Pocket Coffea [2]

Pocket coffea is a coffea-based analysis framework for CMS NanoAOD events.

The objective of this framework is to define the analysis in a declarative manner, rather than writing the analysis code from scratch, defining what you want to do (selections, histograms, etc.) in a configuration file. In simple terms, the framework takes care of the how.

In turn, this framework allows you to work with more complex or specific tasks that cannot be defined in the configuration by writing your own Python code, thus creating new Coffea "processors."

The structured workflow of Pocket Coffea provides a 'recipe' or sequence of standard operations ("BaseProcessor") to go from NanoAOD data to final histograms. This recipe includes steps such as applying weights, defining categories, and filling histograms, which can be modified according to user needs.

One of the advantages of using this framework is that it clearly separates the physical definition of the analysis (parameters, selections) from the execution configuration (which samples to process, in which cluster, etc), making the analysis more orderly and reproducible.

Installation

Pocket Coffea can be installed from pip or directly from the sources [2].

From source, you need to follow the next instrucions.

1. Clone the next repository in your ssh server, if you do this in your EOS space, it must be done in your working folder - /eos/home-(initial user, i.e. “s”)/user(i.e. “sarafer”)/PocketCoffea:

   `git clone https://github.com/PocketCoffea/PocketCoffea.git`

2. Create your environment with the necessary libraries:

```
cd PocketCoffea
python -m venv myenv
source myenv/bin/activate

#-e installs it in "editable" mode so that the modified files are included dynami
cally in the package.
pip install -e .
```

Example Analysis

text about the analysis example.

Configuration Files

The configuration files, are Python files (`.py`) that must contain a **dictionary** called `cfg`. Within this `cfg` dictionary, all the details of the analysis (selections, histograms, samples, etc.) must be specified.

Finally, the framework module called **`Configurator`** must be configured, as it will be responsible for reading and processing this dictionary (cfg) to prepare and execute the analysis.

Now downolading the example configuration:

```
git clone https://github.com/PocketCoffea/AnalysisConfigs.git
```

```
cd AnalysisConfigs/configs/zmumu
```

Ready to Start!

1. Dataset Preparation: Build datasets metadata.

For the datasets of the analysis, we should take the corresponding DAS keys:

```
https://cmsweb.cern.ch/das/request?view=list&limit=50&instance=prod%2Fglobal&input=%2FDYJetsToLL_M-50_TuneCP5_13TeV-amcatnloFXFX-pythia8%2FRunIISummer20UL18NanoAODv9-106X_upgrade2018_realistic_v16_L1v1-v2%2FNANOAODSIM
```

- `/DYJetsToLL_M-50_TuneCP5_13TeV-amcatnloFXFX-pythia8/RunIISummer20UL18NanoAODv9-106X_upgrade2018_realistic_v16_L1v1-v2/NANOAODSIM`

```
https://cmsweb.cern.ch/das/request?view=list&limit=50&instance=prod%2Fglobal&input=%2FSingleMuon%2FRun2018C-UL2018_MiniAODv2_NanoAODv9-v2%2FNANOAOD
```

- `/SingleMuon/Run2018C-UL2018_MiniAODv2_NanoAODv9-v2/NANOAOD`

The workflow begins by creating a JSON configuration file, where each dataset is defined with the corresponding metadata, such as year, cross-section, and era, in a structured format.

Once the JSON file is ready, the `build-datasets` command is executed. This command reads the definition file (JSON), processes each entry, and generates the final JSON files (known as **“filesets”**), which contain the complete lists of file paths that the analysis will use directly as input to the Coffea processor [3].

```
{
  "DYJetsToLL_M-50":{
      "sample": "DYJetsToLL",
      "json_output"    : "datasets/DYJetsToLL_M-50.json",
      "files":[
         { "das_names": ["/DYJetsToLL_M-50_TuneCP5_13TeV-amcatnloFXFX-p
ythia8/RunIISummer20UL18NanoAODv9-106X_upgrade2018_realistic_v16_L1v1
-v2/NANOAODSIM"],
           "metadata": {
              "year":"2018",
              "isMC": true,
              "xsec": 6077.22,
              "part": ""
              }
          }
      ]
  },
   "DATA_SingleMuon": {
      "sample": "DATA_SingleMuon",
      "json_output": "datasets/DATA_SingleMuon.json",
      "files": [
          {
            "das_names": [
               "/SingleMuon/Run2018A-UL2018_MiniAODv2_NanoAODv9-v2/NA
NOAOD"
            ],
            "metadata": {
               "year": "2018",
               "isMC": false,
            "primaryDataset": "SingleMuon",
            "era": "A"
          },
          "das_parents_names": [
            "/SingleMuon/Run2018A-UL2018_MiniAODv2-v3/MINIAOD"
          ]
        },
        {
          "das_names": [
            "/SingleMuon/Run2018B-UL2018_MiniAODv2_NanoAODv9-v2/NA
NOAOD"
          ],
          "metadata": {
            "year": "2018",
            "isMC": false,
            "primaryDataset": "SingleMuon",
            "era": "B"
          },
          "das_parents_names": [
            "/SingleMuon/Run2018B-UL2018_MiniAODv2-v2/MINIAOD"
          ]
        },
        {
          "das_names": [
            "/SingleMuon/Run2018C-UL2018_MiniAODv2_NanoAODv9-v2/NA
NOAOD"
          ],
          "metadata": {
            "year": "2018",
            "isMC": false,
            "primaryDataset": "SingleMuon",
            "era": "C"
          },
          "das_parents_names": [
            "/SingleMuon/Run2018C-UL2018_MiniAODv2-v2/MINIAOD"
          ]
```

```
            },
            {....}
        ]
    }
}
```

The JSON file `datasets_definitions.json` is a dictionary where each entry defines a dataset with the following main keys:

