Nino-hist

Author: Nino

May 26, 2020

CONTENTS

CHAPTER

ONE

INSTALLATION

The recommended way of setting up a development environment:

```
python3 -m venv .env              # Make a new environment in ./.env/
source .env/bin/activate          # Use the new environment
pip install -r requirements.txt # Install the package requirements
pip install -e .                  # Install this package in editable mode
python -m ipykernel install --user --name nino-hist     # Install nino-hist jupyter␣
↪kernel
```

If you want to use Conda, go ahead. Also feel free to use a different directory name, etc. We will be requiring Python 3 here, at least 3.6 or better.

You'll need to run `source .env/bin/activate` if you open a new shell. You can use `deactivate` to turn off the environment in your current shell (*or just open a new one*).

The final line installs the package into your environment so that you can run the code from anywhere as long as the environment is activated.

If, while working on the project, you need any other python packages, such as for plotting, add them to the **requirements.txt** or in **setup.py**.

CHAPTER

TWO

USER GUIDE

2.1 Axes

Nino-hist provides powerful axes extension for boost-histogram. New features include name and title properties, Bool type axis, filling histogram by names. Let's see what's new in Nino-hist axes.

2.1.1 Name

Nino-hist supports customized names for the axes. You can create personalized axes for your histogram like this: `hist.axis.Regular(name='myRegular')`. The idea is this (and is taken directly from the Coffea project): All axes have a *required* name. These names are used (and generally required) throughout the interface.

Name is the identifier of an axis, which means that you can access this `Axis` object by name, but two axes cannot have the same name. Note that the named axes can only be created in `NamedHist` histogram. For a `NamedHist` instance, you can fill it by providing the value list, but you need to specify the assignment objects by their names. Let's see how to give a histogram a name and fill it according to names!

Supposed that you have installed Nino-hist, you can created a `NamedHist` object like this. If you want to change the hist packaging path (for example, using as sub-project), you can go to the `setup.cfg` and modify `[options.packages.find]` in it. Initialize it using a `Regular` axis and an `Integer` axis.

[1]:
```
import hist

h = hist.NamedHist(
    hist.axis.Regular(10, 0, 1, name='myRegular'),
    hist.axis.Integer(-1, 1, name='myInteger')
)

regular = [.15, .15, .25, .35, .55, .55]
integer = [-1, -1, 0, 0, 0, 0]
```

Then we can fill the `h` using the names of its axes.

[2]:
```
h.fill(myRegular=regular, myInteger=integer)
```

[2]:
```
NamedHist(
  Regular(10, 0, 1, metadata={'name': 'myRegular', 'title': None}),
  Integer(-1, 1, metadata={'name': 'myInteger', 'title': None}),
  storage=Double()) # Sum: 6.0
```

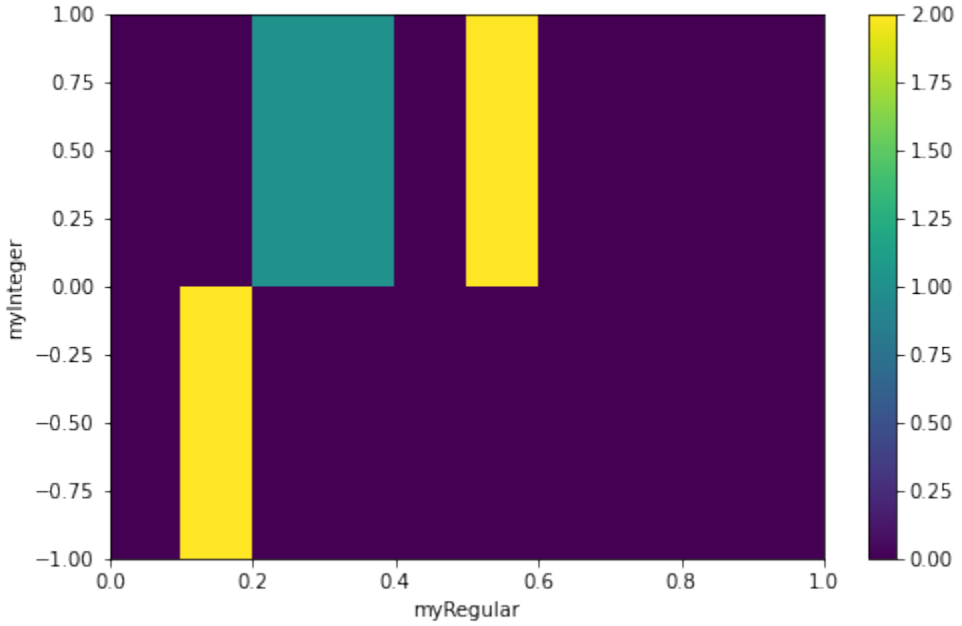
See what's print off contains `# Sum: 6.0`, which implies that 6 values are filled in `h`, i.e., (.15, -1), (.15, -1), (.25, 0), (.35, 0), (.55, 0), (.55, 0). Then verify them!

[3]:
```
h.view()
```

[3]:
```
array([[0., 0.],
       [2., 0.],
       [0., 1.],
       [0., 1.],
       [0., 0.],
       [0., 2.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.]])
```

See? The correspoding bins are filled. Let's visualize `h` to get more intuitive.

[4]:
```
import numpy as np
import matplotlib.pyplot as plt

fig, ax = plt.subplots(figsize=(8,5))
w, x, y = h.to_numpy()
mesh = ax.pcolormesh(x, y, w.T)
ax.set_xlabel(h.axes[0].metadata["name"])
ax.set_ylabel(h.axes[1].metadata["name"])
fig.colorbar(mesh)
fig.show()
```



Note that the names of axes should be the Python identifiers. Besides, we forbid to use '_' ahead. Thus, you aren't allowed to use names like `"\_\_name"`, `"\_name"`, `"name-a"`, etc. Let's see some examples of right and wrong names.

[5]:
```
# wrong names
assert hist.axis.Integer(-1, 1, name='my-Integer')\
or hist.axis.Integer(-1, 1, name='_myInteger')\
or hist.axis.Integer(-1, 1, name='__myInteger')
```

```
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
```

(continues on next page)

(continued from previous page)

```
<ipython-input-5-49f164ee495c> in <module>
      1 # wrong names
----> 2 assert hist.axis.Integer(-1, 1, name='my-Integer')\
      3 or hist.axis.Integer(-1, 1, name='_myInteger')\
      4 or hist.axis.Integer(-1, 1, name='__myInteger')

~/Documents/GitHub/Nino-hist/src/hist/_internal/axis.py in __init__(self, start, stop,
↪ name, title, underflow, overflow, growth)
    189         if re.match(r"^[0-9a-zA-Z][0-9a-zA-Z_]*", name).group() == name:
    190             metadata = dict(name=name)
--> 191         else: raise Exception("Name should be a Python Identifier.")
    192         metadata["title"] = title
    193         super().__init__(

Exception: Name should be a Python Identifier.
```

[6]:
```
# correct names
assert hist.axis.Integer(-1, 1, name='my_Integer')\
and hist.axis.Integer(-1, 1, name='myInteger_')\
and hist.axis.Integer(-1, 1, name='myInteger_0')\
and hist.axis.Integer(-1, 1, name='0_myInteger')
```

In addition to a valid naming convention, we also need to be careful that there cannot be axes with the same names in a histogram, otherwise this will cause ambiguity when filling.

[7]:
```
h = hist.NamedHist(
    hist.axis.Regular(10, 0, 1, name='myRegular'),
    hist.axis.Regular(10, -1, 1, name='myRegular')
)
```

```
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
<ipython-input-7-aad616514c91> in <module>
      1 h = hist.NamedHist(
      2     hist.axis.Regular(10, 0, 1, name='myRegular'),
----> 3     hist.axis.Regular(10, -1, 1, name='myRegular')
      4 )

~/Documents/GitHub/Nino-hist/src/hist/core.py in __init__(self, *args, **kwargs)
     12         names = list(l.name for l in self.axes)
     13         if len(set(names)) != len(self.axes):
---> 14             raise Exception("Name duplicated.")

Exception: Name duplicated.
```

2.1.2 Title

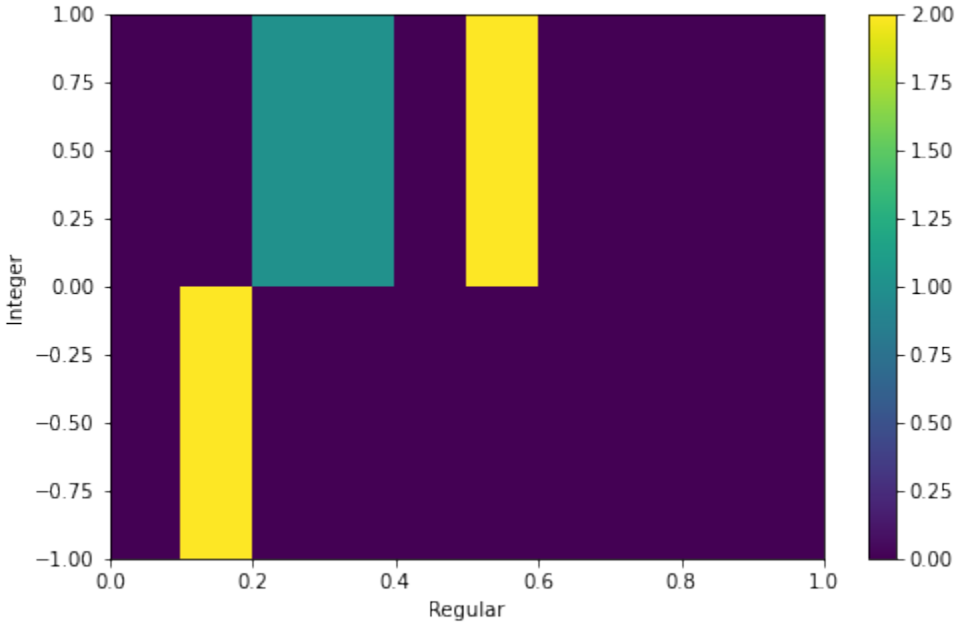
Nino-hist contains a property named title for axes. You can adjust titles of axes directly, or assign the titles to axes when initializing them. We change the titles of histogram named `h` we created above and use it to re-draw the figure.

