Hands-on Analysis Exercise

$$H \rightarrow ZZ \rightarrow 4l$$

with

columnflow

Authors

Matteo Bonanomi, Philip Keicher

Daniel Savoiu, Ana Andrade

June 2024

This exercise was originally created for the Higgs PAG exercise at the CMS Physics Objects & Data Analysis School held in Hamburg in October 2023

Contents

List of Exercises

Chapter 1

Introduction to ColumnFlow

ColumnFlow is a back-end for analyses in order to facilitate processing large amounts of data. It is purely python-based and employs multiple packages that are common in the HEP community and well-maintained. At the time of writing these instructions, the team of developers purely consists of data analysts at the CMS experiment. Therefore, this exercise is structured accordingly. However, ColumnFlow is designed in an experiment agnostic way and it can be extended to other use cases.

Additionally, please note that this hands-on exercise is not meant to fully document all available functionalities. The purpose of this exercise is to give an overview of the most fundamental aspects and concepts that are available at the time of writing. For a more comprehensive overview, please visit the documentation [1]. In case of any questions or comments, feel free to contact the maintainers for example via the git repository [1].

1.1 General structure

The guiding principle of ColumnFlow is that all analyses share basic work packages that need to be done when processing data. Examples for such packages could be the calibration of relevant objects, applying selections to define a fiducial phase space for the analysis or the calculation of some sensitive observables, which are discussed in more detail in later chapters of this document. ColumnFlow defines the work packages as `law` tasks, which can define dependencies amongst each other and will only run necessary tasks to obtain the requested output.

Figure 1.1 depicts an overview of the available tasks and their dependencies. The highlighted regions indicate use cases that are discussed in Chapter 2. This chain of jobs starts with obtaining the list of logical file names (LFNs) that contain the events to be analysed in a flat tuple format (e.g. the nanoAOD format within CMS). The first block is dedicated to prepare these events for further analysis. Such a preparation can entail different things, such as a calibration of the relevant objects in an analysis or the application of selection criteria to define a relevant work