- `dataset`: The unique identifier of the dataset.
- `sample`: This is the label used internally in the framework to categorize the type of events contained in the dataset.
- `json_output`: The path where the final JSON file with the list of files will be saved.
- `files`: List that allows you to combine several DAS datasets, where each element of the list contains:
    1. `das_names`: The list of dataset names in DAS.
    2. `metadata`: Dictionary with specific information about the sample:

       MC: year, isMC, xsec

       Data: year, isMC, era, primaryDataset)

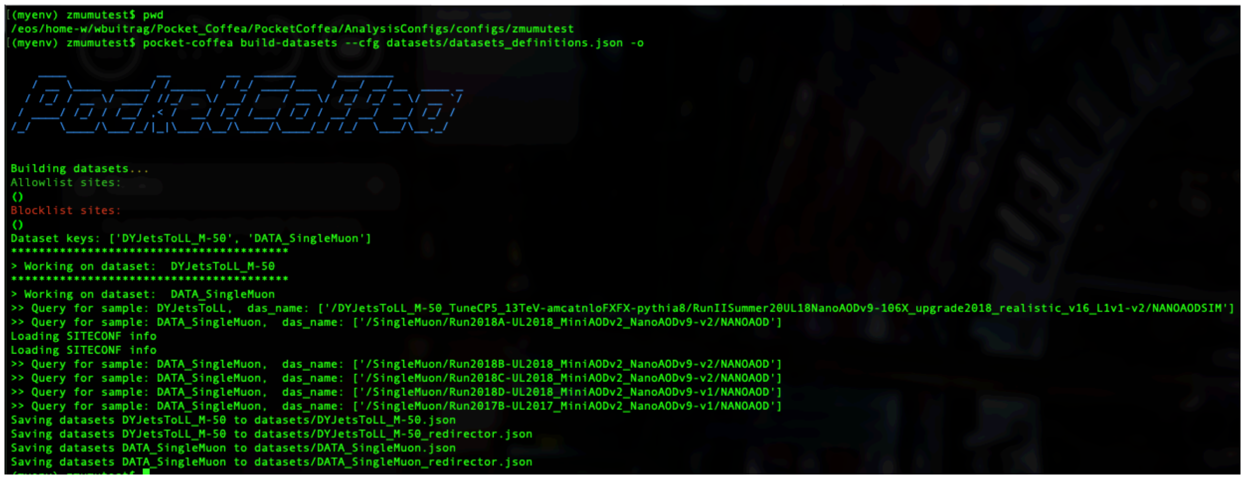
When datasets are built, metadata parameters are directly associated with the corresponding file list. This creates a unique and complete dataset entry.

For Data datasets, the `primaryDataset` key is essential for trigger selection, as it allows the analysis to apply a specific set of triggers only to the corresponding primary dataset.

(e.g. apply the `SingleMuon` trigger only to datasets having `primaryDataset=SingleMuon`) [3]

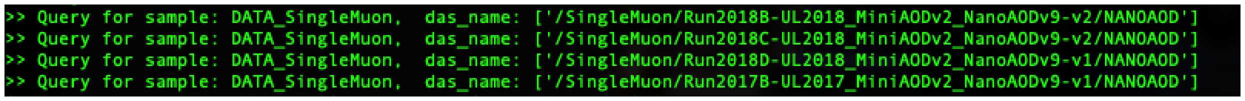
To produce the json files containing the file list:

```
# if you are using singularity, create the ticket outside of it.
voms-proxy-init -voms cms -rfc --valid 168:0

pocket-coffea build-datasets --cfg datasets/datasets_definitions.json -o

# if you are running at CERN it is useful to restrict the data sources to Tiers cl
oser to CERN
pocket-coffea build-datasets --cfg datasets/datasets_definitions.json -o -rs 'T
[123]_(FR|IT|BE|CH|DE)_\w+'
```



```
(myenv) zmumutest$ pwd
/eos/home-w/wbuitrag/Pocket_Coffea/PocketCoffea/AnalysisConfigs/configs/zmumutest
(myenv) zmumutest$ pocket-coffea build-datasets --cfg datasets/datasets_definitions.json -o

Building datasets...
Allowlist sites:
()
Blocklist sites:
()
Dataset keys: ['DYJetsToLL_M-50', 'DATA_SingleMuon']
****************************************
> Working on dataset:  DYJetsToLL_M-50
****************************************
> Working on dataset:  DATA_SingleMuon
>> Query for sample: DYJetsToLL,  das_name: ['/DYJetsToLL_M-50_TuneCP5_13TeV-amcatnloFXFX-pythia8/RunIISummer20UL18NanoAODv9-106X_upgrade2018_realistic_v16_L1v1-v2/NANOAODSIM']
>> Query for sample: DATA_SingleMuon,  das_name: ['/SingleMuon/Run2018A-UL2018_MiniAODv2_NanoAODv9-v2/NANOAOD']
Loading SITECONF info
Loading SITECONF info
>> Query for sample: DATA_SingleMuon,  das_name: ['/SingleMuon/Run2018B-UL2018_MiniAODv2_NanoAODv9-v2/NANOAOD']
>> Query for sample: DATA_SingleMuon,  das_name: ['/SingleMuon/Run2018C-UL2018_MiniAODv2_NanoAODv9-v2/NANOAOD']
>> Query for sample: DATA_SingleMuon,  das_name: ['/SingleMuon/Run2018D-UL2018_MiniAODv2_NanoAODv9-v1/NANOAOD']
>> Query for sample: DATA_SingleMuon,  das_name: ['/SingleMuon/Run2017B-UL2017_MiniAODv2_NanoAODv9-v1/NANOAOD']
Saving datasets DYJetsToLL_M-50 to datasets/DYJetsToLL_M-50.json
Saving datasets DYJetsToLL_M-50 to datasets/DYJetsToLL_M-50_redirector.json
Saving datasets DATA_SingleMuon to datasets/DATA_SingleMuon.json
Saving datasets DATA_SingleMuon to datasets/DATA_SingleMuon_redirector.json
```

