plotit

Pieter David, the CP3-CMS team

Jan 09, 2024

CONTENTS

The plotIt tool was developed to efficiently produce large numbers of stack plots that use the same set of samples. It is a standalone C++ executable that is very good at what it does, but it is not very customisable or flexbile: sometimes one "just" wants to get a few histograms to make a specific plot instead of a whole batch, but still take advantage of the information stored in the configuration file and the naming conventions.

This package tries to bridge that gap: it aims to provide a simple python interface to the de-facto file format defined by plotIt: a YAML configuration file and a director of ROOT files with histograms. Basic plotting methods are provided, but they are currently far from supporting all the styling options of plotIt.

CHAPTER

ONE

TABLE OF CONTENTS

1.1 Getting started

1.1.1 Installation

pyplotit is a pure python package, so the latest version can be installed with

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

or, for an editable install when frequent updates and/or testing of changes is expected, with

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

1.1.2 Example: loading histograms from a plotIt configuration

If you do not have a plotIt configuration and the corresponding ROOT files around, you can use the following commands to generate an example; they are also used here for the rest of the example

```
!wget -q https://gitlab.cern.ch/cp3-cms/pyplotit/-/raw/master/tests/data/ex1_syst.yml
!wget -q https://raw.githubusercontent.com/cp3-llbb/plotIt/master/test/generate_files.C
!mkdir -p files
!root -l -b -q generate_files.C
```

```
Processing generate_files.C...
```

We can load the configuration file `ex1_syst.yml` in pyplotit as follows:

```
import plotit
config, samples, plots, systematics, legend = plotit.loadFromYAML("ex1_syst.yml")
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[2], line 1
----> 1 import plotit
      2 config, samples, plots, systematics, legend = plotit.loadFromYAML("ex1_syst.yml")

File ~/checkouts/readthedocs.org/user_builds/pyplotit/conda/latest/lib/python3.11/site-
↪packages/plotit/__init__.py:1
```

(continues on next page)

(continued from previous page)

```
----> 1 from .plotit import loadFromYAML
      2 from .version import version as __version__
      4 __all__ = ("__version__", "loadFromYAML")

File ~/checkouts/readthedocs.org/user_builds/pyplotit/conda/latest/lib/python3.11/site-
↪packages/plotit/plotit.py:33
     29 import numpy as np
     31 from uhi.typing.plottable import PlottableAxisGeneric, PlottableHistogram,␣
↪PlottableTraits
---> 33 from . import config
     34 from . import histo_utils as h1u
     35 from .logging import logger

File ~/checkouts/readthedocs.org/user_builds/pyplotit/conda/latest/lib/python3.11/site-
↪packages/plotit/config.py:285
    272         return cfg
    274     # def __post_init__(self):
    275     #     if self.x_axis_range is not None:
    276     #         try:
   (...)
    281     #             raise ValueError("Could not parse x-axis-range {0}: {1}".
↪format(self.x_axis_range, e))
    282     #         self.x_axis_range = lims
--> 285 @dataclass
    286 class Legend(BaseConfigObject):
    287     position: Position = Position(x1=0.6, y1=0.6, x2=0.9, y2=0.9)
    288     columns: int = 1

File ~/checkouts/readthedocs.org/user_builds/pyplotit/conda/latest/lib/python3.11/
↪dataclasses.py:1230, in dataclass(cls, init, repr, eq, order, unsafe_hash, frozen,␣
↪match_args, kw_only, slots, weakref_slot)
   1227     return wrap
   1229 # We're called as @dataclass without parens.
-> 1230 return wrap(cls)

File ~/checkouts/readthedocs.org/user_builds/pyplotit/conda/latest/lib/python3.11/
↪dataclasses.py:1220, in dataclass.<locals>.wrap(cls)
   1219 def wrap(cls):
-> 1220     return _process_class(cls, init, repr, eq, order, unsafe_hash,
   1221                           frozen, match_args, kw_only, slots,
   1222                           weakref_slot)

File ~/checkouts/readthedocs.org/user_builds/pyplotit/conda/latest/lib/python3.11/
↪dataclasses.py:958, in _process_class(cls, init, repr, eq, order, unsafe_hash, frozen,␣
↪match_args, kw_only, slots, weakref_slot)
    955         kw_only = True
    956     else:
    957         # Otherwise it's a field of some type.
--> 958         cls_fields.append(_get_field(cls, name, type, kw_only))
    960 for f in cls_fields:
    961     fields[f.name] = f
```