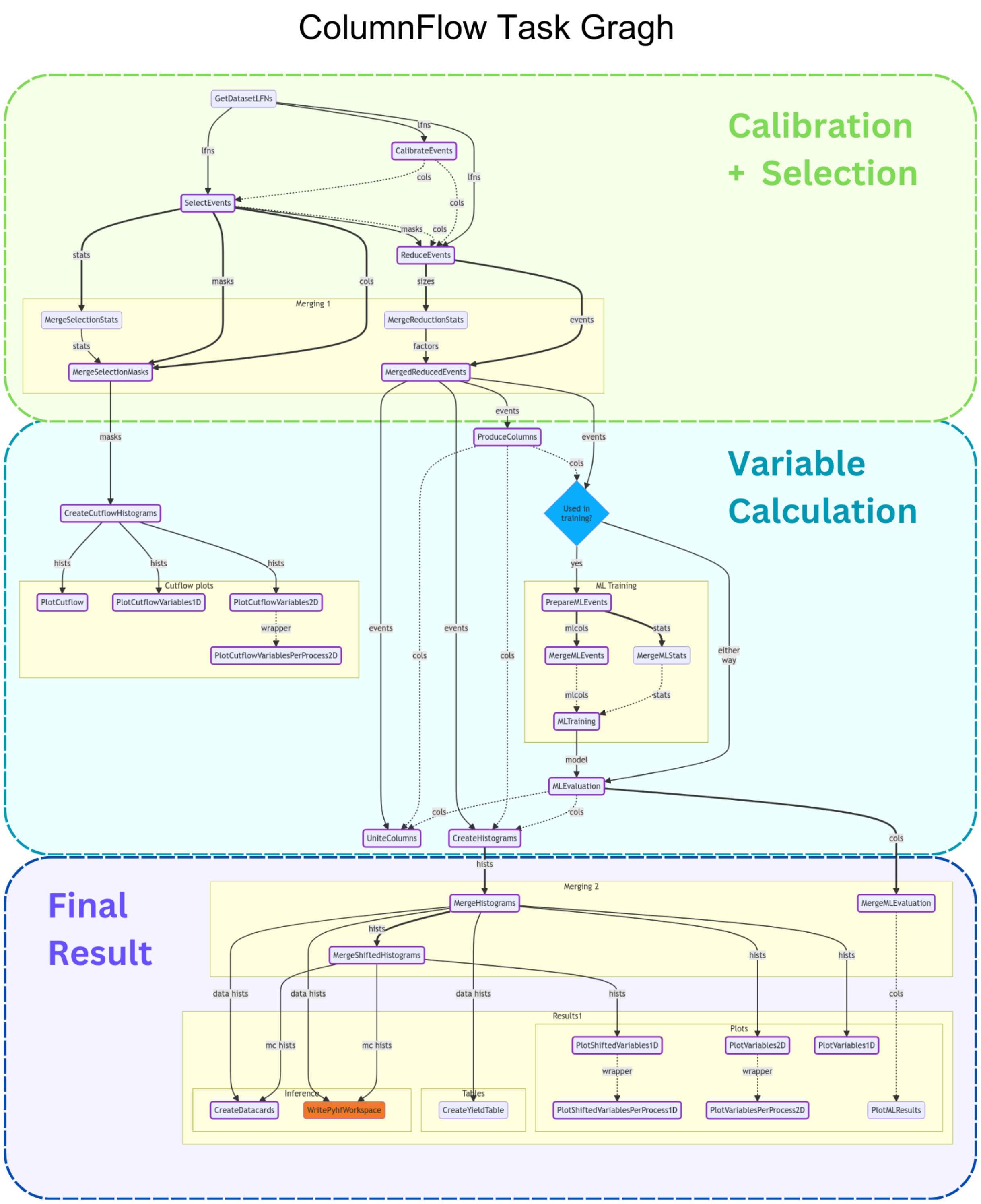


Figure 1.1: **ColumnFlow task graph hierarchy**. The tasks are arranged in three sections that correspond to general work packages in a data analysis. The line widths and styles indicate the behaviour when propagating information between tasks, as illustrated in the GitHub Wiki [1].

space. In order to facilitate a more efficient calculation in later parts of this workflow, the amount of data is reduced as a last step of the first plot.

The second block in Fig. 1.1 is dedicated to the calculation of different observables and metrics. At the time of writing these instructions, this blocks offers metrics such as a summary of efficiencies for different stages of the selections and their effect on observables, the calculation of completely new variables and also more complex calculations based on machine learning. Moreover, it offers the functionality to collect all information of the workflow and save it as a flat tuple in the e.g. ROOT or PARQUET format. The modular structure of the individual tasks allows for an easy extensions to calculate a variety of observables.

Finally, the last block is dedicated to the final observables that are needed for the analysis. Most of these endpoints of the workflow aim to facilitate a data analysis in a binned format, though this is not a hard criterion. This includes producing figures illustrating one- or two-dimensional distributions of multiple physics processes under consideration of a wide variety of systematic uncertainties, as well as the input needed for a statistical inference based on the data (e.g. datacards for the Combine tool within CMS [2]).

This structure allows a full end-to-end analysis. The explicit definition of dependencies in the code and the implicit check for existing outputs provided by `luigi` and `law` result in an automatically organised and reproducible workflow that is easily triggered with a single command. In the following, these capabilities are illustrated using an example that is based on the $H \to 4l$ analysis [3], for which we will build a selection of the aforementioned modules. Please note that this example is by no means as complex and sophisticated as the real CMS analysis, and should therefore not be expected to yield the same results.