Two output files are produced for each dataset:



```
>> Query for sample: DATA_SingleMuon,  das_name: ['/SingleMuon/Run2018B-UL2018_MiniAODv2_NanoAODv9-v2/NANOAOD']
>> Query for sample: DATA_SingleMuon,  das_name: ['/SingleMuon/Run2018C-UL2018_MiniAODv2_NanoAODv9-v2/NANOAOD']
>> Query for sample: DATA_SingleMuon,  das_name: ['/SingleMuon/Run2018D-UL2018_MiniAODv2_NanoAODv9-v1/NANOAOD']
>> Query for sample: DATA_SingleMuon,  das_name: ['/SingleMuon/Run2017B-UL2017_MiniAODv2_NanoAODv9-v1/NANOAOD']
```

The first output file contains all sample paths with a prefix pointing to a specific location on the Grid. The script determines the best available location by consulting **Rucio** (the CMS data management system).

The second output file contains the paths with a global redirector (e.g. `xrootd-cms.infn.it`). This file is identified by the suffix `_redirector.json`.

Note: If you need to add more datasets or update existing ones, you must use the argument `--overwrite`. This option tells the script to **overwrite** the old JSON files with the new information.

Analysis Configuration

Parameters [4]

PocketCoffea distinguishes between parameters for the analysis and specific runs configuration.

- *A parameter* is considered something defining more in general a specific analysis and it is usually common between different runs: e.g. HLT triggers, scale factors working points, JES/JER configuration.
- A configuration is considered specific for a run of the analysis where the users applied selections, cleans objects and output histograms or arrays.

TIP
Parameters are handled as
yaml files with the OmegaConf utility library. A set of common and defaults are defined centrally by PocketCoffea and then the user can overwrite, replace and add analysis specific configurations. Have a look at the Parameters page.[4]

This is done usually in the preamble of the analysis config file (**file config.py**):

```
import os
localdir = os.path.dirname(os.path.abspath(__file__))

# Loading default parameters
from pocket_coffea.parameters import defaults
default_parameters = defaults.get_default_parameters()
```

```
defaults.register_configuration_dir("config_dir", localdir+"/params")

# Here we call the parameters that we defined in the yaml files.
parameters = defaults.merge_parameters_from_files(default_parameters,
                                                  f"{localdir}/params/object_preselection.yaml",
                                                  f"{localdir}/params/triggers.yaml",
                                                  update=True)
```

The `parameters` object can be freely modified to adjust the configuration. Once customized, this object is passed to the `Configurator` class, which uses it to configure the analysis.

A copy of the exact **parameters** you used is **saved along with the results** of your analysis (such as histograms). This is essential for **reproducibility**, as it allows you to know precisely which settings generated each result.

Configuration

A specific analysis *run* is defined in PocketCoffea using an instance of a `Configurator` code class. This class groups all the information about skimming, categorization, datasets, and outputs.

Zmumu Analysis

Configuration

The configurator instance is created inside the main configuration file `example_config.py` and assied to a variable called `cfg`. This special name is used by the framework when the file is passed to the `pocket-coffea run` script to be executed.

```
from pocket_coffea.utils.configurator import Configurator
from pocket_coffea.lib.cut_definition import Cut
from pocket_coffea.lib.cut_functions import get_nObj_min, get_HLTsel
from pocket_coffea.parameters.cuts import passthrough
from pocket_coffea.parameters.histograms import *
from workflow import ZmumuBaseProcessor
from custom_cut_functions import *


cfg = Configurator(
  parameters = parameters,
  datasets = {
    "jsons": [f"{localdir}/datasets/DATA_SingleMuon.json",
          f"{localdir}/datasets/DYJetsToLL_M-50.json"
           ],
    "filter" : {
      "samples": ["DATA_SingleMuon",
             "DYJetsToLL"],
      "samples_exclude" : [],
      "year": ['2018']
    }
  },
  #.....continues
  workflow = ZmumuBaseProcessor,
```

Datasets are specified by passing the list of json to be used and they can be filtered by year of sample type.

Parameters are also passed to the Configurator as well as the **workflow** class to be used in the processing. This class is user defined in the `workflow.py` file locally in the folder.

Define Selections

The selections are performed at two levels:

- **Object Preselection:** Selecting the “good” objects that will be used in the final analysis. (e.g. `JetGood`, `MuonGood`, `ElectronGood` , etc).
- **Event Selection:** Selections on the events that enter the final analysis, done in three steps:

  1. Skim and Trigger: Loose cut on the events and trigger requirements.

  2. Preselection: Baseline event selection for the analysis.

  3. Categorization: Selection to split the events passing the event preselection into different categories (e.g. signal region, control region),

Object Preselection

To select the objects entering the final analysis, we need to spicify a series of cut parameters for the leptons and jets, in the file `params/object_preselection.yaml`. These selections include the P_T, η, acceptance cuts, the object identification working points, the muon isolation, the b-tagging working point, etc.

For the $Z \to \mu\mu$ analysis, we just use the standard definitions for the muon, electron and jet objects:

```
object_preselection:
  Muon:
    pt: 15
    eta: 2.4
    iso: 0.25  #PFIsoLoose
    id: tightId

  Electron:
    pt: 15
    eta: 2.4
    iso: 0.06
    id: mvaFall17V2Iso_WP80

  Jet:
    dr_lepton: 0.4
    pt: 30
    eta: 2.4
    jetId: 2
    puId:
      wp: L
      value: 4
      maxpt: 50.0
    btag:
      wp: M
```

This parameters are used by the functions which filters the object collections in the `workflow.py` file.