(continues on next page)

(continued from previous page)

```
File ~/checkouts/readthedocs.org/user_builds/pyplotit/conda/latest/lib/python3.11/
↪dataclasses.py:815, in _get_field(cls, a_name, a_type, default_kw_only)
    811 # For real fields, disallow mutable defaults.  Use unhashable as a proxy
    812 # indicator for mutability.  Read the __hash__ attribute from the class,
    813 # not the instance.
    814 if f._field_type is _FIELD and f.default.__class__.__hash__ is None:
--> 815     raise ValueError(f'mutable default {type(f.default)} for field '
    816                      f'{f.name} is not allowed: use default_factory')
    818 return f

ValueError: mutable default <class 'plotit.config.Position'> for field position is not␣
↪allowed: use default_factory
```

Most of the returned objects are either (lists of) simple objects that represent a part of the configuration, e.g. a single plot. The classes are implemented as data classes. The list returned in `samples` is based on the entries in the `files` block of the configuration file, but using the grouping specified by their `group` attributes and the list of groups, such that each entry corresponds to a visible contribution in the plots.

Since the `File` and `Group` classes also contain functionality for the efficient loading and summing of the histograms, the pure configuration part is kept in a separate class (also a data class), under the `cfg` attribute. For groups the list of grouped files can be found under `files`.

```
[smp.cfg for smp in samples]
```

Typical plots contain an observed histogram and expectation stack. Since the former may be the sum of multiple datasets, it is also handled as a stack:

```
p = plots[0]
from plotit.plotit import Stack
expStack = Stack([smp.getHist(p) for smp in samples if smp.cfg.type == "MC"])
obsStack = Stack([smp.getHist(p) for smp in samples if smp.cfg.type == "DATA"])
```

The above works because both the `File` and `Group` class have a `getHist` method, which loads a single histogram from a file, or triggers the loading of multiple histograms and adds them up, respectively.

`getHist` returns a small object similar to a smart pointer: for a single file it holds the pointer to the (Py)ROOT histogram, for a group of stack it lazily constructs the sum histogram, or adds up the contents and squared weights arrays, depending on which method is called (more details will be added once the interfaces are more stable). These smart pointer or histogram handle classes also implement the uhi `PlottableHistogram` protocol, so they can directly be used with e.g. mplhep:

```
from matplotlib import pyplot as plt
fig, ax = plt.subplots()
ax.set_xlim(*p.x_axis_range)
import mplhep
mplhep.histplot(obsStack, histtype="errorbar", color="k")
mplhep.histplot(expStack.entries, stack=True, histtype="fill", color=[e.style.fill_color␣
↪for e in expStack.entries])
ax.set_xlabel(p.x_axis, loc="right")
ax.set_ylabel(p.y_axis, loc="top")
mplhep.cms.label(data=True, label="Internal", lumi=config.getLumi())
```

1.2 Overview

Two command-line scripts are provided: `iPlotIt` and `pyPlotIt`. Both have a similar interface as the `plotIt` executable: they take a YAML configuration file as a positional argument, and optional `--histodir` and `--eras` arguments, to pass a different histograms directory (in case they are not in the same directory as the configuration file) and set of data-taking periods (eras) to consider. `pyPlotIt` mimics the `plotIt` batch plot production, but is currently not very useful, given the much more limited support for styling options.

