Building and steering template fits with cabinetry

Kyle Cranmer[1], **Alexander Held[1]**

[1] New York University

25th International Conference on Computing in High Energy & Nuclear Physics
https://indico.cern.ch/event/948465/
May 19, 2021

This work was supported by the U.S. National Science Foundation (NSF) Cooperative Agreement OAC-1836650 (IRIS-HEP).

Introduction

- **Binned template fits** are widely used for **statistical inference** at the LHC and beyond

- **`HistFactory`** is a statistical model for **binned template fits**
  - prescription for constructing probability density functions (pdfs) from small set of building blocks
  - models can be serialized to *workspaces*
  - covers wide range of use cases, extensively used in ATLAS

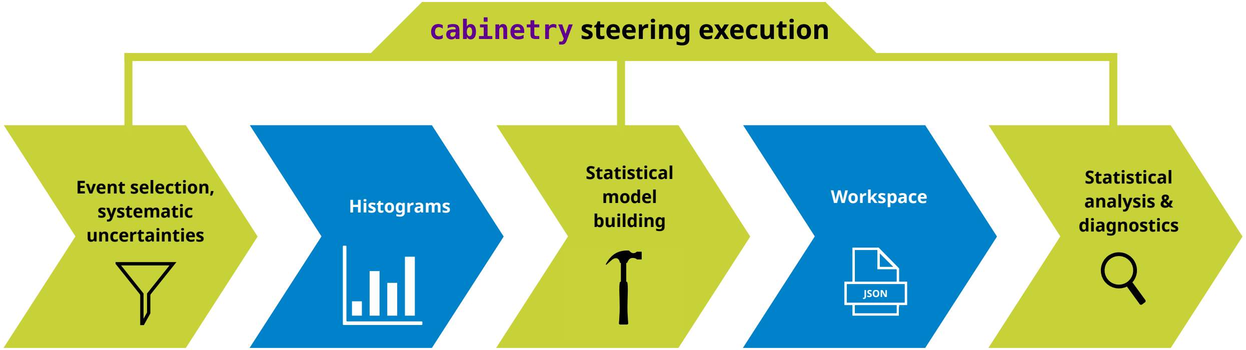
the **`HistFactory`** pdf (pyhf docs)

$$f(n, a \mid \eta, \chi) = \underbrace{\prod_{c \in \text{channels}} \prod_{b \in \text{bins}_c} \text{Pois}\left(n_{cb} \mid \nu_{cb}(\eta, \chi)\right)}_{\text{Simultaneous measurement of multiple channels}} \underbrace{\prod_{\chi \in \boldsymbol{\chi}} c_\chi(a_\chi \mid \chi)}_{\text{constraint terms for “auxiliary measurements”}}$$

- **`cabinetry`** is a **Python library** for constructing and operating **`HistFactory`** models
  - ❯ `pip install cabinetry`
  - uses **`pyhf`** (**`HistFactory`** model in Python)
  - integrates seamlessly with the flourishing Python HEP ecosystem
  - modular design: drop in and out of **`cabinetry`** whenever needed

Working with cabinetry

- **cabinetry** is used to
  - **design** and **construct statistical models** (workspaces) from instructions in **declarative configuration**
    - analyzers specify selections for signal/control regions, (Monte Carlo) samples, systematic uncertainties
    - **cabinetry** steers creation of template histograms (region ⊗ sample ⊗ systematic)
    - **cabinetry** produces **HistFactory** workspaces (serialized fit model)
  - perform **statistical inference**
    - including diagnostics and visualization tools to study and disseminate results



Designing a statistical model

- **declarative configuration** (JSON/YAML/dictionary) specifies everything needed to build a workspace
  - can concisely capture complex region ⊗ sample ⊗ systematic structure

general settings

list of phase space regions (channels)

list of samples (MC/data)