[8]:
```
import numpy as np
import matplotlib.pyplot as plt

h.axes[0].metadata["title"] = "Regular"
h.axes[1].metadata["title"] = "Integer"
```

(continues on next page)

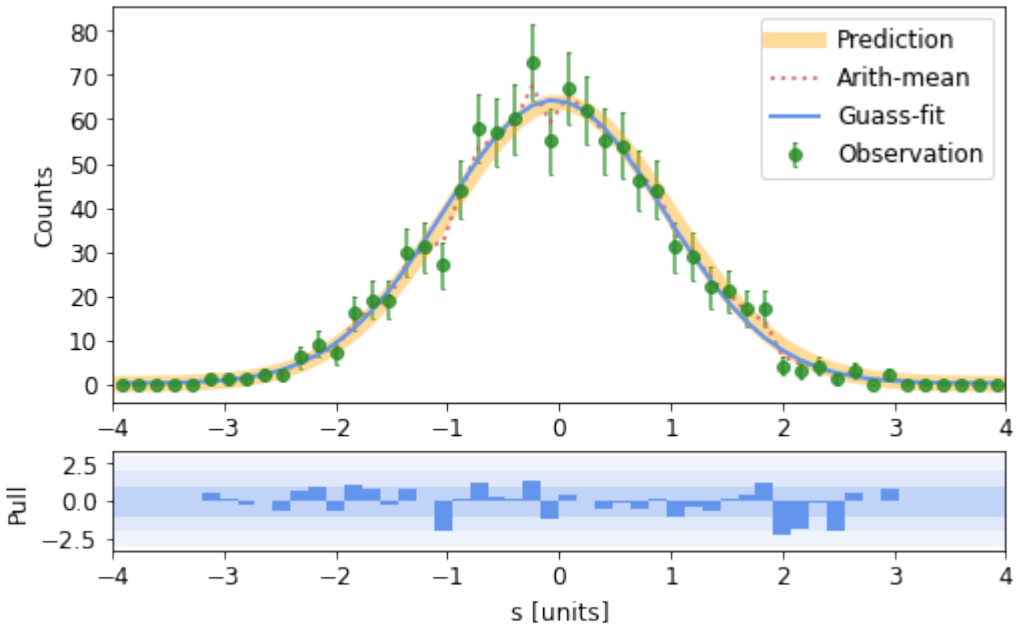
(continued from previous page)

```
fig, ax = plt.subplots(figsize=(8,5))
w, x, y = h.to_numpy()
mesh = ax.pcolormesh(x, y, w.T)
ax.set_xlabel(h.axes[0].metadata["title"])
ax.set_ylabel(h.axes[1].metadata["title"])
fig.colorbar(mesh)
fig.show()
```



Title is not like name property, which is a unique representation of an axes, it is often used to draw plots, such as pull plot. Let's see an example of pull plot. (Note: pull plot is not the new feature of Nino-hist for axes, you can see more details about pull plot latter.)

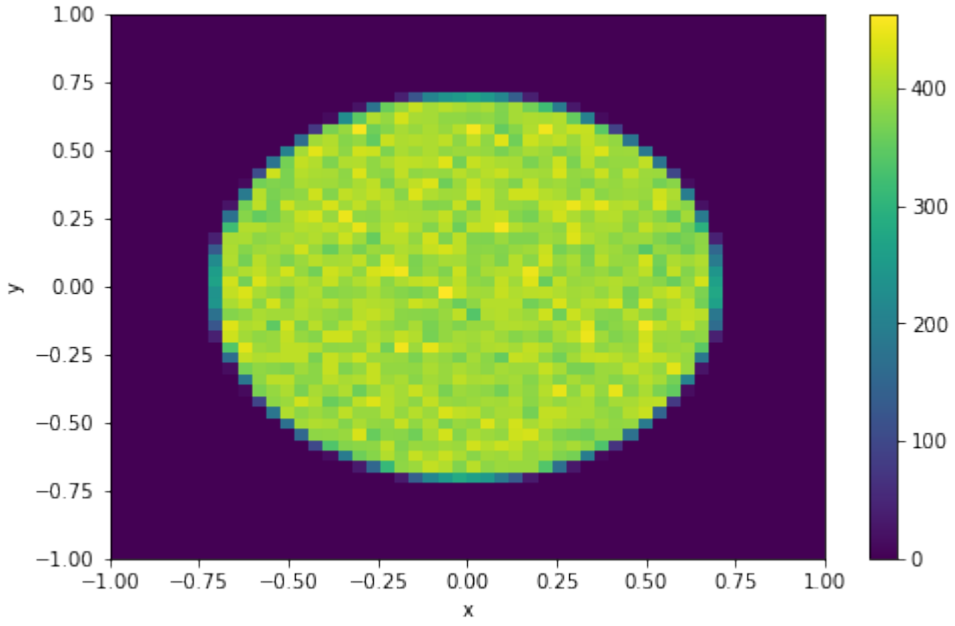
[9]:
```
import hist
import numpy as np
import matplotlib.pyplot as plt


h = hist.Hist(
    hist.axis.Regular(50, -4, 4, name="S", title="s [units]", underflow=False,␣
↪overflow=False)
)

data = np.random.normal(size=1_000)
h.fill(data)

def pdf(x, a=1/np.sqrt(2*np.pi), x0=0, sigma=1, offset=0):
    return a * np.exp(-(x-x0)**2/(2*sigma**2)) + offset

fig = plt.figure(figsize=(8, 5))
fig, _, _ = h.pull_plot(pdf, size="m", fig=fig)
```



2.1.3 Bool axis

Boost-histogram doesn't support `Bool` axis directly. The often used method likes this.

[10]:
```
import boost_histogram as bh
import numpy as np
import matplotlib.pyplot as plt

h = bh.Histogram(
    bh.axis.Regular(50, -1, 1),
    bh.axis.Regular(50, -1, 1),
    bh.axis.Integer(0, 2, underflow=False, overflow=False),
)

x, y = np.random.random_sample([2, 1_000_000])*2 - 1
valid = (x**2 + y**2) < .5

h.fill(x, y, valid)
valid_only = h[:, :, bh.loc(True)]

fig, ax = plt.subplots(figsize=(8,5))
W, X, Y = valid_only.to_numpy()
mesh = ax.pcolormesh(X, Y, W.T)
ax.set_xlabel('x')
ax.set_ylabel('y')
fig.colorbar(mesh)
fig.show()
```



The `Integer` axis is used as a `Bool` axis to screen invalid (x, y) values. Boost-histogram usually uses `Integer` or `Regular` axes to represent `Bool` axes. While in Nino-hist, you can directly call `Bool` axis object, which is more explicit for coding.

[11]:
```
import hist

h = hist.NamedHist(
    hist.axis.Bool(name="A", title="A [units]"),
    hist.axis.Bool(name="B", title="B [units]"),
    hist.axis.Bool(name="C", title="C [units]")
)

valid_a = [True, True, True, True]
valid_b = [True, True, False, False]
valid_c = [False, False, False, True]

h.fill(B=valid_b, A=valid_a, C=valid_c)

h.view()
```

[11]:
```
array([[[0., 0.],
        [0., 0.]],

       [[1., 1.],
        [2., 0.]]])
```

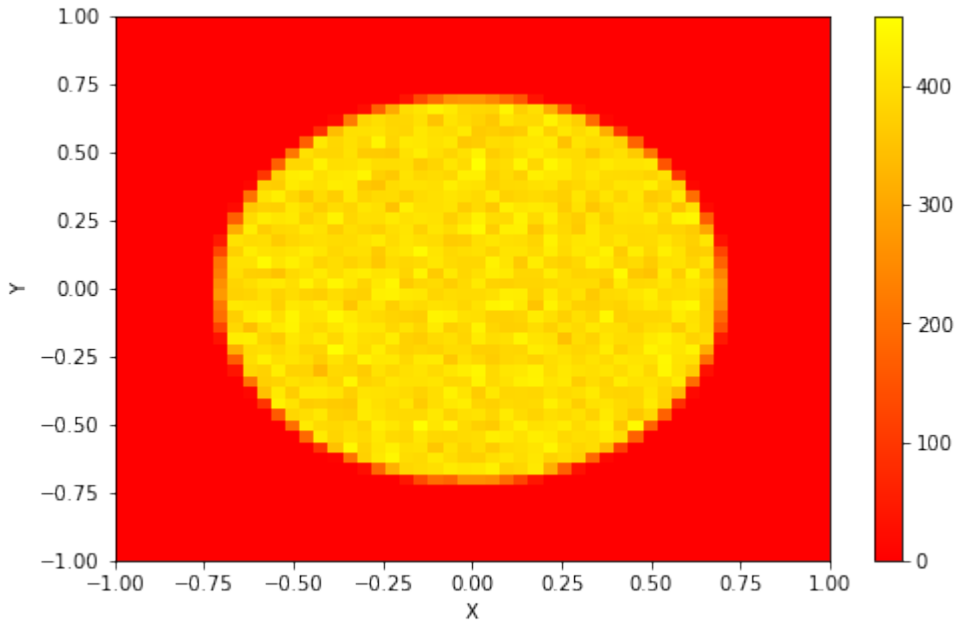
This is a sample 3-D `NameHist` based on three `Bool` axes, having 8 bins. When filled with the lists above, 3 bins have values in their bins, i.e., (A=True, B=True, C=False), (A=True, B=False, C=False), (A=True, B=False, C=True). As shown in `h.view()`, our `Bool` works well and more convenient for interdisciplinary research. In the end of this part, let's re-implemented the figure above.