Event Selection

In PocketCoffea, the event selections are implemented with the dedicated `Cut` object, that stores both the information of the cutting function and its input parameters. Several factory `Cut` objects are available in `pocket_coffea.lib.cut_functions`, otherwise the user can define their own custom `Cut` objects.

Skim

The skim selection of the event is performed “on the fly” to reduce the amount of data to be processed, discarding those that are not of interest from the beginning.

At this stage we also apply the HLT trigger requirements for the analysis. The following steps of the analysis are performed only on the events passing the skim selection, while the others are discarded from the branch `events`, therefore reducing the computational load on the processor.

In the **config file**, we specify two skim cuts:

One is selecting the events with at least one 18 GeV muon and the second is requiring the HLT `SingleMuon` path.

Triggers are specified in a parameter yaml files under the `params` dir (but the location is up to the user). The parameters are then loaded and added to the default parameters in in the preamble of `example_config.py`, which we pass as an argument to the factory function `get_HLTsel()`.

Moreover, standard selections as the following should be applied at the skim stage:

- event flags for MC and Data
- number of primary vertex selection
- application of the golden JSON selection to Data lumisections

```
from pocket_coffea.lib.cut_functions import get_nObj_min, get_HLTsel, eventFlags, get_nPVgood, goldenJson

cfg = Configurator(
   # .....

   skim = [
         eventFlags,
         goldenJson,
         get_nPVgood(1),
         get_nObj_min(1, 18., "Muon"),
         # Asking only SingleMuon triggers since we are only using SingleMuon
PD data
        get_HLTsel(primaryDatasets=["SingleMuon"])],
```

Event Preselection

In the $Z \to \mu\mu$ analysis, we want to select events with exactly two muons with opposite charge. In addition, we requiere a cut on the leading muon pt and on the dilepton invariant mas, to select the Z boson mass window. The parameters are directly passed to the constructor of the `Cut` object as the dictionary `params`.

We can define the function `dimuon` and the `Cut` object `dimuon_presel` in the preamble of the config script:

```
def dimuon(events, params, year, sample, **kwargs):
  # Masks for same-flavor (SF) and opposite-sign (OS)
  SF = ((events.nMuonGood == 2) & (events.nElectronGood == 0))
  OS = events.ll.charge == 0$
  mask = (
    (events.nLeptonGood == 2)
    & (ak.firsts(events.MuonGood.pt) > params["pt_leading_muon"])
    & OS & SF
    & (events.ll.mass > params["mll"]["low"])
    & (events.ll.mass < params["mll"]["high"])
  )

  # Pad None values with False
  return ak.where(ak.is_none(mask), False, mask)

dimuon_presel = Cut(
  name="dilepton",
  params={
    "pt_leading_muon": 25,
        "mll": {'low': 25, 'high': 2000},
    },
    function=dimuon,
)
```

In a scenario of an analysis requiring several different cuts, a dedicated library of cuts and functions can be defined in a separate file and imported in the config file. This is the strategy implemented in the example (and recommended to keep the config file minimal). Have a look at the `custom_cut_functions.py` file.

The `preselections` field in the config file is updated accordingly:

```
cfg = Configurator(
    # .....

    preselections = [dimuon_presel],
```

Categorization

In the toy $Z \to \mu\mu$ analysis, no further categorization of the events is performed. Only a `baseline` category is defined with the `passthrough` factory cut that is just passing the events through without any further selection:

```
cfg = Configurator(
    # .....

    categories = {
        "baseline": [passthrough],
    },
```

If for example $Z \to ee$ events were also included in the analysis, one could have defined a more general "dilepton" preselection and categorized the events as `2e` or `2mu` depending if they contain two electrons or two muons, respectively.

Define weights and variations

The application of the nominal value of scale factors and weights is switched on and off just by adding the corresponding key in the `weights` dictionary:

```
from pocket_coffea.lib.weights.common import common_weights

cfg = Configurator(
    # .....
    weights_classes = common_weights,  # optional

    weights = {
        "common": {
           "inclusive": ["genWeight","lumi","XS",
                         "pileup",
                         "sf_mu_id","sf_mu_iso",
                         ],
           "bycategory" : {
           }
        },
        "bysample": {
        }
     }
    )
```

In our case, we are applying the nominal scaling of Monte Carlo by `lumi * XS / genWeight` together with the pileup reweighting and the muon ID and isolation scale factors. The reweighting of the events is managed by the module `WeightsManager`.

The available weights are defined by the `weights_classes` argument of the Configurator. That is a list of WeightWrapper objects that can be taken from the ones defined in the framework, or it can be customized by the users. The common weights defined by the framework are available in `pocket_coffea.lib.weights.common.common_weights`. If the user doesn't provide a list of weight classes, the common ones are used by default.

To store also the up and down systematic variations corresponding to a given weight, one can specify it in the `variations` dictionary:

```
cfg = Configurator(
    # .....
    variations = {
        "weights": {
            "common": {
                "inclusive": [  "pileup",
                                "sf_mu_id", "sf_mu_iso"
                              ],
                "bycategory" : {
                }
            },
        "bysample": {
        }
        },
    },
```

In this case we will store the variations corresponding to the systematic variation of pileup and the muon ID and isolation scale factors. These systematic uncertainties will be included in the final plots.

Define Histograms

In PocketCoffea histograms can be defined directly from the configuration: look at the `variable` dictionary under `example_config.py`. Histograms are defined with the options specified in **`hist_manager.py`**. An histogram is a collection of `Axis` objects with additional options for excluding/including variables, samples, and systematic variations.