list of systematic uncertainties

list of normalization factors

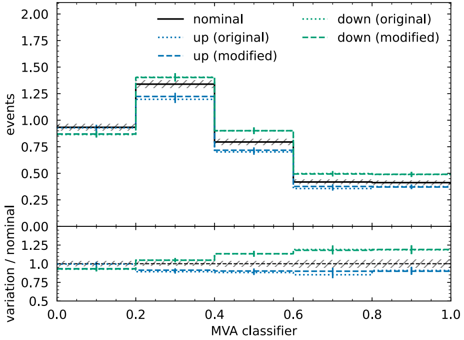
```yaml
General:
  Measurement: "Example"
  InputPath: "input/{SamplePaths}"
  HistogramFolder: "histograms/"
  POI: "Signal_norm"

Regions:
  - Name: "Signal_region"
    Filter: "nJets >= 8"
    Variable: "jet_pt"
    Binning: [200, 300, 400, 500]

Samples:
  - Name: "Data"
    SamplePaths: "data.root"
    Tree: "events"
    Data: True

  - Name: "Signal"
    SamplePaths: "signal.root"
    Tree: "events"
    Weight: "weight_nominal"

  - Name: "Background"
    SamplePaths: "background.root"
    Tree: "events"
    Weight: "weight_nominal"

Systematics:
  - Name: "Luminosity"
    Up:
      Normalization: 0.05
    Down:
      Normalization: -0.05
    Samples: ["Signal", "Background"]
    Type: "Normalization"

  - Name: "ModelingVariation"
    Up:
      Tree: "events_up"
      Weight: "weight_modeling"
    Down:
      Tree: "events_down"
      Weight: "weight_modeling"
    Smoothing:
      Algorithm: "353QH, twice"
    Samples: "Background"
    Type: "NormPlusShape"

NormFactors:
  - Name: "Signal_norm"
    Samples: "Signal"
    Nominal: 1
    Bounds: [0, 10]
```

Template histograms and workspace building

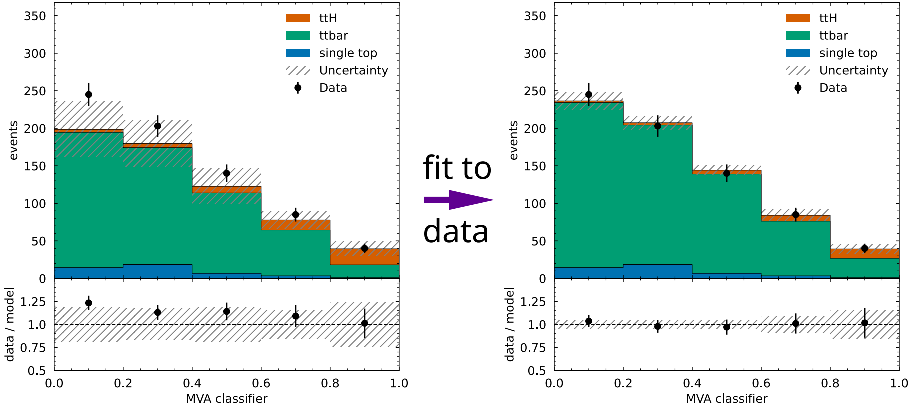
- **workspaces construction** happens in three steps:
  1) create template histograms from columnar data following config instructions
     - backends execute instructions (default: **`uproot`**, experimental: **coffea**)
  2) optional: apply post-processing to templates (e.g. smoothing)
  3) assemble temples into workspace (JSON file)

- utilities provided to **visualize and debug** fit model
- possible to provide **custom code** for template creation

visualization of individual template histograms



fit model visualization



event yield table

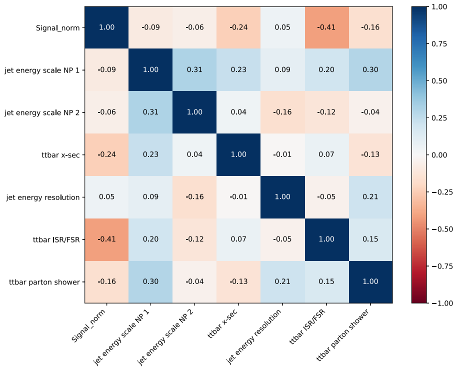
sample	Control region	Signal region
single top	44.74	0.35
ttbar	635.98	13.28
z_ttH	30.90	1.80
total	711.61 ± 28.28	15.43 ± 2.69
data	713.00	14.00

Statistical inference

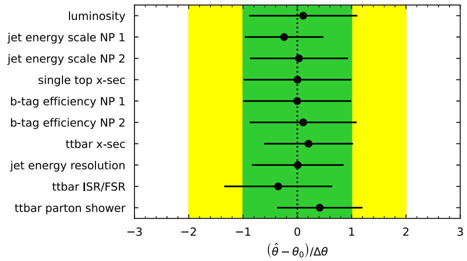
example: to produce both plots below, use 3 lines of Python to call the **cabinetry** API
or single CLI instruction: `$ cabinetry fit --pulls --corrmat ws.json`