[12]:
```
import boost_histogram as bh
import hist
import numpy as np
```

(continues on next page)

(continued from previous page)

```
import matplotlib.pyplot as plt


h = hist.NamedHist(
    hist.axis.Regular(50, -1, 1, name="X"),
    hist.axis.Regular(50, -1, 1, name="Y"),
    hist.axis.Bool(name="V"),
)

x, y = np.random.random_sample([2, 1_000_000])*2 - 1
valid = (x**2 + y**2) < .5
h.fill(Y=y, X=x, V=valid)

valid_only = h[:, :, bh.loc(True)]

fig, ax = plt.subplots(figsize=(8,5))
W, X, Y = valid_only.to_numpy()
mesh = ax.pcolormesh(X, Y, W.T, cmap='autumn')
ax.set_xlabel('X')
ax.set_ylabel('Y')
fig.colorbar(mesh)
fig.show()
```



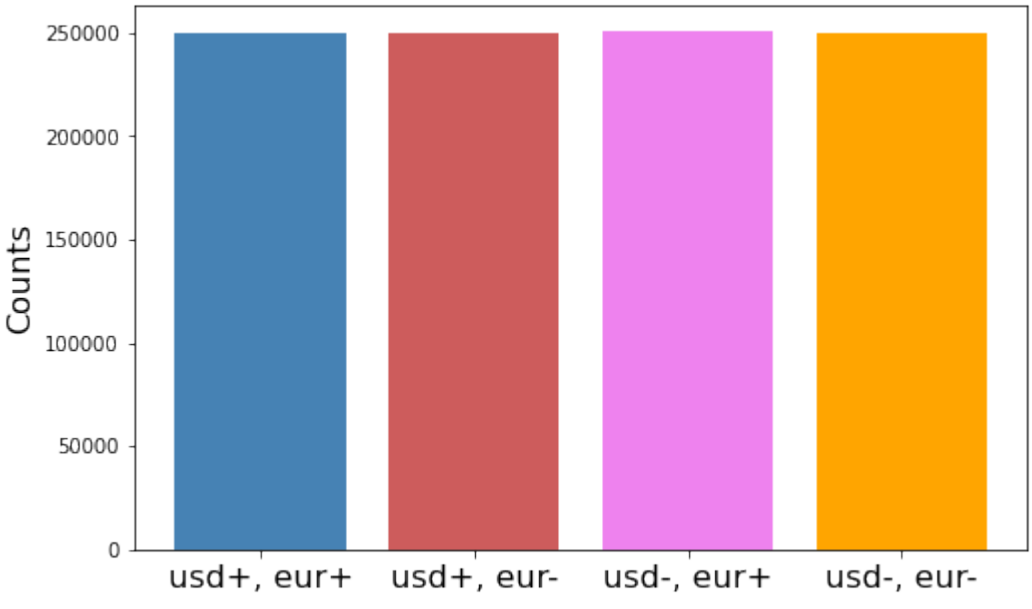
2.2 NamedHist

Nino-hist provides a customized histogram object named `NamedHist` for boost-histogram. The new feature of `NamedHist` is to fill the axes with their names in the random order. Let's see how it works in Nino-hist `NamedHist`.

Supposed that you have installed Nino-hist, you can created a `NamedHist` object like this. If you want to change the hist packaging path (for example, using as sub-project), you can go to the `setup.cfg` and modify `[options.packages.find]` in it.

Now, let's flip two coins.

[1]:
```
import hist
import numpy as np
import matplotlib.pyplot as plt

h = hist.NamedHist(
    hist.axis.Bool(name='USD', title='usd font'),
    hist.axis.Bool(name='EUR', title='eur font')
)
usd = np.random.rand(1_000_000) > 0.5
eur = np.random.rand(1_000_000) > 0.5

h.fill(USD=usd, EUR=eur)

bar = [h[0, 0], h[0, 1], h[1, 0], h[1, 1]]
x = range(len(bar))
bar_color = ['steelblue', 'indianred', 'violet', 'orange']
fig, ax = plt.subplots(figsize=(8,5))
ax.bar(x, bar, color=bar_color)
plt.xticks(x, ("usd+, eur+", "usd+, eur-", "usd-, eur+",\
                                    "usd-, eur-"), size=16)
plt.ylabel("Counts", size=16)
fig.show()
```

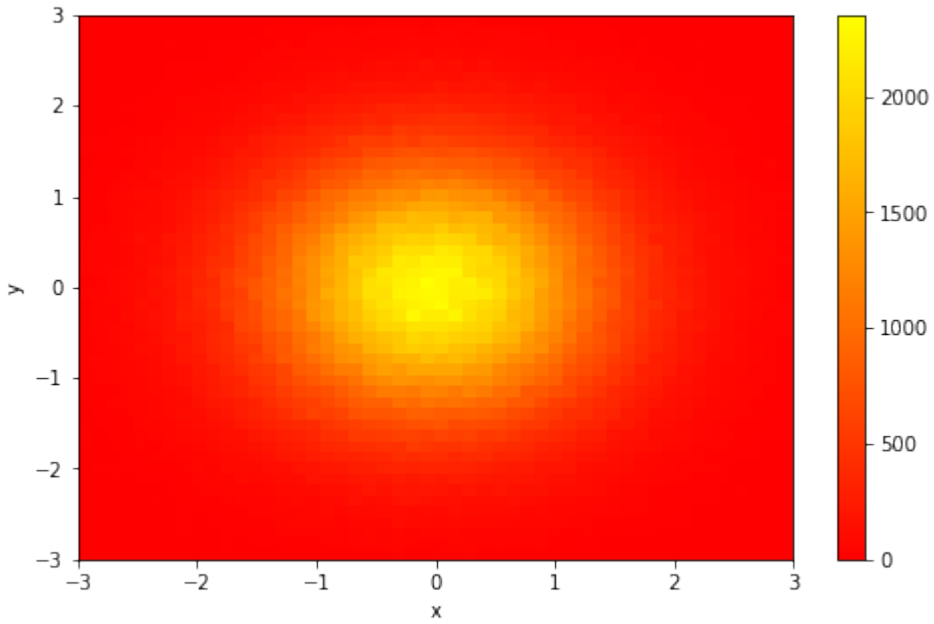


Shown in the figure above, the four possibilities would have almost same counts when generating seeds randomly.

`NamedHist` is simple and friendly to newers. In `NamedHist`, we do not have to follow the order of axes initialization,

to remember their names is just ok. Let's see another example of `NamedHist`.

[2]:
```
import boost_histogram as bh
import hist
import numpy as np
import matplotlib.pyplot as plt


h = hist.NamedHist(
    hist.axis.Regular(50, -3, 3, name="x"),
    hist.axis.Regular(50, -3, 3, name="y"),
)

x = np.random.randn(1_000_000)
y = np.random.randn(1_000_000)
h.fill(y=y, x=x)

fig, ax = plt.subplots(figsize=(8,5))
w, x, y = h.to_numpy()
mesh = ax.pcolormesh(x, y, w.T, cmap='autumn')
ax.set_xlabel('x')
ax.set_ylabel('y')
fig.colorbar(mesh)
fig.show()
```



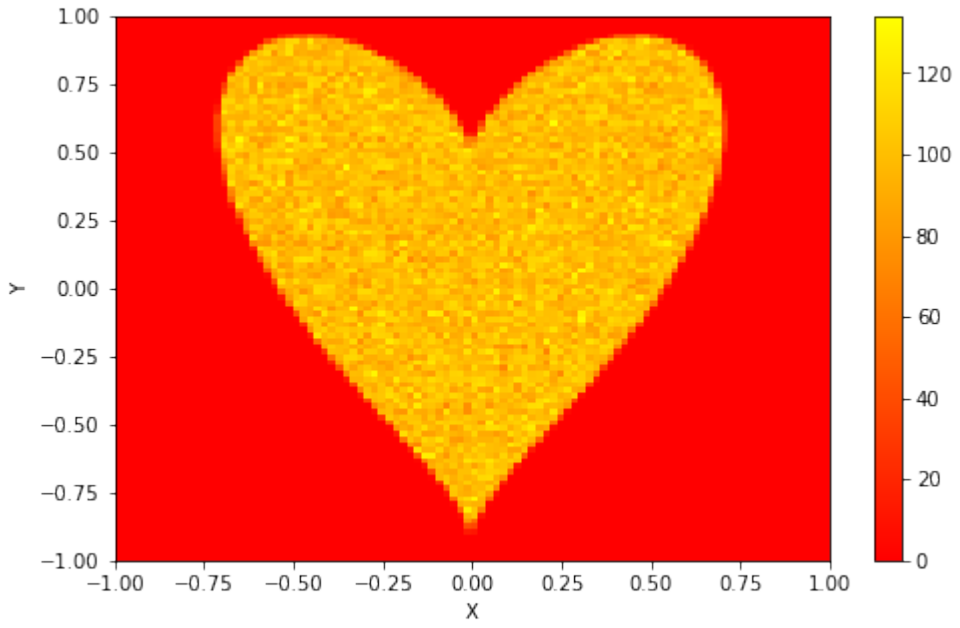
You can also use names to get and set items like this.

[3]:
```
import boost_histogram as bh
import hist
import numpy as np
import matplotlib.pyplot as plt


h = hist.NamedHist(
    hist.axis.Regular(100, -1, 1, name="X"),
```

(continues on next page)

(continued from previous page)

```
    hist.axis.Regular(100, -1, 1, name="Y"),
    hist.axis.Bool(name="V"),
)

x, y = np.random.random_sample([2, 1_000_000])*2 - 1
valid = np.abs(x)**2 + (y + .2 - np.power(np.abs(x), 2/3))**2 < .5
h.fill(Y=y, X=x, V=valid)

valid_only = h[{'V':bh.loc(True)}]    # this will cause dimentionality reduction
# valid_and_invalid = h[{'V':slice(None, None, bh.sum)}]
fig, ax = plt.subplots(figsize=(8,5))
W, X, Y = valid_only.to_numpy()
mesh = ax.pcolormesh(X, Y, W.T, cmap='autumn')
ax.set_xlabel('X')
ax.set_ylabel('Y')
fig.colorbar(mesh)
fig.show()
```



As you can see, name is a import property for `NamedHist` like the id of axes. You can assign the names to the axes in your histogram and use it to fill, to get item, etc. Have a try by yourself!

2.3 Hist

2.3.1 General

Nino-hist provides a customized histogram object named `Hist` for boost-histogram. The new feature of `Hist` is drawing pull plot, in which barring ranges, calculate arithmetic mean, fitting observations are conducted. Let's see how it works in Nino-hist `Hist`.

Initialize a `Hist` instance like this and fill. We are going to see some new features of `Hist`.

[1]:
```
import hist
import numpy as np
import matplotlib.pyplot as plt

h = hist.Hist(
    hist.axis.Regular(50, -4, 4, name="S", title="s [units]", underflow=False,␣
↪overflow=False)
)

data = np.random.normal(size=1_000)

h.fill(data)
```

[1]:
```
Hist(Regular(50, -4, 4, underflow=False, overflow=False, metadata={'name': 'S', 'title
↪': 's [units]'}), storage=Double()) # Sum: 1000.0
```

2.3.2 Pull plot

Pull plot only accept a callable parameter. If you put another types in it, it will raise a type error. Then let's see what happens if we use bool type as input.