In order to create a user defined histogram add `Axis` as a list (1 element for 1D-hist, 2 elements for 2D-hist)

```
cfg = Configurator(
   # .....

  variables : {

        # 1D plots
        "mll" : HistConf( [Axis(coll="ll", field="mass", bins=100, start=50, stop=
150, label=r"$M_{\ell\ell}$ [GeV]")]
   },

   # coll : collection/objects under events
   # field: fields under collections
   # bins, start, stop: # bins, axis-min, axis-max
   # label: axis label name
```

The `collection` is the name used to access the fields of the `events` main dataset (the NanoAOD Events tree). The `field` specifies the specific array to use. If a field is global in the `events`, e.g. a user defined arrays in events, just use `coll=events`. Please refer to the `Axis` code for a complete description of the options.

- There are some predefined
`hist` dictionaries ready to be built for the user:

```
cfg = Configurator(
   # .....

  variables : {
      **count_hist(name="nJets", coll="JetGood",bins=8, start=0, stop=8),
      # Muon kinematics
      **muon_hists(coll="MuonGood", pos=0),
      # Jet kinematics
      **jet_hists(coll="JetGood", pos=0),
   },
```

Run the Processor

Run the coffea processor to get `.coffea` output files. The `coffea` processor can be run locally or be scaled out to clusters:

- `iterative` execution runs single thread locally and it useful for debugging
- `futures` execute the processor in multiple threads locally and it can be useful for fast processing of a small amount of file.
- `dask@lxplus` uses the dask scheduler with lxplus the configuration to send out workers on HTCondor jobs.

The executors are configured in `pocket_coffea/executors` module for a set of predefined grid-sites. Please get in contact with the developers and open a PR if you want to include your specific site to the supported list.

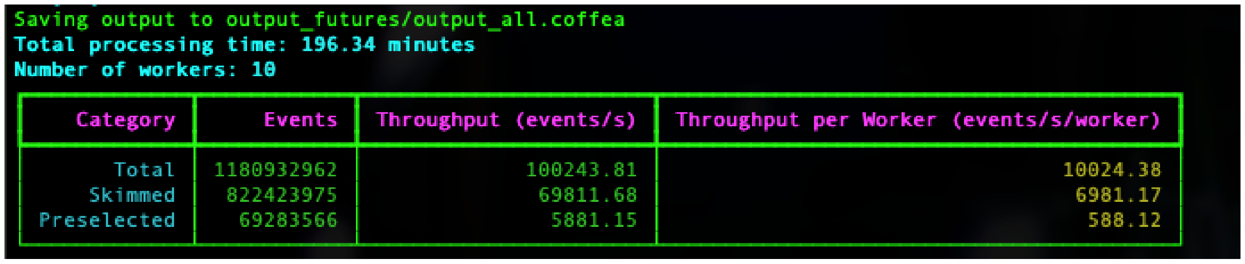
In this tutorial we **assume the use of lxplus**, but the example should work fine also on other sites with the `iterative` or `futures` setup. The Dask scheduler execution needs to be configured properly to send out jobs in your facility.

```
## First let's test the running locally with  --test for iterative processor with ``--limit-chunks/-lc``(default:2) and ``--limit-files/-lf``(default:1)
pocket-coffea run --cfg example_config.py --test --process-separately  -o output_test
```

```
pocket-coffea run --cfg example_config.py  --executor dask@lxplus  --scaleout 10  -o output_dask
```

```
# physical nodes in your machine

pocket-coffea run --cfg example_config.py --executor futures --scaleout 10 -o output_futures
```

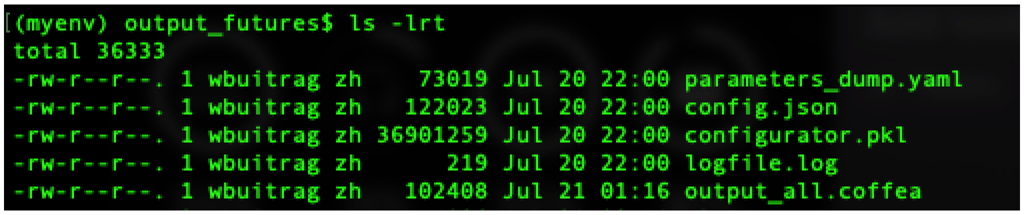
```
# To run with custom options

pocket-coffea run --cfg example_config.py --executor dask@lxplus --custom-run-options custom_run_options.yaml  -o output_dask
```



```
Saving output to output_futures/output_all.coffea
Total processing time: 196.34 minutes
Number of workers: 10

| Category    | Events     | Throughput (events/s) | Throughput per Worker (events/s/worker) |
|       Total | 1180932962 |             100243.81 |                                10024.38 |
|     Skimmed |  822423975 |              69811.68 |                                 6981.17 |
| Preselected |   69283566 |               5881.15 |                                  588.12 |
```

Produce Plots

Once the processing is done, the output folder looks like



```
(myenv) output_futures$ ls -lrt
total 36333
-rw-r--r--. 1 wbuitrag zh    73019 Jul 20 22:00 parameters_dump.yaml
-rw-r--r--. 1 wbuitrag zh   122023 Jul 20 22:00 config.json
-rw-r--r--. 1 wbuitrag zh 36901259 Jul 20 22:00 configurator.pkl
-rw-r--r--. 1 wbuitrag zh      219 Jul 20 22:00 logfile.log
-rw-r--r--. 1 wbuitrag zh   102408 Jul 21 01:16 output_all.coffea
```

Along with the analysis results (such as histograms), **three configuration files** are saved to ensure reproducibility and facilitate post-processing.

1. **Parameters:** A copy of all **parameters** used (those from the `.yaml` files) is saved.
2. **JSON configuration:** A version of the main configuration file is saved in **JSON** format, which is easy for the user to read.
3. **`Configurator` object (`cloudpickle`):** The entire `Configurator` object is saved in a binary format called `cloudpickle`. This file is "machine-readable" and is crucial for post-processing (e.g., for making graphs), as it allows you to reload the entire analysis configuration into another Python script.