- implementations for all **common inference tasks** exist
  - includes associated visualizations
  - results validated against **ROOT**-based implementation

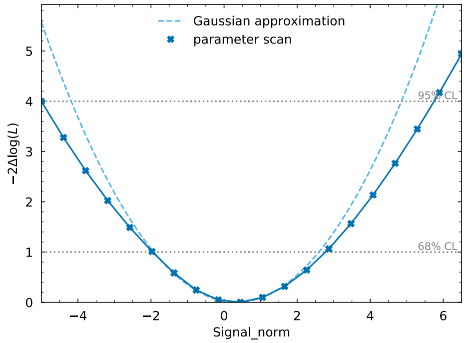
parameter correlations



nuisance parameter pulls



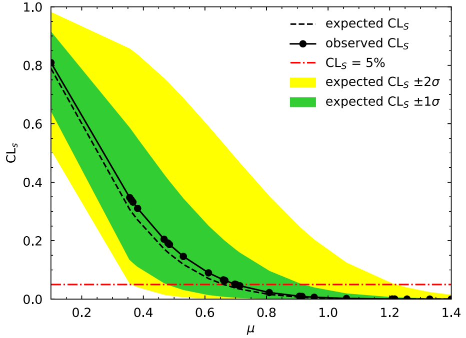
likelihood scans



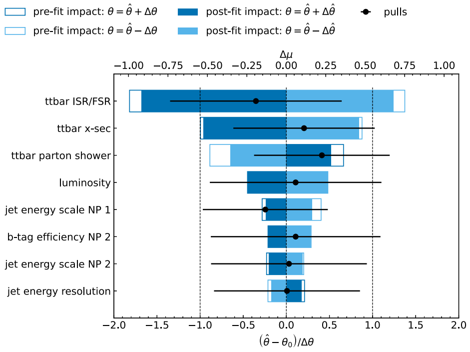
discovery significance

```
$ cabinetry significance workspaces/example_workspace.json
INFO - cabinetry.fit - calculating discovery significance
INFO - cabinetry.fit - observed p-value: 1.13053295%
INFO - cabinetry.fit - observed significance: 2.280
INFO - cabinetry.fit - expected p-value: 0.42110716%
INFO - cabinetry.fit - expected significance: 2.635
```

upper parameter limits



nuisance parameter impacts



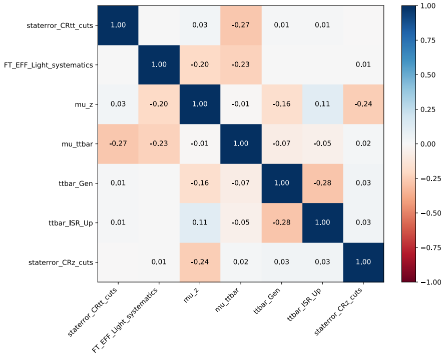
Working with an unknown workspace

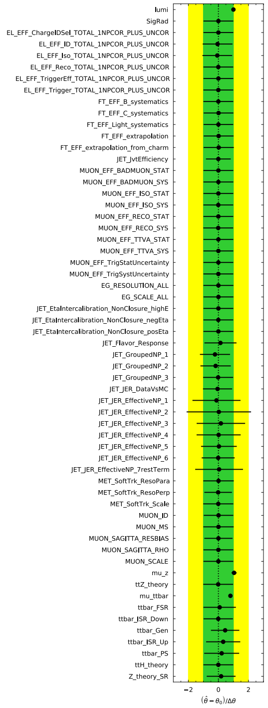
Search for bottom-squark pair production with the ATLAS detector in final states containing Higgs bosons, *b*-jets and missing transverse momentum