[2]:
```
nino = True
ax1, ax2 = h.pull_plot(nino)      # "Only callable parameter is accepted in pull plot."
```

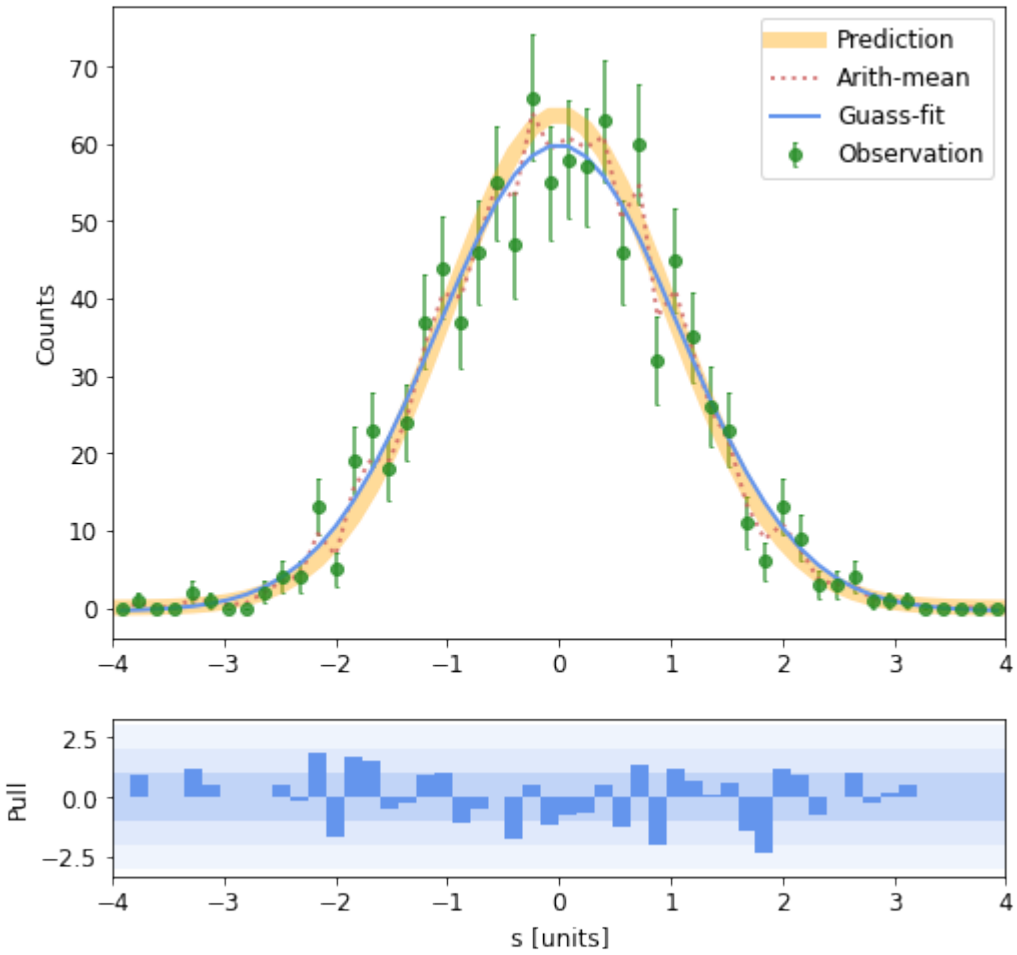
```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-2-6bc8622df33a> in <module>
      1 nino = True
----> 2 ax1, ax2 = h.pull_plot(nino)      # "Only callable parameter is accepted in␣
↪pull plot."

~/Documents/GitHub/Nino-hist/src/hist/general.py in pull_plot(self, func, fig, ax,␣
↪pull_ax, size, theme)
     22         if callable(func) == False:
     23             raise TypeError(
---> 24                 "Callable parameter func is supported in pull plot."
     25             )
     26

TypeError: Callable parameter func is supported in pull plot.
```

Opps… As expected, this function was not called successfully. To get back to the point, we use the callable object as a parameter.

Define the function (the assumed distribution of observations) you are going to use pull plot to compare.

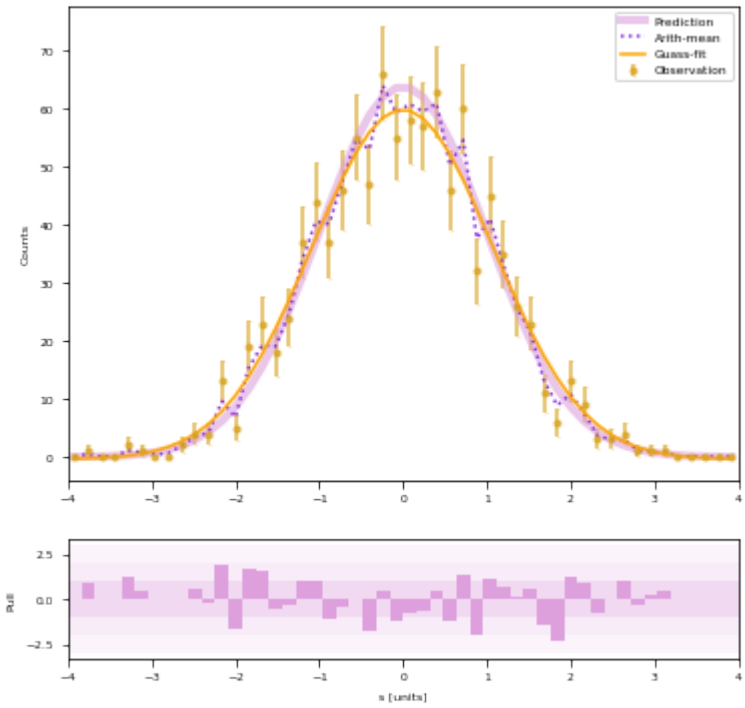
[3]:
```
def pdf(x, a=1/np.sqrt(2*np.pi), x0=0, sigma=1, offset=0):
    return a * np.exp(-(x-x0)**2/(2*sigma**2)) + offset
```

[4]:
```
fig, ax, pull_ax = h.pull_plot(pdf, size='m', theme='Chrome')
```

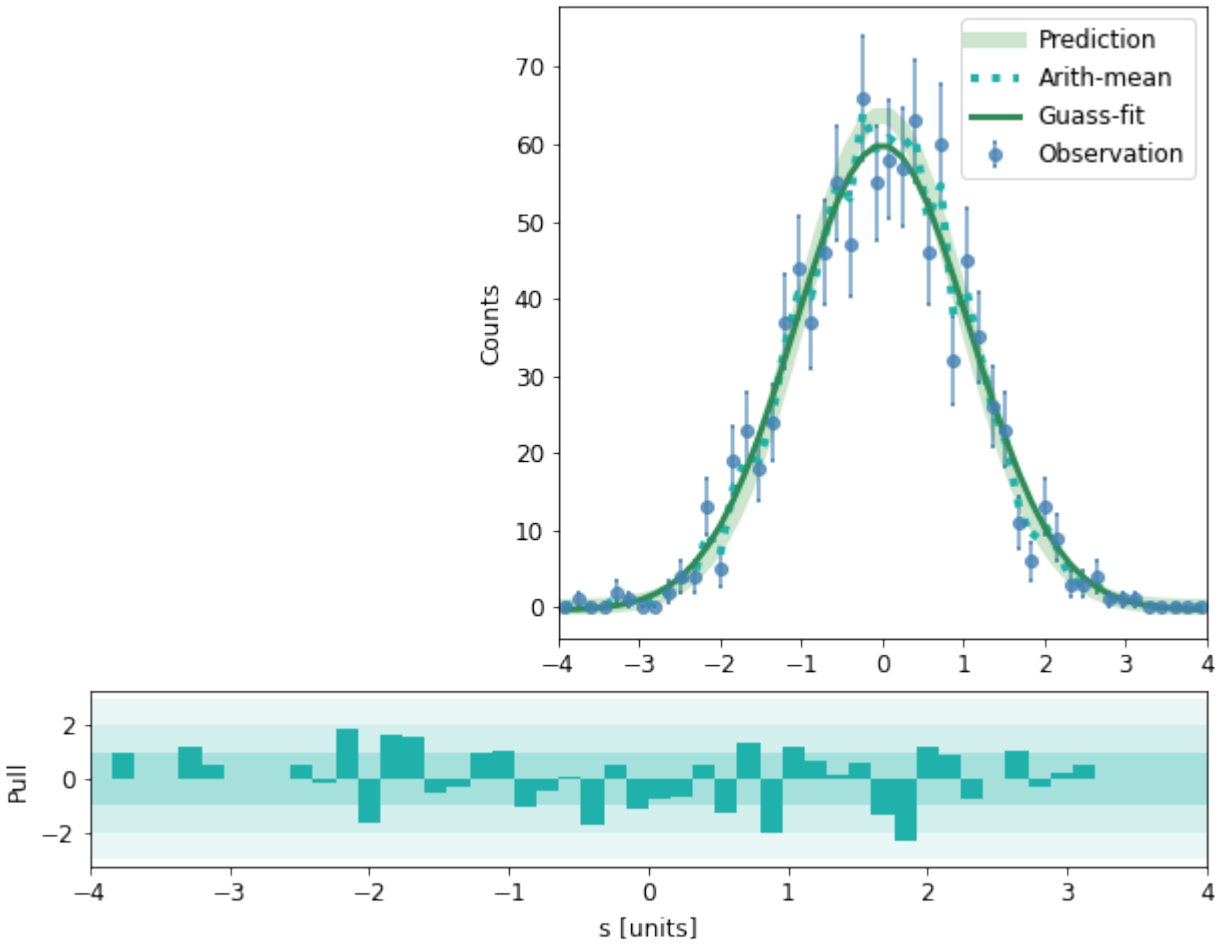


It works! You can see the pull plot above and it returns the figure and axis objects of this plot like this. So you can save or load it anywhere (using pickle if needed).