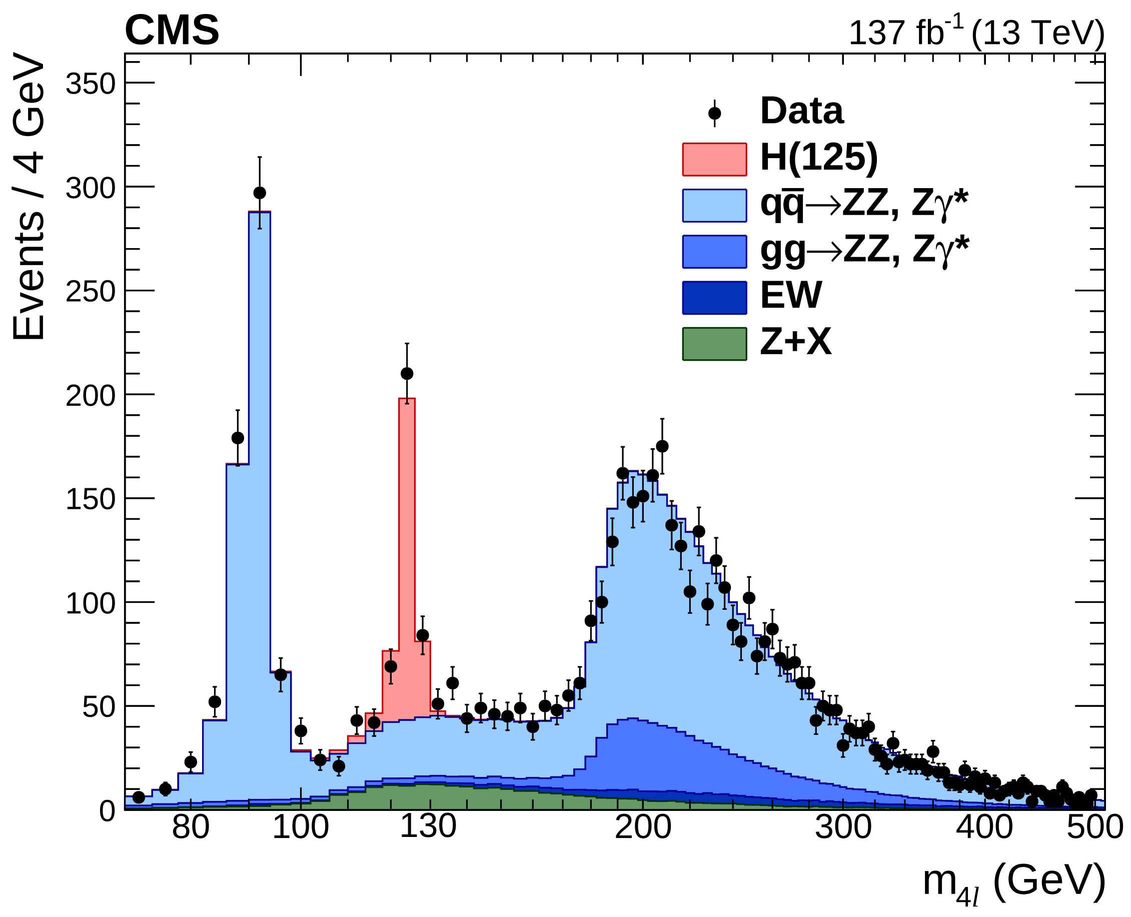


Figure 1.2: **Reconstructed four-lepton invariant mass m_{4l} with full Run 2 dataset**. The SM Higgs boson signal with $m_H = 125\,\text{GeV}$, denoted as H(125), and the ZZ backgrounds are normalized to the SM expectation. The $Z + X$ background is normalized to the estimation from data. Figure taken from Ref. [3].

1.2 Physics example: $H \to ZZ \to 4l$

The goal of this exercise is to reconstruct the standard model (SM) Higgs boson mass, using a selection targeting the four-lepton final state. This is considered a *golden* channel to measure the properties of the Higgs boson because:

- it is a **fully resolved final state** – the Higgs boson can be reconstructed from the reconstructed particles;
- we have an excellent **mass resolution** – due to the high lepton-p_T resolution, we have optimal shape reconstruction of m_{4l};
- there is a **large signal to background ratio** – it is easy to discriminate between the peak of the reconstructed four-lepton mass (m_{4l}) and the overall flat background shape.

1.3 Installation & setup

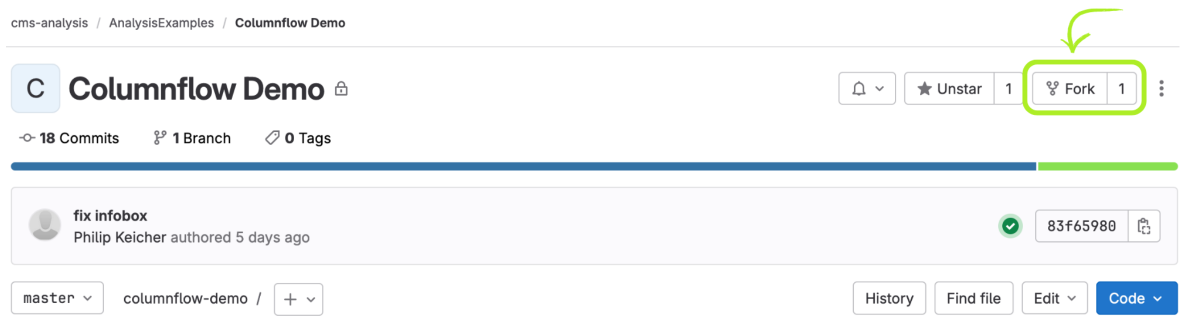
Note: ColumnFlow only runs on Linux and may require up to 4 GB of disc space.