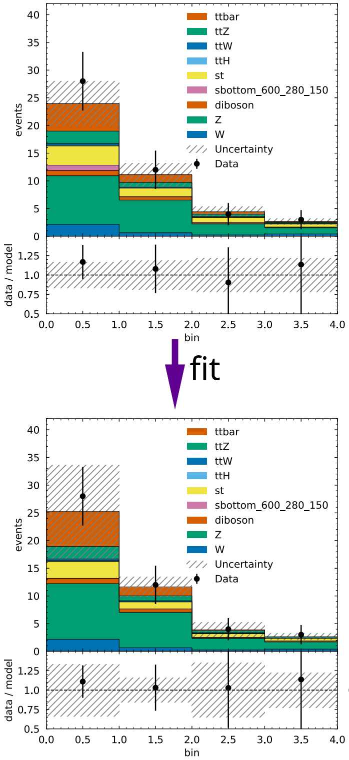
- pick a **workspace** from **HEPData**: 10.17182/hepdata.89408.v3 (analysis: JHEP 12 (2019) 060)
  - download with **pyhf**, start performing inference and studying fit model with **cabinetry** in seconds
- try it out: run on Binder!

```python
import cabinetry

ws = cabinetry.workspace.load("ws.json")
model, data = cabinetry.model_utils.model_and_data(ws)
cabinetry.visualize.data_MC(model, data)  # visualize pre-fit

fit_results = cabinetry.fit.fit(model, data)  # fit to data
cabinetry.visualize.data_MC(model, data, fit_results=fit_results)  # visualize post-fit
cabinetry.visualize.pulls(fit_results, exclude="mu_SIG")
cabinetry.visualize.correlation_matrix(fit_results, pruning_threshold=0.2)
```

sample	CRtt_cuts	CRz_cuts	SR_metsigST
W	16.66	0.00	3.48
Z	0.35	55.35	19.82
diboson	2.72	5.71	1.92
sbottom_600_280_150	0.00	0.00	0.00
st	24.56	0.00	5.45
ttH	2.82	0.10	0.27
ttW	3.48	0.00	0.63
ttZ	7.52	12.64	3.72
ttbar	86.48	1.30	8.11
total	144.59 ± 119.52	75.09 ± 8.67	43.40 ± 10.07
data	143.00	73.00	47.00



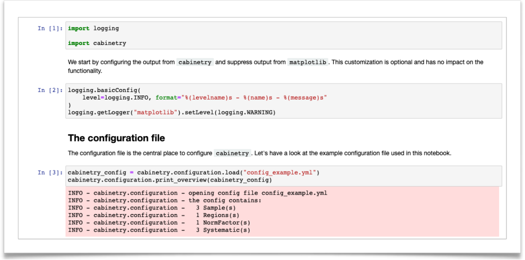




(workspace contains additional channels not shown here)

Future directions

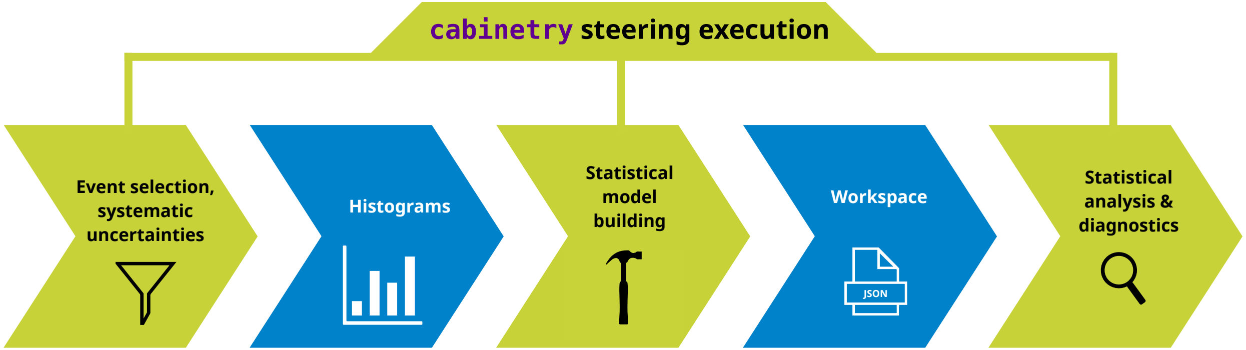
tutorial repository



- **`cabinetry`** is being **actively developed**
  - everything shown in these slides (and more) is available: try it out on Binder!
- **next steps and goals:**
  - short term: improved visualization API for simplified figure handling and customization
  - more generic handling of templates related to interpolation
    - include infrastructure to support generic systematics beyond **`HistFactory`** "up/down" types
  - longer term: support end-to-end automatic differentiation
    - optimize analysis selection and design via gradient descent, see **`neos`** for an example
  - *your ideas?*
    - your contributions and thoughts are welcome!

Summary

- **cabinetry** is
  - a modular, Python-based library to create and operate statistical models for inference with template fits
  - leveraging the power of many libraries in a growing Python HEP ecosystem
  - openly developed on GitHub
  - available to try it out yourself on Binder!

