEl) >= 2),
>>>     weight=[ elIDSF(looseEl[0]), elIDSF(looseEl[1]) ]
>>>     )
```

bamboo.scalefactors.**get_scalefactor**(*objType*, *key*, *combine=None*, *additionalVariables=None*, *sfLib=None*, *paramDefs=None*, *lumiPerPeriod=None*, *periods=None*, *getFlavour=None*, *isElectron=False*, *systName=None*, *seedFun=None*, *defineOnFirstUse=True*)

Construct a scalefactor callable

Warning

This function is deprecated. Use correctionlib and *get_correction()* instead.

Parameters

- **objType** – object type: `"lepton"`, `"dilepton"`, or `"jet"`
- **key** – key in `sfLib` (or tuple of keys, in case of a nested dictionary), or JSON path (or list thereof) if `sfLib` is not specified
- **sfLib** – dictionary (or nested dictionary) of scale factors. A scale factor entry is either a path to a JSON file, or a list of pairs `(periods, path)`, where `periods` is a list of periods found in `lumiPerPeriod` and `path` is a path to the JSON file with the scale factors corresponding to those run periods.
- **combine** – combination strategy for combining different run periods (`"weight"` or `"sample"`)
- **paramDefs** – dictionary of binning variable definitions (name to callable)
- **additionalVariables** – additional binning variable definitions (TODO: remove)
- **lumiPerPeriod** – alternative definitions and relative weights of run periods
- **periods** – Only combine scale factors for those periods
- **isElectron** – if True, will use supercluster eta instead of eta (for `"lepton"` type only) (TODO: find a better way of doing that)
- **systName** – name of the associated systematic nuisance parameter
- **seedFun** – (only when combining scalefactor by sampling) callable to get a random generator seed for an object, e.g. `lambda l : l.idx+42`
- **defineOnFirstUse** – wrap with *defineOnFirstUse()* (to define as a column and reuse afterwards), this is enabled by default since it is usually more efficient

Returns

a callable that takes `(object, variation="Nominal")` and returns a floating-point number proxy

bamboo.scalefactors.**lumiPerPeriod_default = {'Run2016B': 5750.491, 'Run2016C': 2572.903, 'Run2016D': 4242.292, 'Run2016E': 4025.228, 'Run2016F': 3104.509, 'Run2016G': 7575.824, 'Run2016H': 8650.628, 'Run2017B': 4793.97, 'Run2017C': 9632.746, 'Run2017D': 4247.793, 'Run2017E': 9314.581, 'Run2017F': 13539.905, 'Run2018A': 14027.614, 'Run2018B': 7066.552, 'Run2018C': 6898.817, 'Run2018D': 31747.582, 'Run271036to275783': 6274.191, 'Run275784to276500': 3426.131, 'Run276501to276811': 3191.207, 'Run315264to316360': 8928.97, 'Run316361to325175': 50789.746}**

Integrated luminosity (in 1/pb) per data taking period

bamboo.scalefactors.**makeBtagWeightItFit**(*jets*, *sfGetter*)

Construct the full event weight based on b-tagging scale factors (continous/iterativeFit)

Combines the b-tagging scale factors into the event weight needed to correct the simulation (the event weight can then directly be passed to a selection), by making a product of the scale factors over all jets. See the note about correcting the normalization when using these scale factors.

Parameters

- **jets** – the jet collection in the event
- **sfGetter** – a callable that takes the the hadron flavour (*int*) and returns the correction object for the b-tagging scale factors of that jet flavour (i.e., itself a callable that takes the jet and returns the scale factor) See *bamboo.scalefactors.get_bTagSF_itFit()*.

Returns

a weight proxy (with all systematic variations configured in the scale factors)

Example

```
>>> btvSF = lambda flav: get_bTagSF_itFit("btv.json", "deepJet", "btagDeepFlavB",␣
↪flav, ...)
>>> btvWeight = makeBtagWeightItFit(tree.jet, btvSF)
>>> sel_btag = sel.refine("btag", cut=..., weight=btvWeight)
```

bamboo.scalefactors.**makeBtagWeightMeth1a**(*jets*, *tagger*, *wps*, *workingPoints*, *sfGetter*, *effGetters*)

Construct the full event weight based on b-tagging scale factors (fixed working point) and efficiencies

Combines the b-tagging scale factors and MC efficiencies for fixed working points into the event weight needed to correct the simulation (the event weight can then directly be passed to a selection). The weight is computed according to Method 1a, # noqa: B950 with support for several working points.

While the scale factors can be loaded from the POG-provided JSON files, the efficiencies need to be computed by the analyzers.

Parameters

- **jets** – the jet collection in the event
- **tagger** – the name of the tagger in the event
- **wps** – a list of working points for which the b tagging must be corrected, e.g. *["L", "T"]* for computing the weight using scale factors for the *L* and *T* working points. Note: always provide the working points in increasing order of "tightness"!
- **workingPoints** – a dictionary providing the working point value for the discriminator
- **sfGetter** – a callable that takes the working point (*str*) and the hadron flavour (*int*) and returns the correction object for the b-tagging scale factors of that working point and jet flavour (i.e., itself a callable that takes the jet and returns the scale factor). See *bamboo.scalefactors.get_bTagSF_fixWP()*.

- **effGetters** – a dictionary with keys = working points and values = callables that can be evaluated on a jet and return the b-tagging efficiency for that working point. Typically, these would be correction objects parameterized using the jet pt, eta and hadron flavour.

Returns

a weight proxy (with all systematic variations configured in the scale factors)

Example

```
>>> btvSF = lambda wp, flav: get_bTagSF_fixWP("btv.json", "deepJet", wp, flav, ...)
>>> btvEff = {"M": get_correction("my_btag_eff.json", ...)}
>>> btvWeight = makeBtagWeightMeth1a(tree.jet, "btagDeepFlavB", ["M"], {"M": 0.2783}
↪,
>>>                                  btvSF, btvEff)
>>> sel_btag = sel.refine("btag", cut=..., weight=btvWeight)
```

ROOT utilities

The *bamboo.root* module collects a set of thin wrappers around ROOT methods, and centralizes the import of the Cling interpreter global namespace in PyROOT. For compatibility, it is recommended that user code uses `from bamboo.root import gbl` rather than `import ROOT as gbl` or `from cppyy import gbl`.

bamboo.root.**addDynamicPath**(*libPath*)

Add a dynamic library path to the ROOT interpreter

bamboo.root.**addIncludePath**(*incPath*)

Add an include path to the ROOT interpreter

bamboo.root.**findLibrary**(*libName*)

Check if a library can be found, and returns the path in that case

bamboo.root.**loadDependency**(*bambooLib=None*, *includePath=None*, *headers=None*, *dynamicPath=None*, *libraries=None*)

Load a C++ extension

Parameters

- **bambooLib** – name(s) of the bamboo extension libraries, if any
- **includePath** – include directory for headers
- **headers** – headers to load explicitly (which can depend on other headers in the inclue path)
- **dynamicPath** – dynamic library path to add
- **libraries** – additional shared libraries to load

bamboo.root.**loadHeader**(*headerName*)

Include a C++ header in the ROOT interpreter

bamboo.root.**loadLibrary**(*libName*)

Load a shared library in the ROOT interpreter

class bamboo.root.**once**(*fun*)

Function decorator to make sure things are not loaded more than once

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

b

INDEX

Symbols

A

B

H

I

L

M

N

T

U

V

W

Y