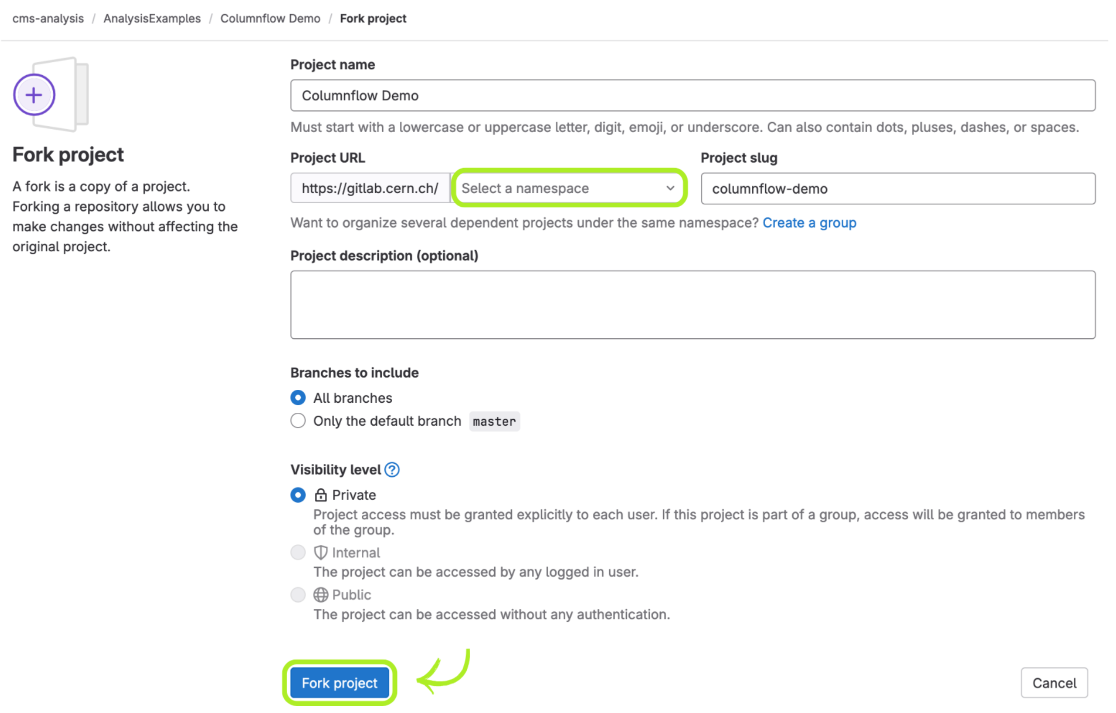
Also, the machine where you run this exercise must be mounted with CERN AFS.

Start by going to the GitLab repository of this exercise:

https://gitlab.cern.ch/cms-analysis/analysisexamples/columnflow-demo

To have your own copy of the code, fork the repository into your personal area. You can do this by clicking the `Fork` button on the upper right corner of the page. To set your Project URL please type your CERN username in the `Select a namespace` option.




cms-analysis / AnalysisExamples / Columnflow Demo / Fork project
Fork project

A fork is a copy of a project.
Forking a repository allows you to make changes without affecting the original project.

Project name: Columnflow Demo

Must start with a lowercase or uppercase letter, digit, emoji, or underscore. Can also contain dots, pluses, dashes, or spaces.

Project URL: https://gitlab.cern.ch/ Select a namespace

Project slug: columnflow-demo

Want to organize several dependent projects under the same namespace? Create a group

Project description (optional)

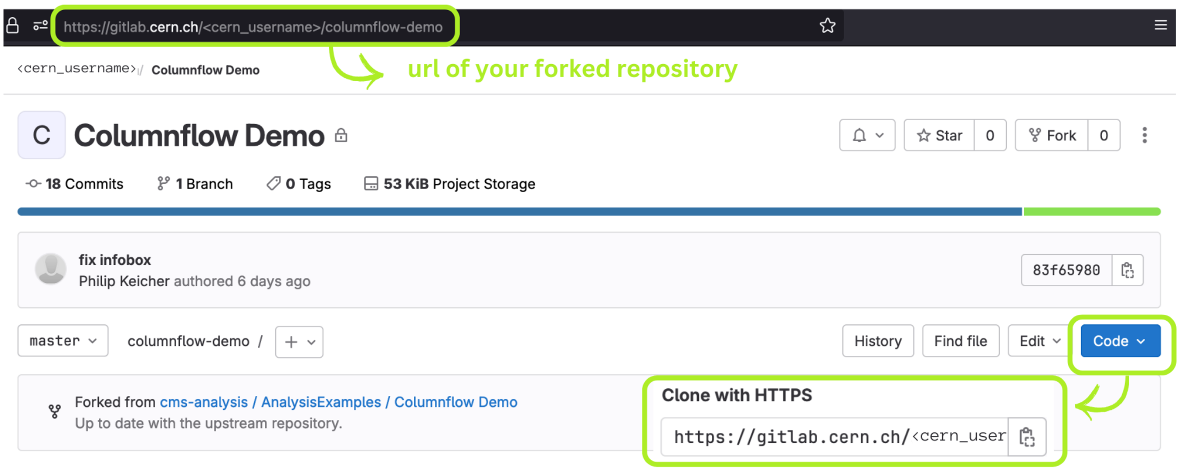
Branches to include
- [✓ All branches]
- Only the default branch `master`

Visibility level
- [✓ Private] Project access must be granted explicitly to each user. If this project is part of a group, access will be granted to members of the group.
- Internal: The project can be accessed by any logged in user.
- Public: The project can be accessed without any authentication.