[5]:
```
fig = plt.figure(figsize=(6, 6))
fig, _, _ = h.pull_plot(pdf, size="s", fig=fig, theme='spring')
fig.savefig("img/fig-img.png")
```



[6]:
```
fig = plt.figure(figsize=(10, 10))
grid = fig.add_gridspec(5, 5, wspace=0.3, hspace=0.3)
ax = fig.add_subplot(grid[0:3, 2:])
pull_ax = fig.add_subplot(grid[3:4, :], sharex=ax)

h.pull_plot(pdf, size="m", fig=fig, ax=ax, pull_ax=pull_ax, theme='winter')
fig.savefig("img/ax-img.png")
```



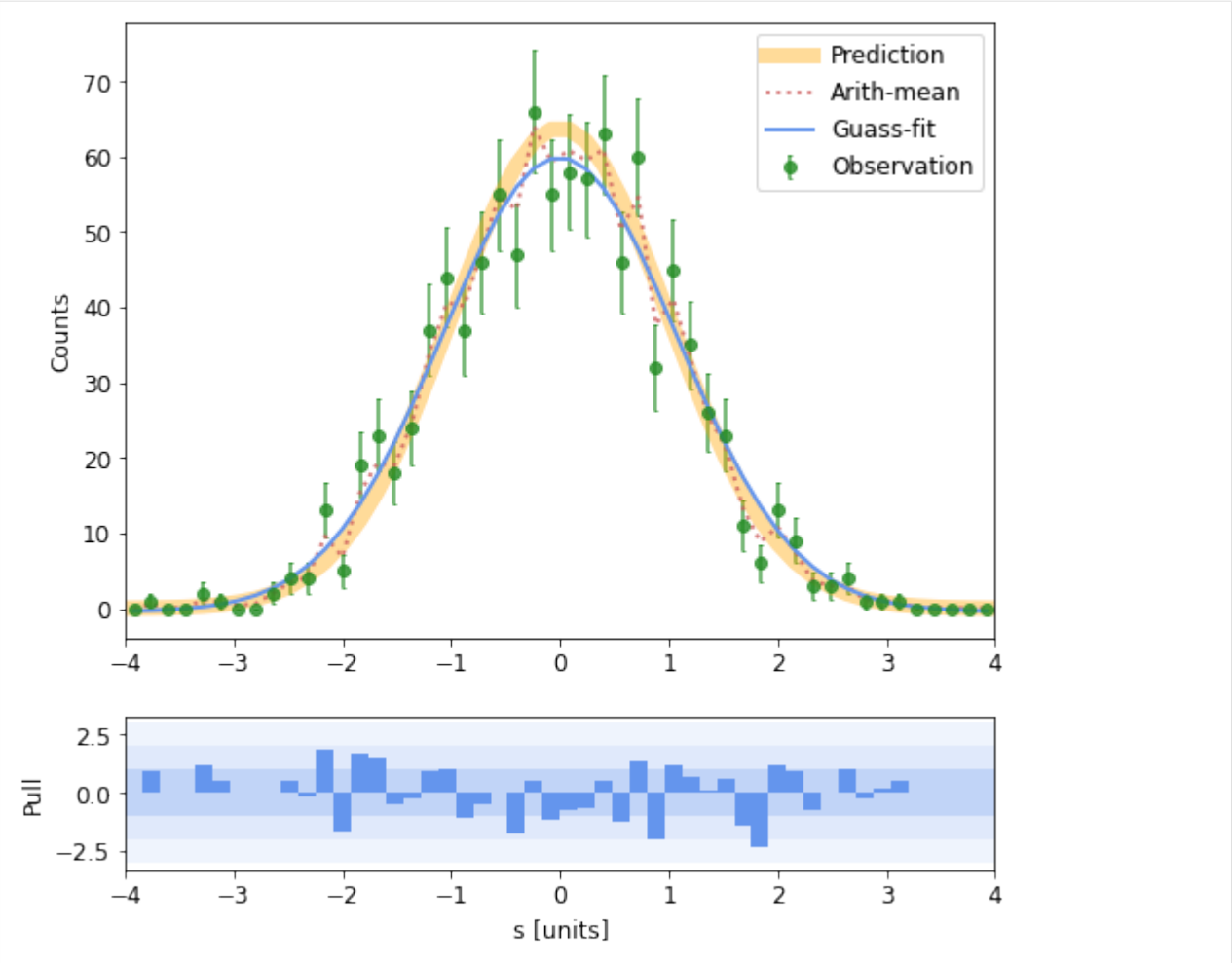
Nino-hist is a user-friendly tool, which mean that you can easily use it even you are not a professionally know Python. For this propose, `Themes` are provided for users. The possible options for `theme` parameter include: 'chrome', 'dark', 'light', 'spring', 'summer', 'autumn', 'winter', 'cool', 'hot'. We are going to enrich these themes in future. If you expect to use pull plot more professionally, you can view the pull plot pro in the next part for help.

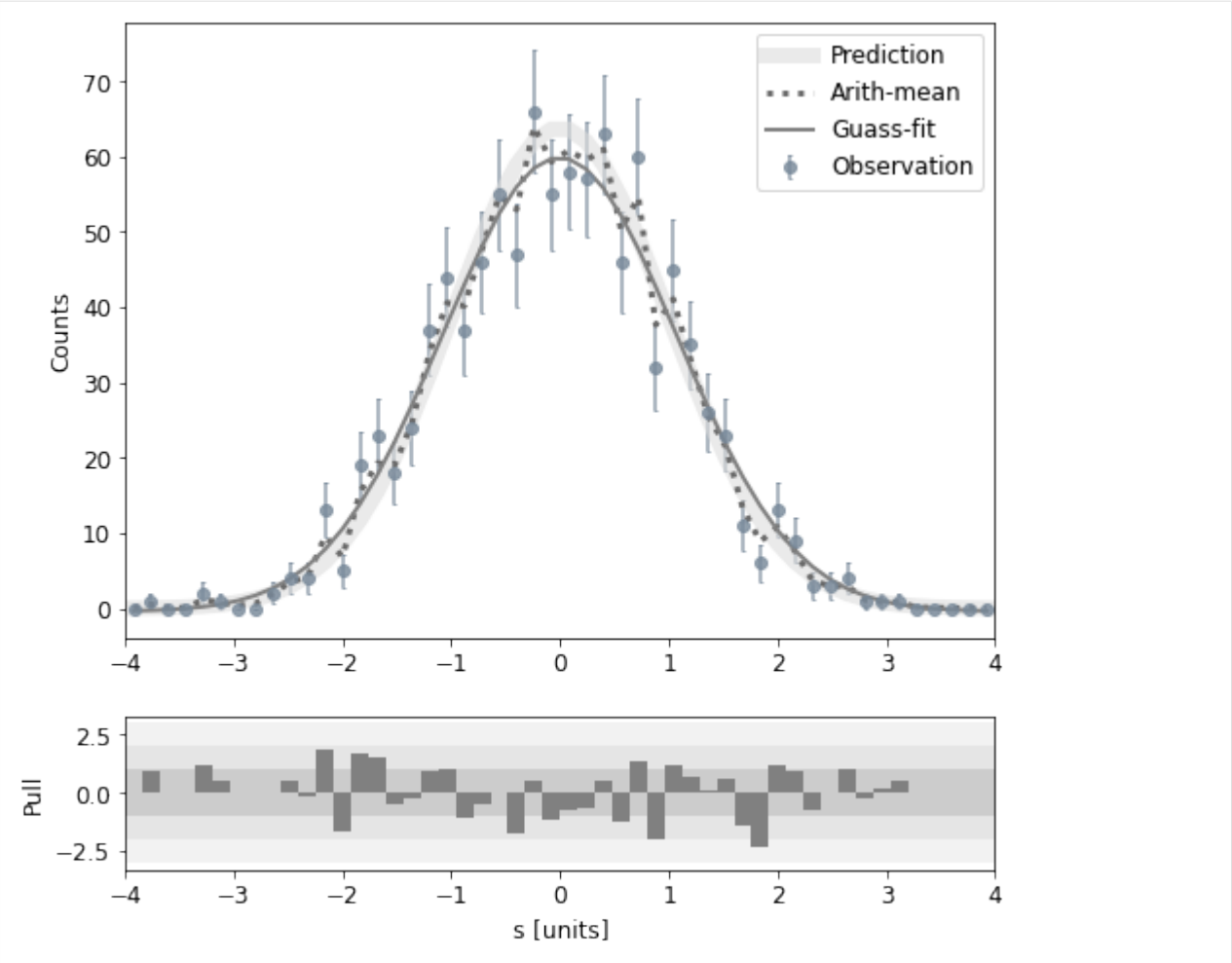
If convenience is the main focus, you can create a pull plot with only a `callable func` parameter, as the `size` and `theme` parameters have default setting, i.e., 'large' and 'chrome'. For `size`, you will have 'huge', 'large', 'medium', 'small', 'tiny' options. No requirements on lower or upper, and even the first letters are acceptable.