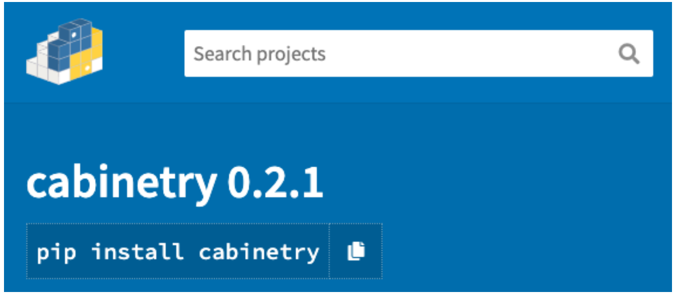


Backup

Links to cabinetry

- **cabinetry:**
  - can be installed via `$ pip install cabinetry`
    - `cabinetry[contrib]` for extra features
  - is open source and publicly developed
    - developed on GitHub
    - published on PyPI
    - documented on Read the Docs
    - part of IRIS-HEP

cabinetry on PyPI



cabinetry on GitHub

documentation on Read the Docs

Statistical analysis: the HistFactory model

- **HistFactory** is the standard model used in ATLAS for **binned statistical analysis**
  - **pyhf** is a python implementation of this model
  - the **HistFactory** model specifies how to construct the likelihood function
  - **cabinetry** turns a declarative specification about cuts, systematics etc. into a statistical model
  - **pyhf** turns that model into a likelihood function

prediction (sum of samples)

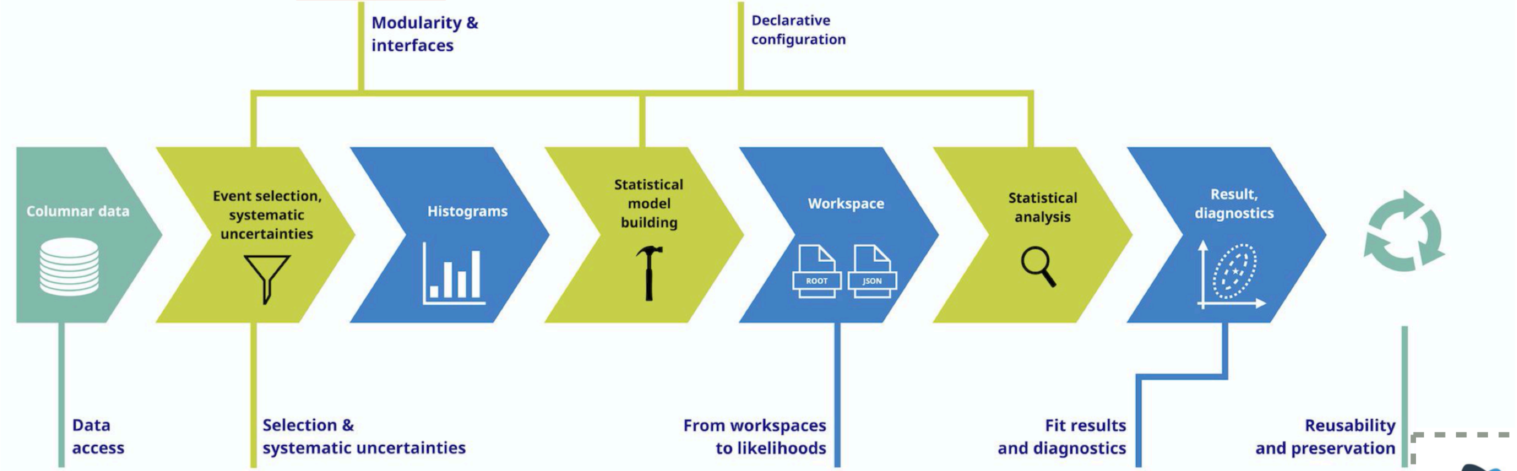
constrained parameters
controlling systematic variations

$$f(n, a \,|\, \eta, \chi) = \underbrace{\prod_{c \in \text{channels}} \prod_{b \in \text{bins}_c} \text{Pois}\left(n_{cb} \,|\, \nu_{cb}(\eta, \chi)\right)}_{\text{Simultaneous measurement of multiple channels}} \underbrace{\prod_{\chi \in \boldsymbol{\chi}} c_\chi(a_\chi \,|\, \chi)}_{\text{constraint terms for “auxiliary measurements”}}$$

cabinetry within the broader ecosystem

possibilities for specifying cuts / translating between languages

func_adl
formulate



ServiceX
uproot
Awkward Array
hep_tables
coffea

Boost histogram

pyhf
differentiable likelihoods

interface generally well defined

yadage
recast

not explored yet

synergies with hist

fitting-as-a-service
to be explored

Why `cabinetry`?