`iPlotIt` is the best place to get started: it loads a configuration file and then opens an IPython shell to inspect it, and interactively load and manipulate histograms. Usually it can be used as

```
iPlotIt plots.yml
```

The available objects are:

- `config`, the `Configuration` object corresponding to the top level of the YAML file (excluding the sections that are parsed separately)
- `samples`, a list of `Group` or ungrouped `File` objects (stateful, see below), which correspond to the `groups` and `files` sections of the configuration file and can be used to retrieve the histograms for a plot
- `plots`, a list of `Plot` objects, which corresponds to the `plots` section of the configuration file
- `systematics`, a list of systematic uncertainties (`SystVar` objects), which corresponds to the `systematics` section of the configuration file
- `legend`, the parsed `legend` section, with the list of entries

From a script the same objects can be obtained by calling the `loadFromYAML()` method. There is one difference: this method returns a list of plots, whereas `iPlotIt` provides a dictionary where each plot is stored with its `name` attribute as a key—so they are equivalent, the latter is only done for convenience.

Each file contains a histogram (possibly with systematic variations) for every plot. These are combined in groups if the file belongs to a group, or directly added as a contribution to a stack in the plot. The following example illustrates how to retrieve the histograms, and construct the expected and observed stacks for a plot:

```
mcSamples = [smp for smp in samples if smp.cfg.type == "MC"]
dataSamples = [smp for smp in samples if smp.cfg.type == "DATA"]
expStack = Stack(entries=[smp.getHist(plot) for smp in mcSamples])
obsStack = Stack(entries=[smp.getHist(plot) for smp in dataSamples])
```

The drawing of the stacks depends on the type: for MC the contributions, which can be accessed as `expStack.entries` are usually drawn stacked in different colours; for data only the sum is drawn. The `getHist` method of the samples returns a `FileHist` for `File` or a `GroupHist` for `Group`, which are a smart pointer to a `TH1F` object or the on-demand constructed sum of them for the different files in the group, respectively. These are described in more detail in the next section.

1.3 Architecture

This package was designed to potentially replace plotIt in the long run, so a few design choices were made with performance in mind, and others slightly over-engineered to provide maximal flexibility for future development. The two main distinctions to keep in mind are between configuration and stateful classes, and between raw histogram pointers and smart pointers.

The former is relatively straightforward, but causes some duplication: the configuration file is initially parsed to classes that represent the configuration, but carry no additional state; they are essentially the dictionaries from the YAML

parsing, but with some additional structure based on the type information. For many things this is sufficient, but for loading histograms from files the files need to be opened, and for efficiency a pointer to the open file should be stored. This is why stateful `File` and `Group` classes exist in `plotit.plotit`, which carry the configuration-only part as their `cfg` attribute.

Smart histogram pointers are introduced for performance reasons: the most time-consuming part of running `plotIt` in practice is opening ROOT files and retrieving histograms (this can be hundreds of histograms spread out over dozens of files for a single plot, with typical runs producing hundreds of plots), and these histograms are also what drives the memory usage when producing histograms in batch mode. The `FileHist` class allows to control when histograms are read from the file: it provides a handle to the histogram, but postpones loading it from disk until the contents is first accessed. It is also possible to force loading and unloading the `TH1` objects, which allows a simple implementation of the strategy adopted by `plotIt`, where all histograms needed for a set of plots are loaded from each file in one go, and cleaned up after the plots are produced.

`FileHist` is part of a class hierarchy, with `BaseHist` defining the common interface and basic functionality, and `MemHist` and `SumHist` implementing the same interface as `FileHist` for histograms that are not loaded from a file and groups of histograms that should be added, respectively. `Stack` is an extension of `SumHist` that represents a stack of groups and files. The common interface provides direct access to the `TH1` objects, as well as access to the contents and `sumw2` arrays as NumPy arrays, which allows to adopt a very pythonic style for implementing custom plots or other scripts.

1.4 Reference

1.4.1 YAML configuration parsing

A plotIt YAML configuration file should have the following structure:

```yaml
configuration:
  # Configuration block
files:
  file_name:
    # File block
  ...
groups: # optional
  group_name:
    # Group block
  ...
plots:
  plot_name:
    # Plot block
  ...
systematics: # optional
  # just name (for shape) or name with systematic block
  ...
legend: # optional
  # legend block
```

Such YAML files can be parsed with the `loadFromYAML()` method. It will return instances of the classes defined in the `plotit.config` module, whose attribute listings below serve as a reference of the allowed attributes in each block, and their types.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search