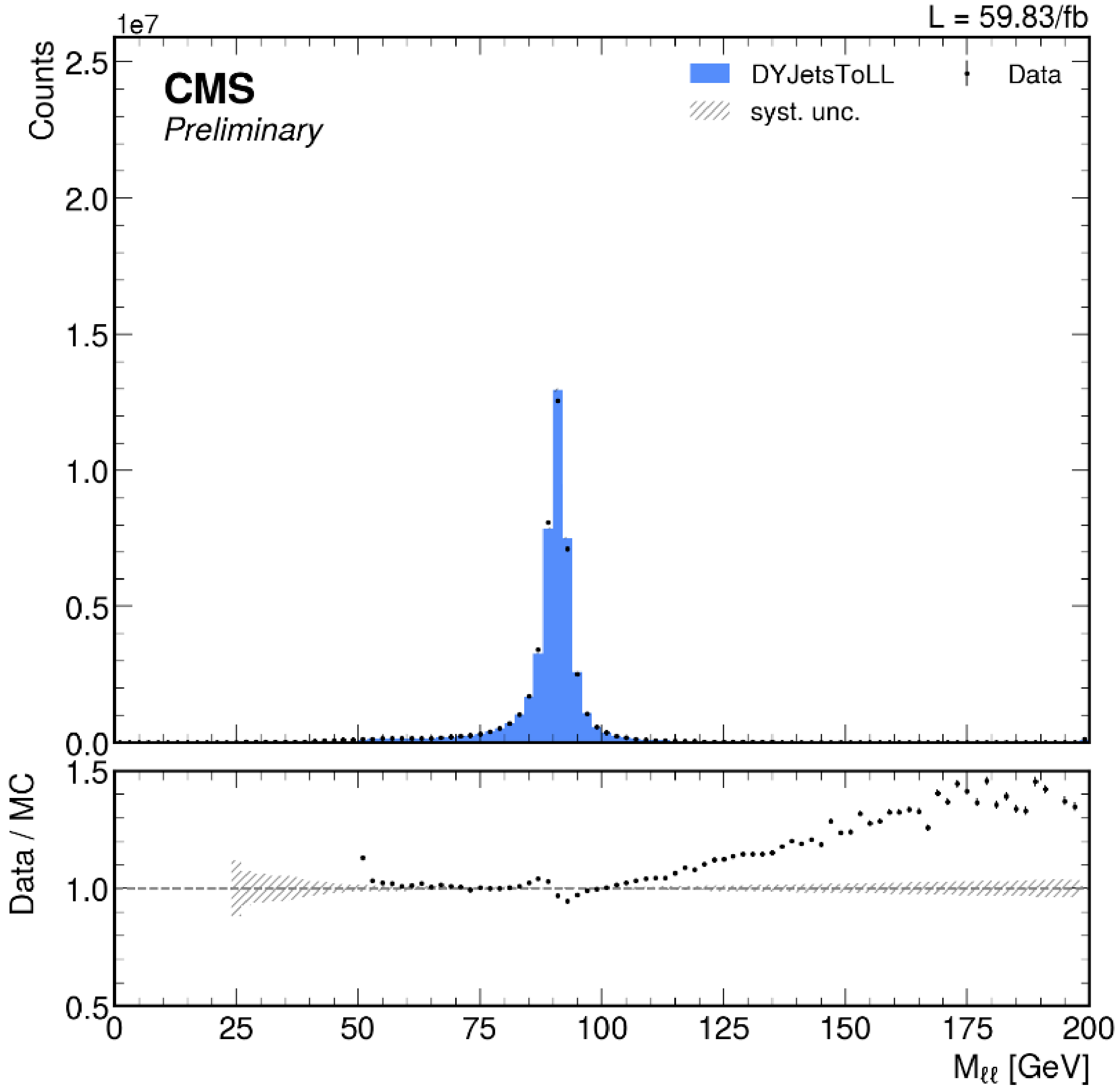
By running:

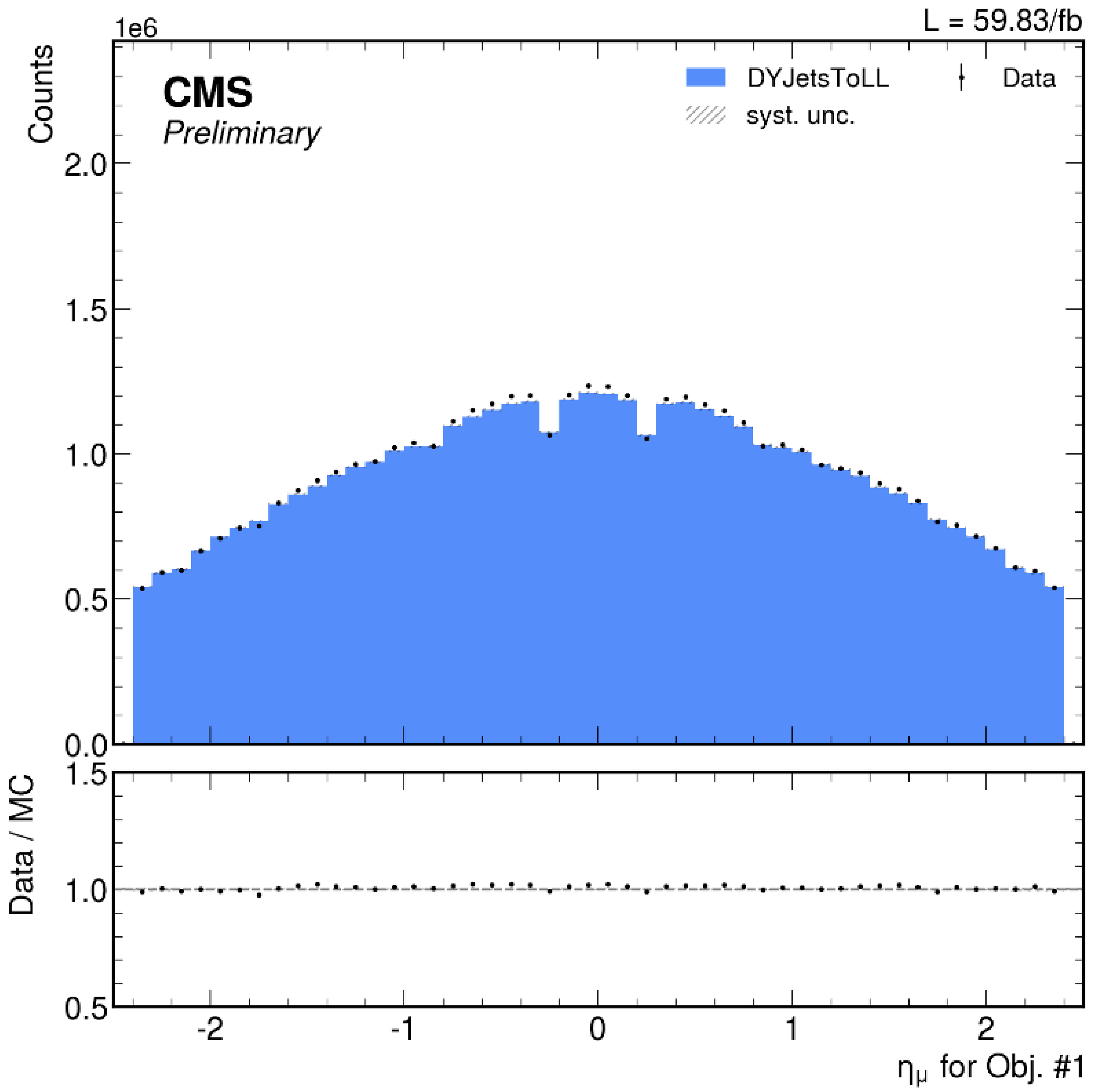
```
cd output_futures

pocket-coffea make-plots -i output_all.coffea --cfg parameters_dump.yaml -o plots
```

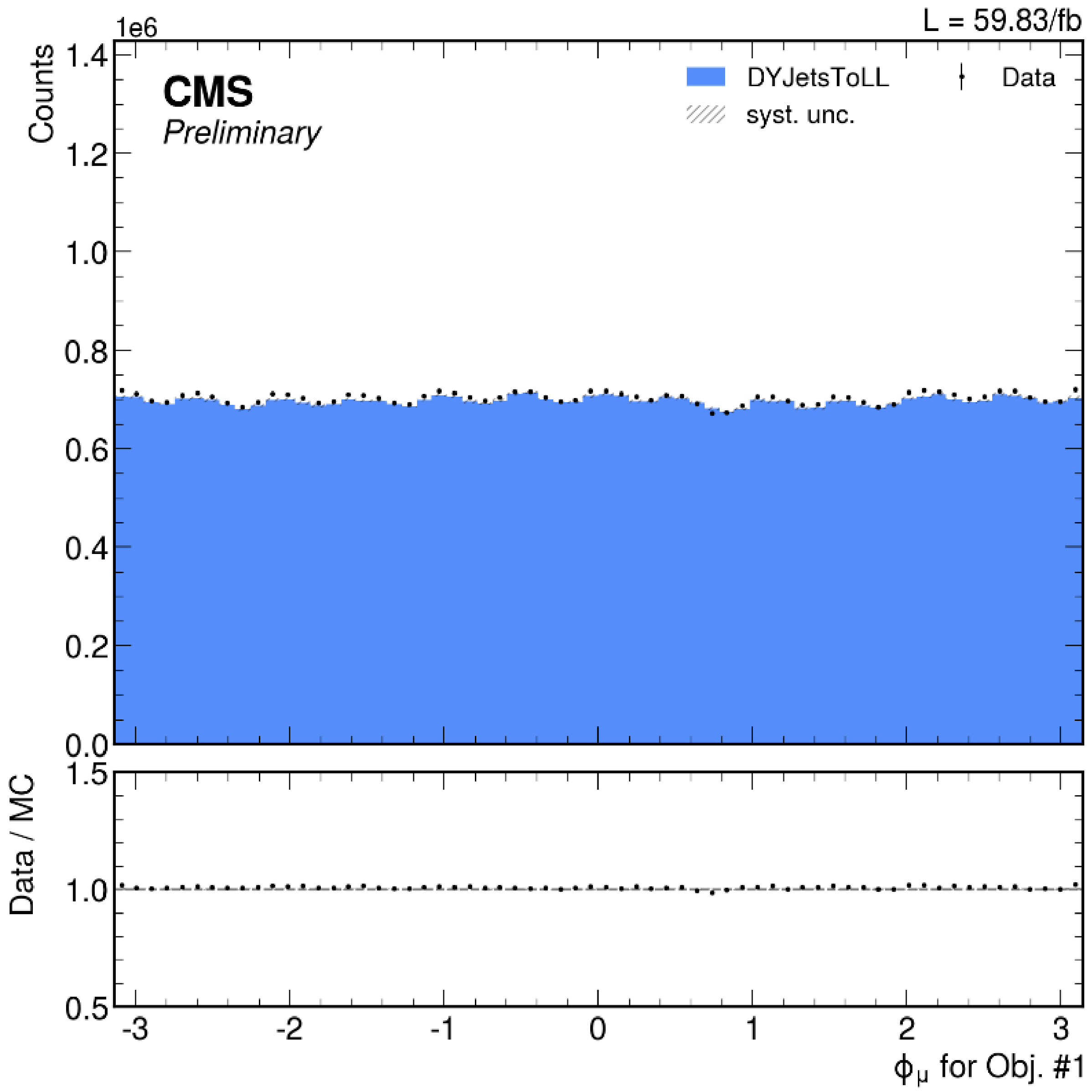
all the plots are created in the `plots` folder. The structure of the output and the dataset metadata dictionary contained in the output file are used to compose the samples list forming the stack of histograms.