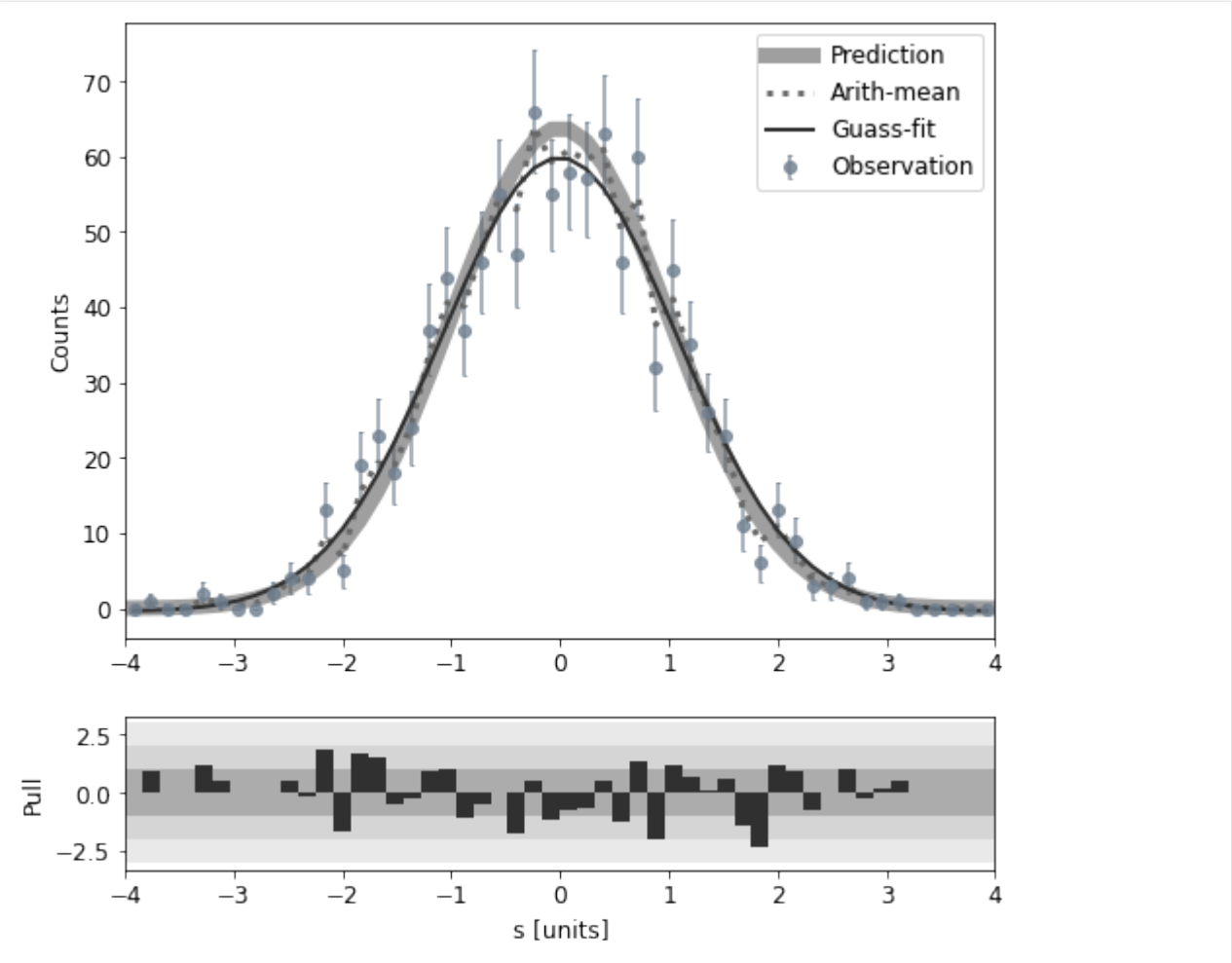
Note that if you are using you own figure rather than using the figure created by default, the `size` represents the plot size instead of figure size. So inputting a huge figure while using `size='small'` will result in large blank in your returned figure. You are supposed to choose a `size` near your figure size or adjust figure size aforehand.

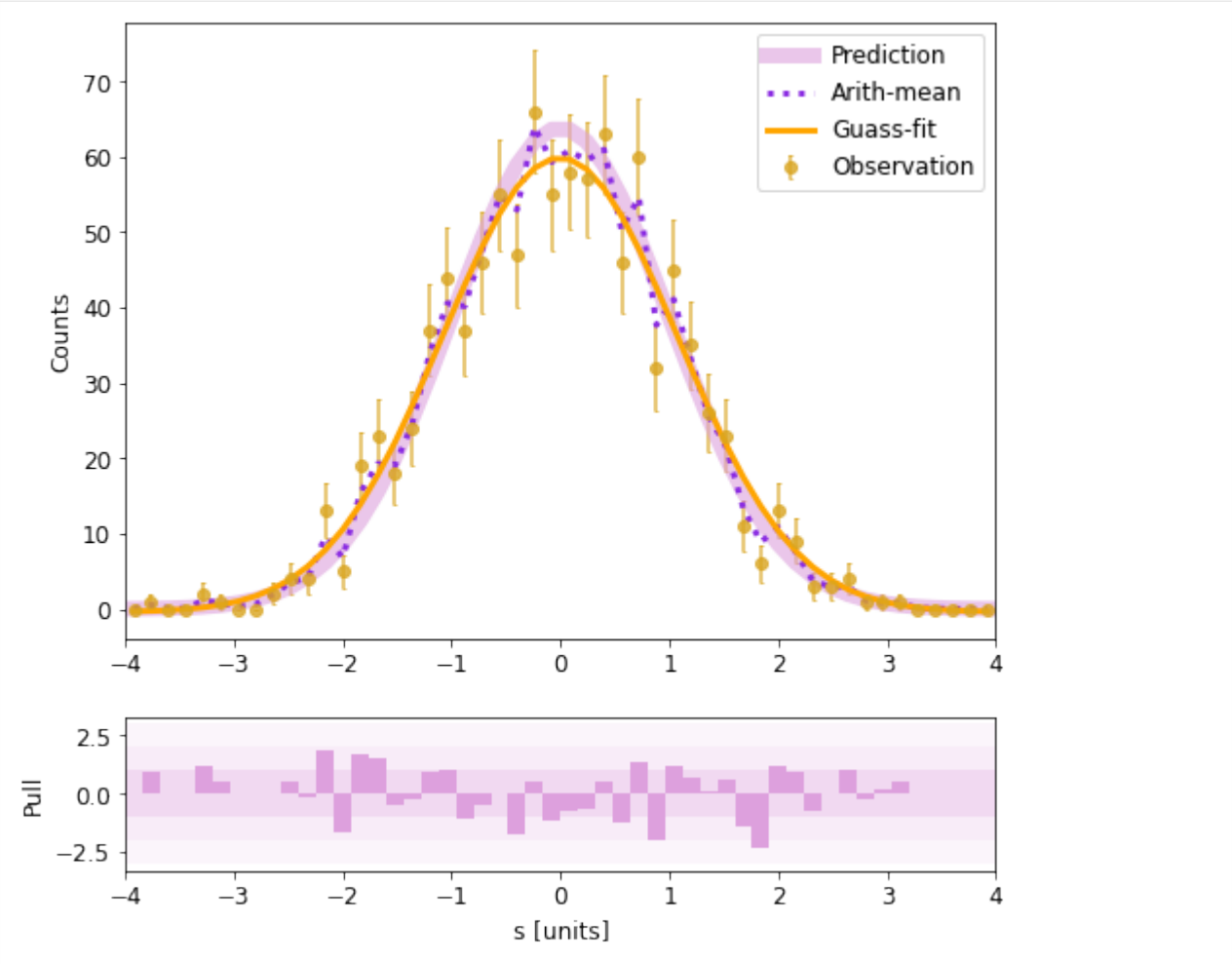
Iterate through all the theme options to produce beautiful images!

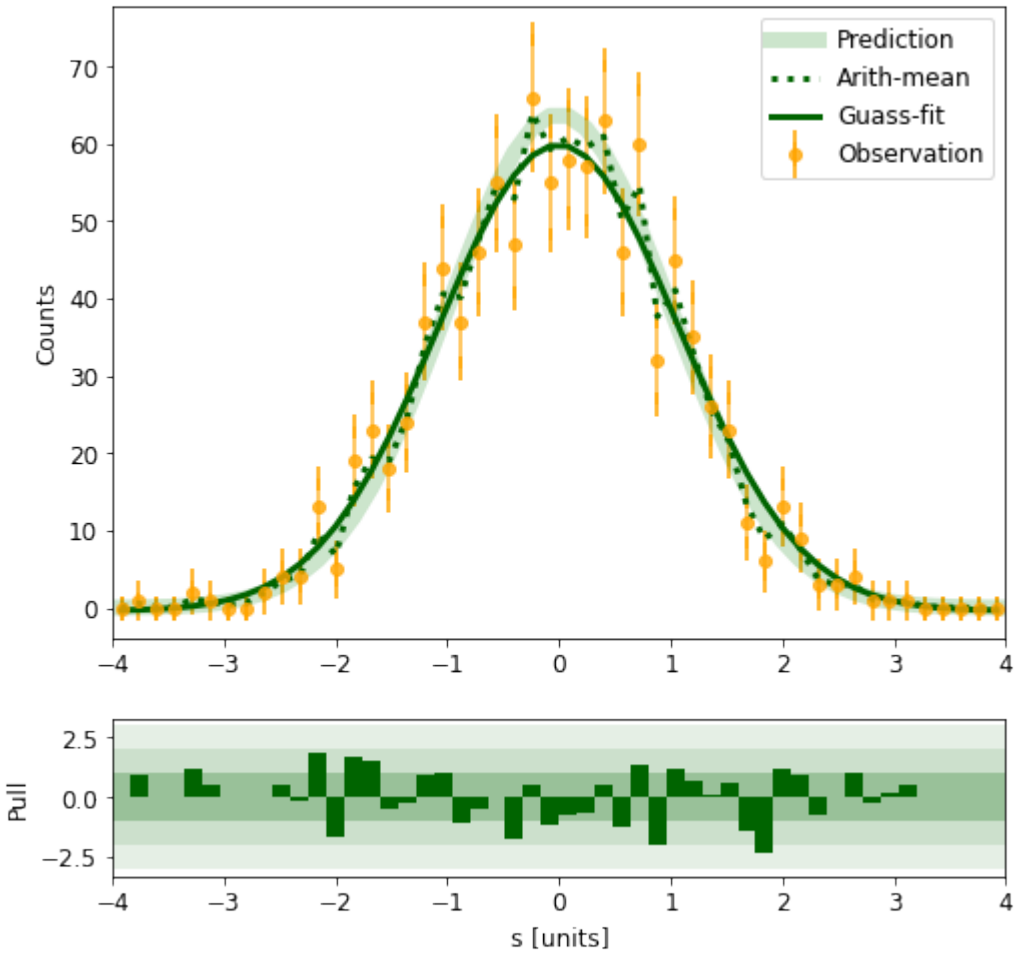
[7]:
```
themes=['Chrome', 'Light', 'Dark', 'Spring',\
        'Summer', 'Autumn', 'Winter', 'Cool', 'Hot']

for t in themes:
    h.pull_plot(pdf, size="m", theme=t)
    plt.savefig(f"./img/{t}.png")
```