After clicking the `Fork project` button, your fork url should be:

https://gitlab.cern.ch/<cern_username>/columnflow-demo

In your forked project, go to the `Code` button on the right hand side of the page and copy the address under the `Clone with HTTPS` option. If you have an SSH key registered on GitLab prior to this exercise, you can also use the `Clone with SSH` option.


https://gitlab.cern.ch/<cern_username>/columnflow-demo
url of your forked repository
<cern_username> / Columnflow Demo
Columnflow Demo
Star 0
Fork 0
18 Commits
1 Branch
0 Tags
53 KiB Project Storage
fix infobox
Philip Keicher authored 6 days ago
83f65980
master
columnflow-demo /
History
Find file
Edit
Code
Forked from cms-analysis / AnalysisExamples / Columnflow Demo
Up to date with the upstream repository.
Clone with HTTPS
https://gitlab.cern.ch/<cern_user

Next, open a new terminal window and clone your code to your machine by running one of the following commands (depending on which cloning method you chose):

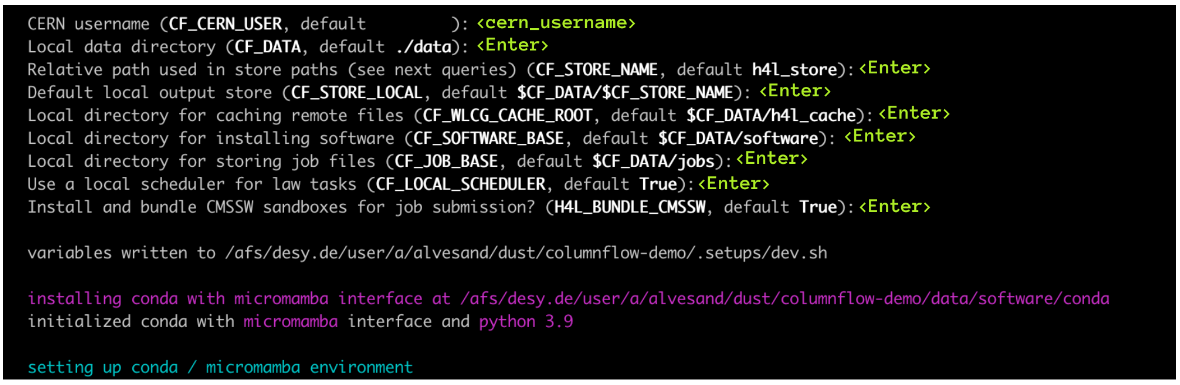
```
git clone --recursive https://gitlab.cern.ch/<cern_username>/columnflow-
    demo.git
```

```
git clone --recursive ssh://git@gitlab.cern.ch:7999/<cern_username>/
    columnflow-demo.git
```

The directory you have thus created will be referred to as `basedir`. You can now go inside your local repository and install ColumnFlow. The `setup.sh` bash script will initialize the software environment with `micromamba`. Here, we define `dev` as the setup name, but you are free to name it as you wish.

```
cd columnflow-demo
source setup.sh dev
```

You will be asked to define a series of variables, the first of which is your CERN username. For all other variables you can keep the default name by just pressing `Enter`. Variables specific to this exercise will start with `H4L_`, while ColumnFlow specific variables start with `CF_`. You can find all variables in the `.setups/dev.sh` bash file. We invite you to check out this file and familiarize yourself with these variables.

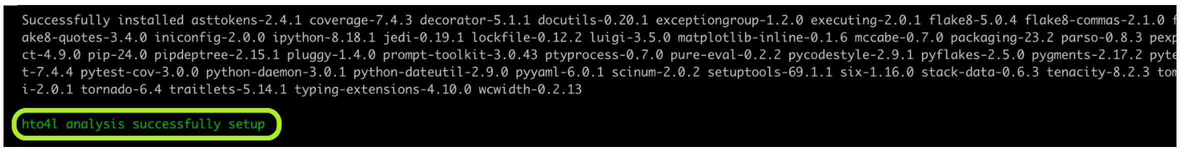


```
CERN username (CF_CERN_USER, default         ): <cern_username>
Local data directory (CF_DATA, default ./data): <Enter>
Relative path used in store paths (see next queries) (CF_STORE_NAME, default h4l_store): <Enter>
Default local output store (CF_STORE_LOCAL, default $CF_DATA/$CF_STORE_NAME): <Enter>
Local directory for caching remote files (CF_WLCG_CACHE_ROOT, default $CF_DATA/h4l_cache): <Enter>
Local directory for installing software (CF_SOFTWARE_BASE, default $CF_DATA/software): <Enter>
Local directory for storing job files (CF_JOB_BASE, default $CF_DATA/jobs): <Enter>
Use a local scheduler for law tasks (CF_LOCAL_SCHEDULER, default True): <Enter>
Install and bundle CMSSW sandboxes for job submission? (H4L_BUNDLE_CMSSW, default True): <Enter>

variables written to /afs/desy.de/user/a/alvesand/dust/columnflow-demo/.setups/dev.sh

installing conda with micromamba interface at /afs/desy.de/user/a/alvesand/dust/columnflow-demo/data/software/conda
initialized conda with micromamba interface and python 3.9

setting up conda / micromamba environment
```