- **why `cabinetry`?**
  - pure Python and no ROOT dependency, fills gap in Python ecosystem
    - made possible by **`uproot`**, **`awkward-array`**, **`boost-histogram`** , **`pyhf`**
  - modular approach: avoid lock-in
    - benefit from growing columnar analysis ecosystem (**`coffea`** etc.)
  - openly developed, fully available to broader community beyond a specific experiment
  - follow good practices with extensive automated testing (see coverage)
  - chance to take different design decisions informed by years of experience with existing tools
    - in particular: declarative approach, but allow custom code injection at core steps in the workflow

- **why the name?**
  - a workspace is like a cabinet — it organizes data into many bins (like drawers in a cabinet)
  - the building of these “workspace cabinets” is **`cabinetry`**

Design considerations

Design considerations (1)

- **`modularity`**
  - functionality is factorized wherever possible
    - use the pieces you want / need, without needing to commit to *only* using **`cabinetry`**
  - no interplay between workspace building and fitting
    - use fitting / debugging utilities for any **`HistFactory`** workspace, including **`ROOT`** version (convert with **`pyhf`**)
  - modularity makes it easier to interface new technologies / libraries that may appear in the future

- **usability as a library**
  - **`cabinetry`** can be used as a framework (e.g. via the CLI), but more control is possible by using it as a library
    - control program flow by selecting what function to call when, and modify parameters as needed

Design considerations (2)

- **workspace building**
  - core library implements logic to generate instructions for histogram building
  - histogram building is factorized (see `cabinetry.contrib`)
  - can use custom code called by **`cabinetry`** for histogram building
    - avoids re-implementation of logic for histogram building and workspace creation

- **fitting**
  - lightweight containers storing results (example: `FitResults`)
  - easy to convert to other formats as needed, or serialize to file as JSON/YAML/ROOT…

example:
results from a MLE fit

```
{
    "bestfit": array([ 0.99973053,  0.99999815,  0.8519698 , -0.0154429 ,  0.15844803]),
    "uncertainty": array([0.02871064, 0.02867999, 0.41313841, 0.98982413, 0.16965913]),
    "labels": ["staterror_SR[bin_0]", "staterror_SR[bin_1]", "Signal_norm", "Luminosity", "Modeling"],
    "corr_mat": [[ 1.00000000e+00  8.22161732e-05 -9.09372332e-02 -5.05369226e-03 3.24459262e-01],
                 [ 8.22161732e-05  1.00000000e+00 -2.70496536e-01  1.19516344e-04 -9.01633528e-03],
                 ...
                 [ 3.24459262e-01 -9.01633528e-03 -5.30876145e-01  5.51226603e-01 1.00000000e+00]],
    "best_twice_nll": 7.512073503579112,
    "goodness_of_fit": 0.8723436438567955
}
```

Configuration

Configuration structure

- configuration is built from blocks of settings, a JSON schema describes this structure
- `General` settings used for global parameters

general settings

list of phase space regions (channels)

list of samples (MC/data)

```
General:
  Measurement: "Example"
  InputPath: "input/{SamplePaths}"
  HistogramFolder: "histograms/"
  POI: "Signal_norm"

Regions:
  - Name: "Signal_region"
    Filter: "nJets >= 8"
    Variable: "jet_pt"
    Binning: [200, 300, 400, 500]

Samples:
  - Name: "Data"
    SamplePaths: "data.root"
    Tree: "events"
    Data: True

  - Name: "Signal"
    SamplePaths: "signal.root"
    Tree: "events"
    Weight: "weight_nominal"

  - Name: "Background"
    SamplePaths: "background.root"
    Tree: "events"
    Weight: "weight_nominal"

Systematics:
  - Name: "Luminosity"
    Up:
      Normalization: 0.05
    Down:
      Normalization: -0.05
    Samples: ["Signal", "Background"]
    Type: "Normalization"

  - Name: "ModelingVariation"
    Up:
      Tree: "events_up"
      Weight: "weight_modeling"
    Down:
      Tree: "events_down"
      Weight: "weight_modeling"
    Smoothing:
      Algorithm: "353QH, twice"
    Samples: "Background"
    Type: "NormPlusShape"

NormFactors:
  - Name: "Signal_norm"
    Samples: "Signal"
    Nominal: 1
    Bounds: [0, 10]
```