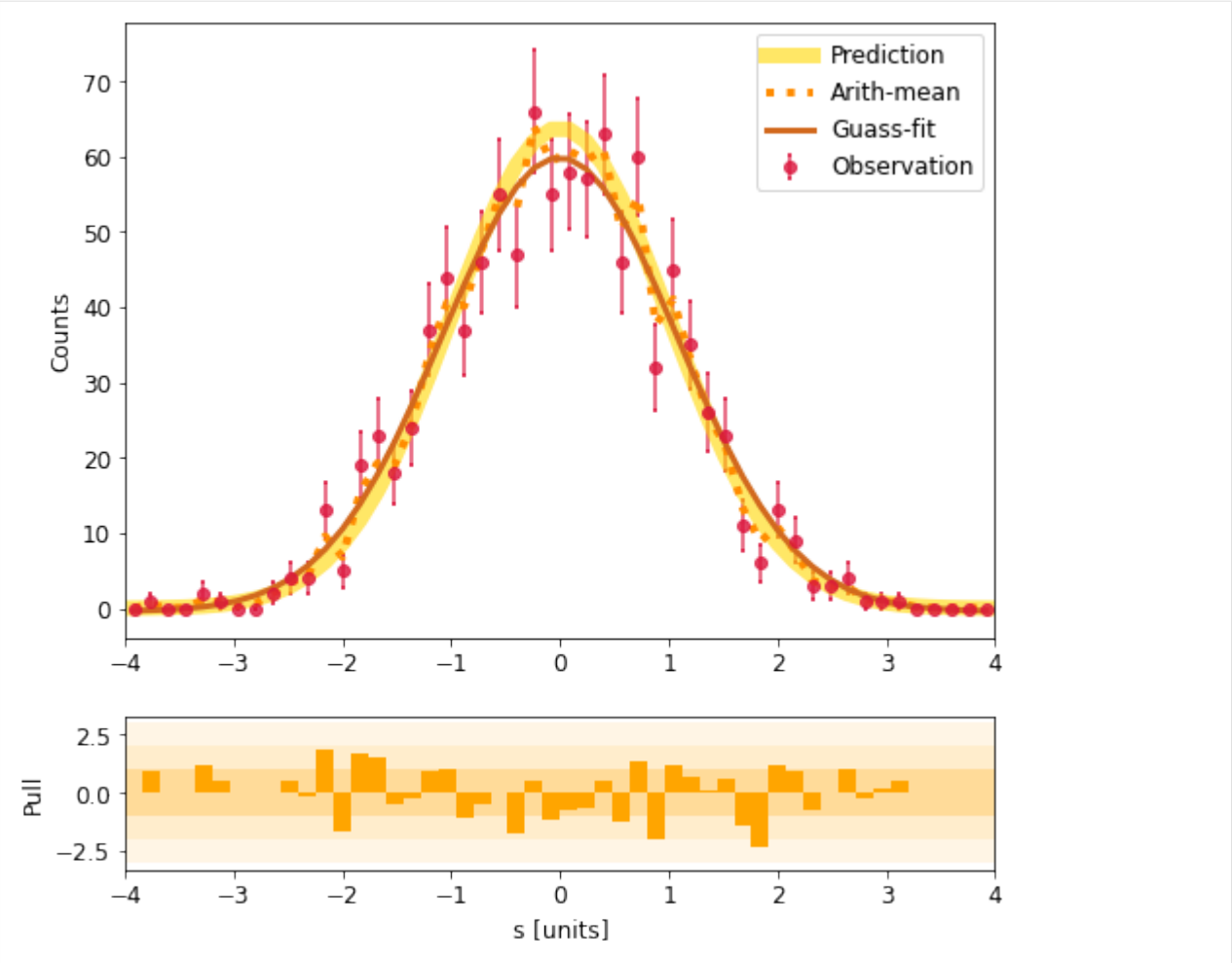

Prediction
Arith-mean
Guass-fit
Observation
Counts
70
60
50
40
30
20
10
0
−4
−3
−2
−1
0
1
2
3
4
Pull
2.5
0.0
−2.5
s [units]

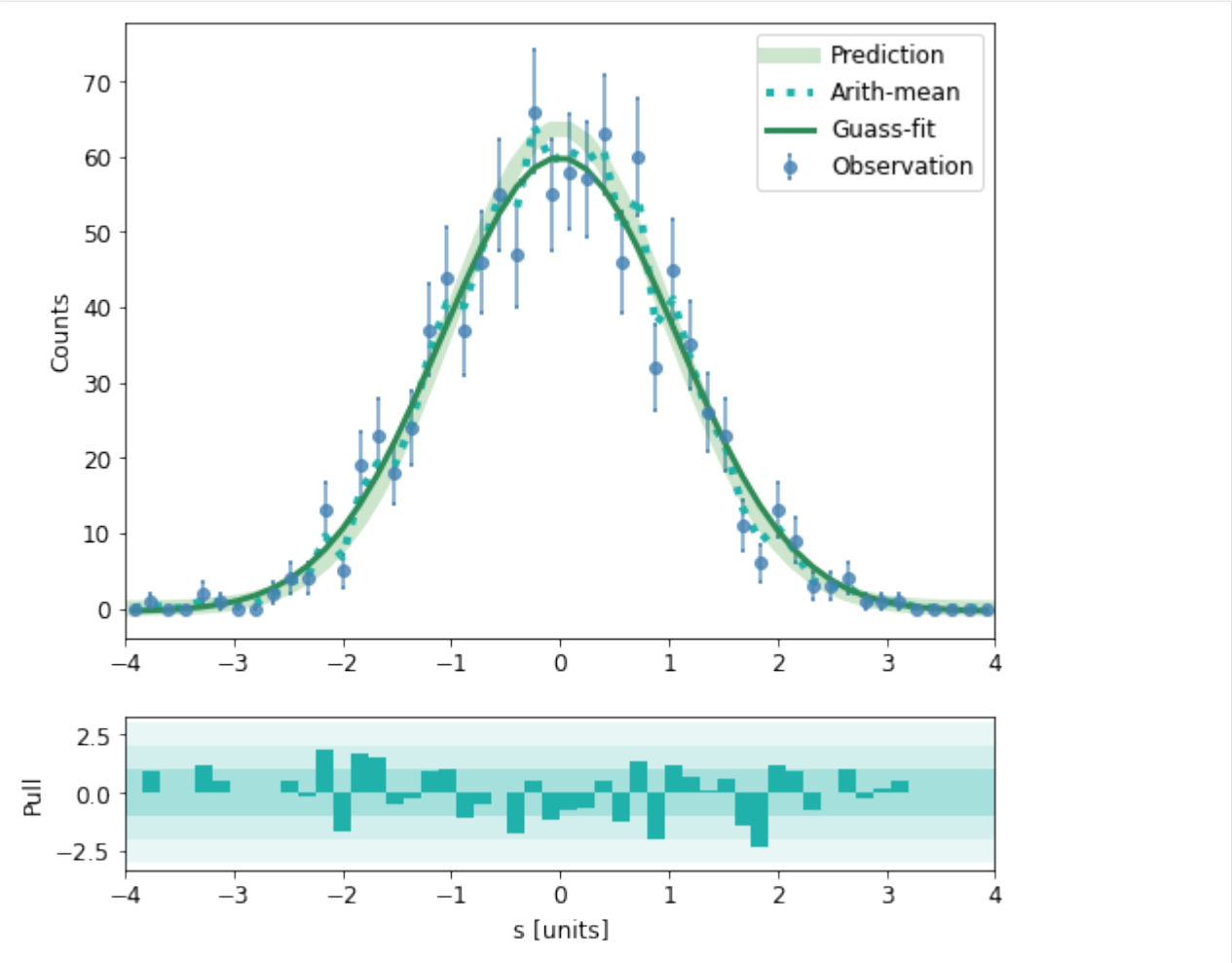

Prediction
Arith-mean
Guass-fit
Observation
Counts
70
60
50
40
30
20
10
0
−4
−3
−2
−1
0
1
2
3
4
Pull
2.5
0.0
−2.5
s [units]

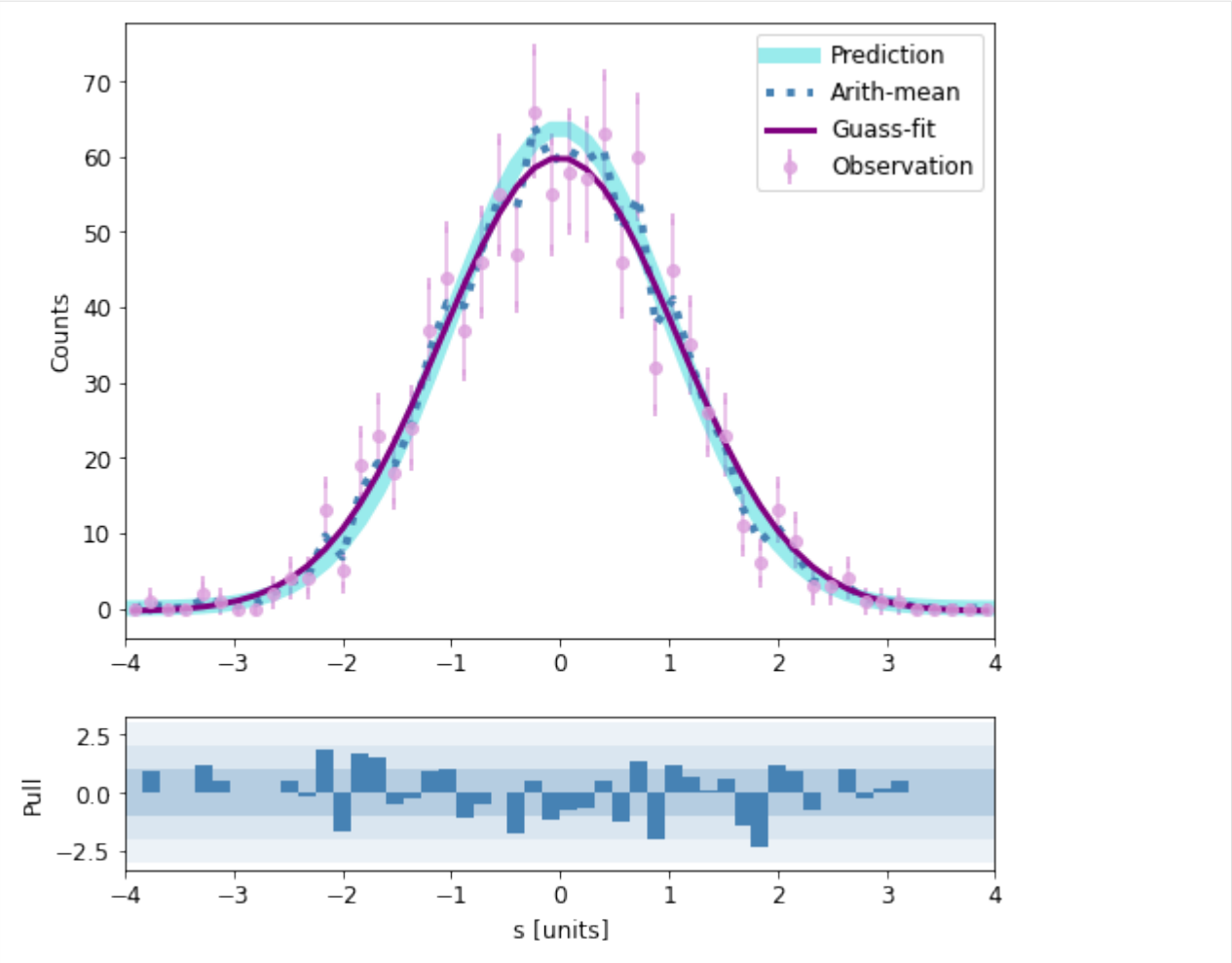

Prediction
Arith-mean
Guass-fit
Observation
Counts
70
60
50
40
30
20
10
0
−4
−3
−2
−1
0
1
2
3
4
Pull
2.5
0.0
−2.5
s [units]