Note that the first installation of the software can take up to several minutes.

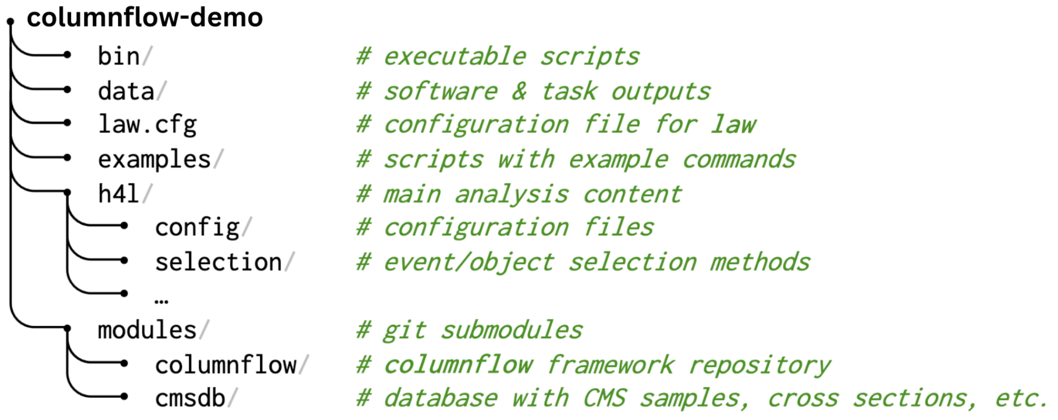
Every time you want to work with ColumnFlow (e.g. if you open a new terminal window), you will need to source the `setup.sh` script again.

Once the installation is complete you should see a line of green text stating that the analysis has been successfully set up. You are now ready to start working with ColumnFlow!



```
Successfully installed asttokens-2.4.1 coverage-7.4.3 decorator-5.1.1 docutils-0.20.1 exceptiongroup-1.2.0 executing-2.0.1 flake8-5.0.4 flake8-commas-2.1.0 f
ake8-quotes-3.4.0 iniconfig-2.0.0 ipython-8.18.1 jedi-0.19.1 lockfile-0.12.2 luigi-3.5.0 matplotlib-inline-0.1.6 mccabe-0.7.0 packaging-23.2 parso-0.8.3 pexp
ct-4.9.0 pip-24.0 pipdeptree-2.15.1 pluggy-1.4.0 prompt-toolkit-3.0.43 ptyprocess-0.7.0 pure-eval-0.2.2 pycodestyle-2.9.1 pyflakes-2.5.0 pygments-2.17.2 pyte
t-7.4.4 pytest-cov-3.0.0 python-daemon-3.0.1 python-dateutil-2.9.0 pyyaml-6.0.1 scinum-2.0.2 setuptools-69.1.1 six-1.16.0 stack-data-0.6.3 tenacity-8.2.3 tom
i-2.0.1 tornado-6.4 traitlets-5.14.1 typing-extensions-4.10.0 wcwidth-0.2.13

hto4l analysis successfully setup
```

Inside of your newly created `columnflow-demo` directory, you will find the following project structure:



```
columnflow-demo
├── bin/              # executable scripts
├── data/             # software & task outputs
├── law.cfg           # configuration file for law
├── examples/         # scripts with example commands
├── h4l/              # main analysis content
│   ├── config/       # configuration files
│   ├── selection/    # event/object selection methods
│   └── …
└── modules/          # git submodules
    ├── columnflow/   # columnflow framework repository
    └── cmsdb/        # database with CMS samples, cross sections, etc.
```

This exercise is organized in the form of `law` tasks, where different tasks create some form of output. You can view the available tasks by running:

```
law index --verbose
```