list of systematic uncertainties

list of normalization factors

Regions

- specify a list of phase space regions via `Regions`

```yaml
General:
  Measurement: "Example"
  InputPath: "input/{SamplePaths}"
  HistogramFolder: "histograms/"
  POI: "Signal_norm"

Regions:
  - Name: "Signal_region"
    Filter: "nJets >= 8"
    Variable: "jet_pt"
    Binning: [200, 300, 400, 500]

Samples:
  - Name: "Data"
    SamplePaths: "data.root"
    Tree: "events"
    Data: True

  - Name: "Signal"
    SamplePaths: "signal.root"
    Tree: "events"
    Weight: "weight_nominal"

  - Name: "Background"
    SamplePaths: "background.root"
    Tree: "events"
    Weight: "weight_nominal"

Systematics:
  - Name: "Luminosity"
    Up:
      Normalization: 0.05
    Down:
      Normalization: -0.05
    Samples: ["Signal", "Background"]
    Type: "Normalization"

  - Name: "ModelingVariation"
    Up:
      Tree: "events_up"
      Weight: "weight_modeling"
    Down:
      Tree: "events_down"
      Weight: "weight_modeling"
    Smoothing:
      Algorithm: "353QH, twice"
    Samples: "Background"
    Type: "NormPlusShape"

NormFactors:
  - Name: "Signal_norm"
    Samples: "Signal"
    Nominal: 1
    Bounds: [0, 10]
```

event selection requirements

variable to bin in

binning to use in histograms

Samples

- list all samples (Monte Carlo and data) via `Samples`

```yaml
General:
  Measurement: "Example"
  InputPath: "input/{SamplePaths}"
  HistogramFolder: "histograms/"
  POI: "Signal_norm"

Regions:
  - Name: "Signal_region"
    Filter: "nJets >= 8"
    Variable: "jet_pt"
    Binning: [200, 300, 400, 500]

Samples:
  - Name: "Data"
    SamplePaths: "data.root"
    Tree: "events"
    Data: True

  - Name: "Signal"
    SamplePaths: "signal.root"
    Tree: "events"
    Weight: "weight_nominal"

  - Name: "Background"
    SamplePaths: "background.root"
    Tree: "events"
    Weight: "weight_nominal"
```

```yaml
Systematics:
  - Name: "Luminosity"
    Up:
      Normalization: 0.05
    Down:
      Normalization: -0.05
    Samples: ["Signal", "Background"]

    Up:
      Tree: "events_up"
      Weight: "weight_modeling"
    Down:
      Tree: "events_down"
      Weight: "weight_modeling"
    Smoothing:
      Algorithm: "353QH, twice"
    Samples: "Background"
    Type: "NormPlusShape"

NormFactors:
  - Name: "Signal_norm"
    Samples: "Signal"
    Nominal: 1
    Bounds: [0, 10]
```

the paths combine to form the path per sample (can also use `RegionPaths` in there)

sample-specific settings

Systematics & normalization factors

- list instructions for systematic uncertainties via `Systematics` and define normalization factors with `NormFactors`

use same structure for up/down templates

defines a `HistFactory` `NormSys`

settings specified for a template override nominal settings used for sample

`NormSys` and `HistoSys`

which samples to act on

```yaml
Systematics:
  - Name: "Luminosity"
    Up:
      Normalization: 0.05
    Down:
      Normalization: -0.05
    Samples: ["Signal", "Background"]
    Type: "Normalization"

  - Name: "ModelingVariation"
    Up:
      Tree: "events_up"
      Weight: "weight_modeling"
    Down:
      Tree: "events_down"
      Weight: "weight_modeling"
    Smoothing:
      Algorithm: "353QH, twice"
    Samples: "Background"
    Type: "NormPlusShape"

NormFactors:
  - Name: "Signal_norm"
    Samples: "Signal"
    Nominal: 1
    Bounds: [0, 10]
```

Python API* and CLI

* only showing high-level API here, see e.g. `model_utils` for lower level utilities