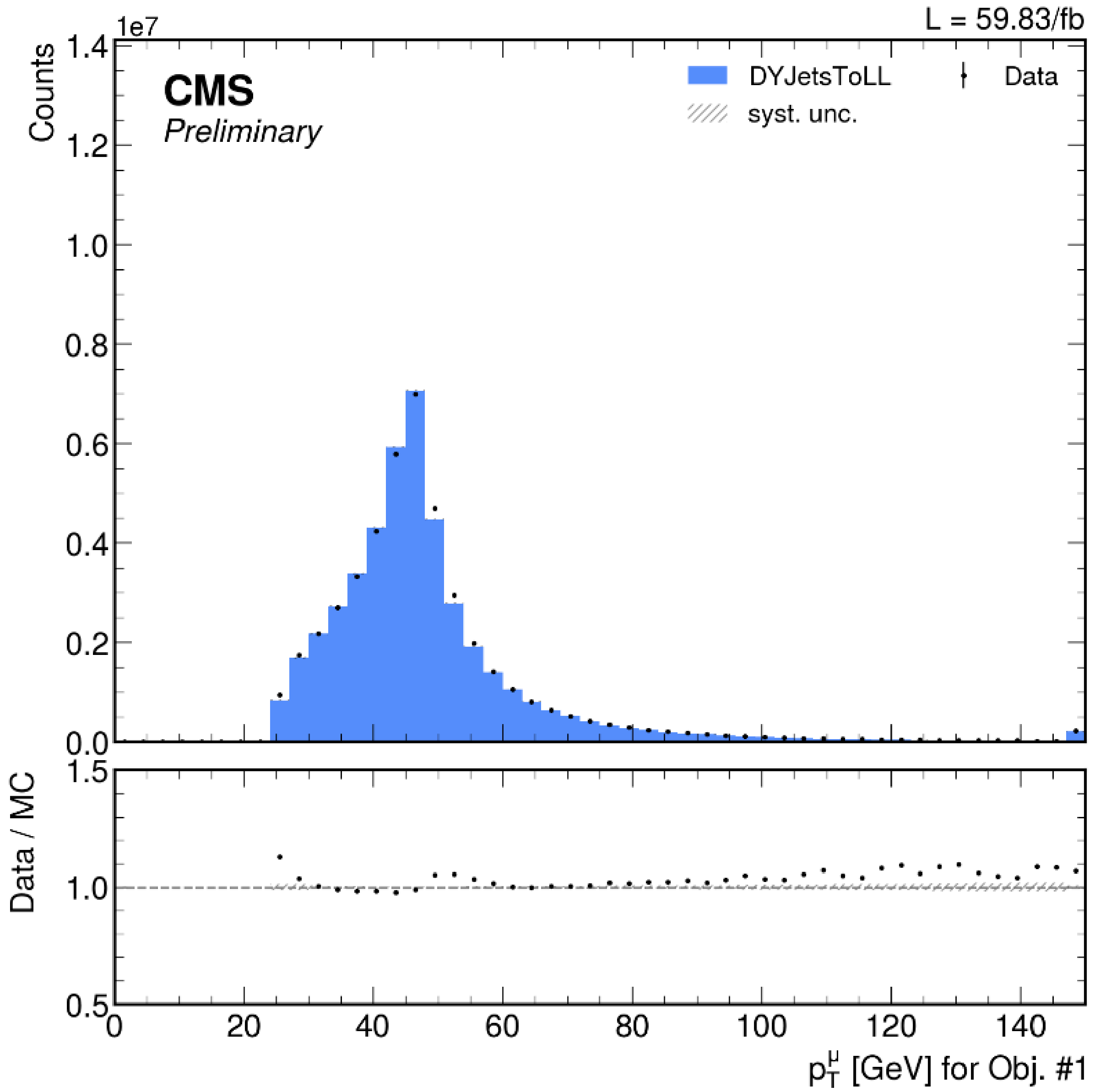
Additionally, one can customize the plotting style by passing a custom yaml file with the desired plotting options an additional argument:

```
pocket-coffea make-plots -i output_all.coffea --cfg parameters_dump.yaml -o plots -op plotting_style.yaml
```


1e7
L = 59.83/fb
Counts
CMS
Preliminary
DYJetsToLL
Data
syst. unc.
Data / MC
0.0
0.5
1.0
1.5
2.0
2.5
0.5
1.0
1.5
0
25
50
75
100
125
150
175
200
$M_{\ell\ell}$ [GeV]


L = 59.83/fb
1e6
Counts
CMS
Preliminary
DYJetsToLL
Data
syst. unc.
2.0
1.5
1.0
0.5
0.0
Data / MC
1.5
1.0
0.5
-2
-1
0
1
2
η_μ for Obj. #1


1e6
L = 59.83/fb
Counts
CMS
Preliminary
DYJetsToLL
Data
syst. unc.
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
Data / MC
1.5
1.0
0.5
-3
-2
-1
0
1
2
3
ϕ_{μ} for Obj. #1


L = 59.83/fb
1e7
Counts
CMS
Preliminary
DYJetsToLL
Data
syst. unc.
Data / MC
p_T^μ [GeV] for Obj. #1
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
1.5
1.0
0.5
0
20
40
60
80
100
120
140