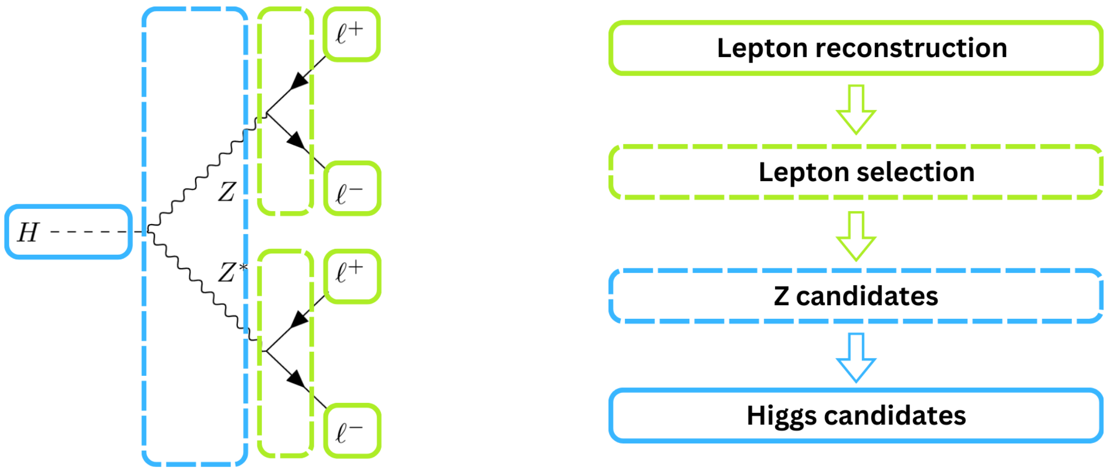
This exercise will focus on the following tasks:

- `cf.CalibrateEvents` / `cf.SelectEvents`
- `cf.ProduceColumns`
- `cf.PlotCutflow`
- `cf.PlotVariables1D` / `cf.PlotVariables2D`
- `cf.CreateDatacards`

By default, these tasks will save their output on a remote file system (e.g. `WLGC`), for which you will require a `voms-proxy`. If you would like to save certain/all outputs locally, we recommend to create a directory on a system with a larger amount of disk space (e.g. `EOS`). For such cases, you will need to update the `law.cfg` file accordingly.

1.4 Analysis strategy

In order to find Higgs boson candidates, we need to reconstruct the four leptons in the final state. To select the four lepton candidates in the first place, we will need to write a `Selector` (Section 2.4).



Chapter 2

Basic Functionalities

This chapter illustrates how to employ the most basic features of ColumnFlow. By the end of it, you should be able to perform a calibration, apply a selection, calculate an observable and finally also produce the corresponding distribution for multiple processes. Please note that this chapter is merely meant to summarize the most important aspects of these features. For a more in-depth discussion and presentation, please consider Ref. [1].

2.1 Configuring the workflow

Concepts to (briefly) introduce here

- Order objects: Analysis, configs
- law config for module resolution
- brief walk through through demo config?

Might want to move this to Chapter 1.

2.2 The mother of all: TaskArrayFunctions

Before getting started with the details of the implementation, we will cover the basic structure of the most relevant building blocks of ColumnFlow. These objects are all derived from the so-called `TaskArrayFunction`, which defines hooks and interfaces to propagate information from your objects to the ColumnFlow tasks.

This class of objects can for example explicitly define some runtime dependencies with the following member variables:

uses is a set of column names that are to be retrieved from disk. You can also provide other `TaskArrayFunction`s here, in which case their `uses` set is appended to this one.

produces is a set of columns that are to be written to disk. You can also provide other `TaskArrayFunction`s here, in which case their `produces` set is appended to this one.

sandbox is a hook that is propagated to the actual Task instance that is run and calls your module. This defines the software environment in which your module needs to run, which allows for a granular definition of the required software and can minimize the overhead of your software packages.

For convenience, all `TaskArrayFunction`s provide decorators to easily define new modules:

```python
# assuming you want to define the TaskArrayFunction example
@example(
  # for example, request the Jet pt, all Electron information and everything another
  uses = {
    "Jet.pt",  # request transverse momentum for all jets
    "Electron.*",  # request all information for electrons
    some_other_TaskArrayFunction  # propagate everything some_other_TaskArrayFunction needs to this example TaskArrayFunction
    },
    # define which outputs are to be written to disk
    produces={
      "some_fancy_output"
    },
    # define in what kind of software environment this module should be run
    sandbox="some_cool_sandbox"
)
def your_new_example_module(events: ak.Array, **kwargs):
    # this is the main body of your module, do something here...
```

Additionally, `TaskArrayFunction`s provide a set of hooks, three of which are of special importance and are briefly introduced in the following:

init defines instructions that are to be done when this object is first initialised.

requires adds object-specific requirements on top of the pre-existing Task-level requirements. This allows to explicitly define dependencies and can for example ensure that the output of another module is calculated before starting with the current task.

setup is run before actually entering the main body of your module that performs e.g. calculations. This is for example useful to parse the output of the aforementioned requirements such that your object can also use it.