The Python API - from config to fit results (1)

create template histograms from config instructions

apply optional post-processing, e.g. smoothing

visualize all templates: useful debugging utility

create a **`HistFactory`** workspace

```python
import cabinetry

config = cabinetry.configuration.load("config.yml")

# create template histograms
cabinetry.template_builder.create_histograms(config, method="uproot")

# perform histogram post-processing
cabinetry.template_postprocessor.run(config)

# visualize systematics templates
cabinetry.visualize.templates(config)

# build a workspace
ws = cabinetry.workspace.build(config)

# run a fit
model, data = cabinetry.model_utils.model_and_data(ws)
fit_results = cabinetry.fit.fit(model, data, minos=["Signal_norm"])

# visualize pulls and correlation matrix
cabinetry.visualize.pulls(fit_results)
cabinetry.visualize.correlation_matrix(fit_results, pruning_threshold=0.1)

# visualize the post-fit model prediction and data
cabinetry.visualize.data_MC(model, data, config=config, fit_results=fit_results)

# rank nuisance parameters by their impact on the POI
ranking_results = cabinetry.fit.ranking(model, data)
cabinetry.visualize.ranking(ranking_results)
```

The Python API - from config to fit results (2)

- **cabinetry** produces all instructions for building template histograms
  - histograms produced by backend (e.g. with **uproot**), called by **cabinetry**
  - can inject custom code, which **cabinetry** will call for histogram building instead
- custom histogramming code allows generation of templates from arbitrarily complex rules

```python
import cabinetry

config = cabinetry.configuration.load("config.yml")

# create template histograms
cabinetry.template_builder.create_histograms(config, method="uproot")
```

```python
import boost_histogram as bh
import numpy as np
import cabinetry

my_router = cabinetry.route.Router()

# define a custom template builder function that is executed for data samples
@my_router.register_template_builder(sample_name="Data")
def build_data_hist(reg: dict, sam: dict, sys: dict, tem: str) -> bh.Histogram:
    hist = bh.Histogram(
        bh.axis.Variable(reg["Binning"], underflow=False, overflow=False),
        storage=bh.storage.Weight(),
    )
    yields = np.asarray([100, 102, 103, 104])
    variance = np.asarray([0.1, 0.1, 0.1, 0.1])
    hist[...] = np.stack([yields, variance], axis=-1)
    return hist

cabinetry.template_builder.create_histograms(config, router=my_router)
```

The Python API - from config to fit results (3)

- some inference examples with `cabinetry`:

direct access to pdf and data (plus auxiliary data)

maximum likelihood fit

visualizations: see following slides

nuisance parameters ranked by impact

```python
import cabinetry

config = cabinetry.configuration.load("config.yml")

# create template histograms
cabinetry.template_builder.create_histograms(config, method="uproot")

# perform histogram post-processing
cabinetry.template_postprocessor.run(config)

# visualize systematics templates
cabinetry.visualize.templates(config)

# build a workspace
ws = cabinetry.workspace.build(config)

# run a fit
model, data = cabinetry.model_utils.model_and_data(ws)
fit_results = cabinetry.fit.fit(model, data, minos=["Signal_norm"])

# visualize pulls and correlation matrix
cabinetry.visualize.pulls(fit_results)
cabinetry.visualize.correlation_matrix(fit_results, pruning_threshold=0.1)

# visualize the post-fit model prediction and data
cabinetry.visualize.data_MC(model, data, config=config, fit_results=fit_results)

# rank nuisance parameters by their impact on the POI
ranking_results = cabinetry.fit.ranking(model, data)
cabinetry.visualize.ranking(ranking_results)
```

Template overrides

- users might want to build histograms in ways that cannot easily be supported
- `cabinetry` provides a decorator for user-provided functions to build histograms that match a pattern

```python
import boost_histogram as bh
import numpy as np
import cabinetry

my_router = cabinetry.route.Router()

# define a custom template builder function that is executed for data samples
@my_router.register_template_builder(sample_name="Data")
def build_data_hist(reg: dict, sam: dict, sys: dict, tem: str) -> bh.Histogram:
    hist = bh.Histogram(
        bh.axis.Variable(reg["Binning"], underflow=False, overflow=False),
        storage=bh.storage.Weight(),
    )
    yields = np.asarray([100, 102, 103, 104])
    variance = np.asarray([0.1, 0.1, 0.1, 0.1])
    hist[...] = np.stack([yields, variance], axis=-1)
    return hist

cabinetry.template_builder.create_histograms(config, router=my_router)
```

use function for samples with name `Data` (wildcards supported)

user-provided function returns histogram

Command line interface

- a **command line interface** exists for model building and inference
  - less control over execution, more immediate
  - useful to quickly debug unknown workspaces
  - documentation: command line interface

```
$ cabinetry templates config.yml
$ cabinetry postprocess config.yml
$ cabinetry workspace config.yml workspace.json
$ cabinetry fit --pulls --cormat --goodness_of_fit workspace.json
```

goodness-of-fit via saturated model