These hooks can be added to an existing `TaskArrayFunction` instance with dedicated decorators like so:

```python
# assuming you have defined your_new_example_module from the example
 above

# define your init function
@your_new_example_module.init
def some_init_func_name(self):
    # do something when your_new_example_module is first initialized

# define some special requirements for your module
@your_new_example_module.requires
def some_func_name_for_requires(self):
    # add some requirements

# do something before entering the main body of your module
@your_new_example_module.setup
def some_setup_func_name(self, **kwargs):
    # prepare your module so it runs smoothly
```

In the following, these concepts will be shown with concrete details.

2.3 Writing a Calibrator

`Calibrator`s are dedicated `TaskArrayFunctions` that perform a calibration of objects, such as jets, leptons or missing transverse energy. Since this calibration modifies the four-momenta in the events, they can influence the selection of a given analysis. Therefore, calibrations should generally be performed before applying analysis selections. The associated task within the workflow is `cf.CalibrateEvents`, which is executed before the selection modules within the task graph.

ColumnFlow provides generally-used calibrations for different objects which follow the common (CMS) guidelines. For more information about which calibrations are implemented and how to use them, we recommend to consult the current status of the documentation [1].

The `H4L` analysis includes an exemplary `Calibrator` in `h4l/calibration/jets.py`. In this module, we perform a calibration of jets in our events, which is based on the implementation that comes with ColumnFlow itself.

First the relevant modules are imported. Note that `awkward` is loaded with the `maybe_import` mechanism. This is necessary due to the encapsulated structure of the underlying software stack. In the scope of this exercise, we don't want to consider all the different sources of uncertainties that are associated with jet calibration yet. Therefore, we use the `derive` mechanism of `TaskArrayFunctions` to define a new class called `jec_nominal`, which inherits from the original `jec` `Calibrator` but overwrites the corresponding class member variable.

Next, we define our new `Calibrator` class `jet_energy` as shown before. Since we want to call the `jec_nominal` class within this `Calibrator`, we need to add it to the `uses` set. This will load all columns that `jec_nominal` needs, and will additionally make `jec_nominal` accessible within the main body of our new `Calibrator` as shown below. We also want to save all columns that `jec_nominal` produces to disk for later use, which is why we need to add it to the `produces` set as well.

All `TaskArrayFunctions` have access to information of the current point within the task graph, such as the `config` object mentioned in Sec. 2.1 and the current dataset that is processed. Their behavior can depend on this information, which is shown for the jet energy resolution (JER) calibration of our new `jet_energy` module. JER needs to be applied to simulated samples only, which is realized in the code correspondingly. The set of columns to be loaded from disk is also dynamically configured in the `init` function of the `jet_energy` `Calibrator` such that columns corresponding to JER are only added to the `uses` and `produces` sets if necessary.

Exercise 2.1: Writing a Calibrator

Familiarize yourself with how the `jet_energy` `Calibrator` works.

Table 1: **Selection criteria for leptons**. Shown are the selection criteria for electrons (muons) at the 'loose' and 'tight'

2.4 Writing a Selector

The `Selector` class should be used to implement analysis selections. This is a crucial step in the workflow since the decision to keep or reject objects or even whole events is performed here. Since the selection usually depends on for example four momenta of the objects within the events, it is executed after the calibration. The corresponding task is called `cf.SelectEvents`. For more information, please consider Ref. [1].

In this part of the tutorial, we will write selections for electrons and muons. **Loose Electrons**

-

2.5 Writing a Producer

Chapter 3

Advanced Topics

Chapter 4

Advanced Topics

4.1 Defining categories

4.2 Defining Systematic Uncertainties

4.3 Define Sets of Weights to use for Templates

4.4 Writing datacards

Bibliography

[1] The ColumnFlow Team. *The ColumnFlow project*. Version 3b8e0f3. Documentation available at `https://columnflow.readthedocs.io/en/latest/`. 2024. URL: `https://github.com/columnflow/columnflow`.

[2] Aram Hayrapetyan et al. "The CMS Statistical Analysis and Combination Tool: COMBINE". In: (Apr. 2024). arXiv: `2404.06614 [physics.data-an]`.

[3] The CMS Collaboration. *Measurements of production cross sections of the Higgs boson in the four-lepton final state in proton–proton collisions at $\sqrt{s} = 13\,TeV$*. June 2021. DOI: `10.1140/epjc/s10052-021-09200-x`. URL: `http://dx.doi.org/10.1140/epjc/s10052-021-09200-x`.