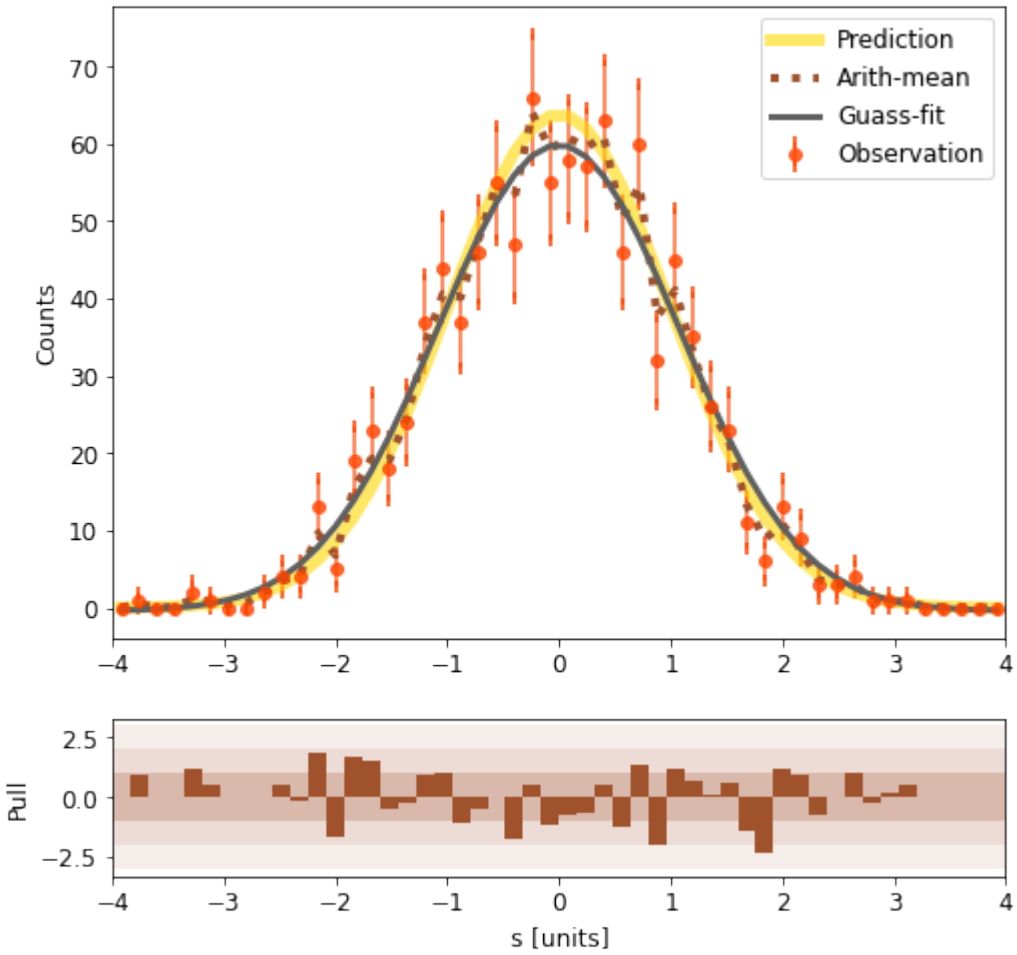

Prediction
Arith-mean
Guass-fit
Observation
Counts
70
60
50
40
30
20
10
0
−4
−3
−2
−1
0
1
2
3
4
Pull
2.5
0.0
−2.5
s [units]


Prediction
Arith-mean
Guass-fit
Observation
Counts
70
60
50
40
30
20
10
0
−4
−3
−2
−1
0
1
2
3
4
Pull
2.5
0.0
−2.5
s [units]


Prediction
Arith-mean
Guass-fit
Observation
Counts
70
60
50
40
30
20
10
0
−4
−3
−2
−1
0
1
2
3
4
Pull
2.5
0.0
−2.5
s [units]


Prediction
Arith-mean
Guass-fit
Observation
Counts
70
60
50
40
30
20
10
0
−4
−3
−2
−1
0
1
2
3
4
Pull
2.5
0.0
−2.5
s [units]


Prediction
Arith-mean
Guass-fit
Observation
Counts
70
60
50
40
30
20
10
0
−4
−3
−2
−1
0
1
2
3
4
Pull
2.5
0.0
−2.5
s [units]



2.3.3 Pull plot pro

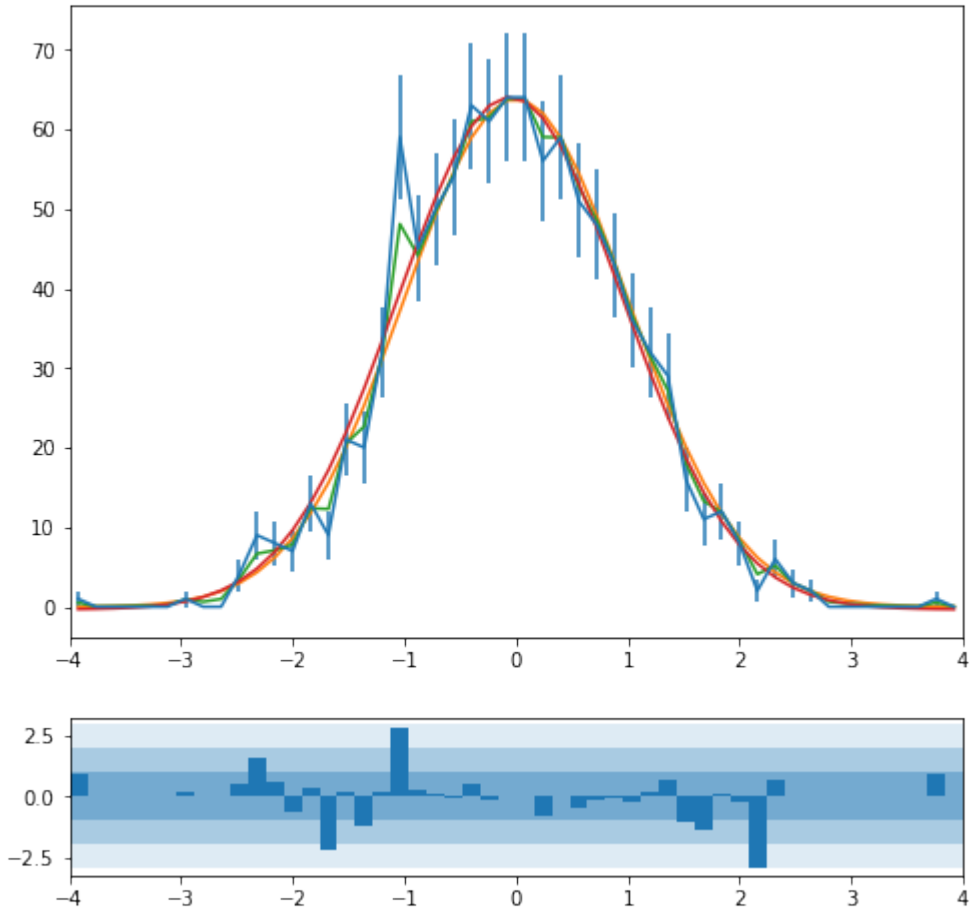
We also provide a more customized pull plot method – pull plot pro. Pull plot pro has offer more choices for users, such as setting color, alpha, markersize… Let's see what's new!

[8]:
```
import hist
import numpy as np
import matplotlib.pyplot as plt
```

[9]:
```
def pdf(x, a=1/np.sqrt(2*np.pi), x0=0, sigma=1, offset=0):
    return a * np.exp(-(x-x0)**2/(2*sigma**2)) + offset
```

[10]:
```
h = hist.Hist(
    hist.axis.Regular(50, -4, 4, name="S", title="s [units]", underflow=False,␣
↪overflow=False)
)

data = np.random.normal(size=1_000)
h.fill(data)

h.pull_plot_pro(pdf)
```

[10]:
```
(<Figure size 576x576 with 2 Axes>,
 <matplotlib.axes._subplots.AxesSubplot at 0x12faa1590>,
 <matplotlib.axes._subplots.AxesSubplot at 0x12fd7c390>)
```



The feature of auto-construting figure and axes is still support, with 8*8 figure size. Themes are no longer supported, if you do not specify the params, the plotting will be in the default Matplotlib styles.

We specify some params for pull plot pro. Just like normal plotting params in Matplotlib, what you need to do is just to add axis names' abbrevations to normal Matplotlib params keywords:

- errorbar param -> `eb_param`
- valueplot param -> `vp_param`
- meanplot param -> `mp_param`
- fitplot param -> `fp_param`
- barplot param -> `bar_param`
- patchplot param -> `pp_param`

Let's see a more complicated example. In this example, we change the style of pull plot pro to the one like 'Autumn' and set a beautiful gradient effect to the patch plot.

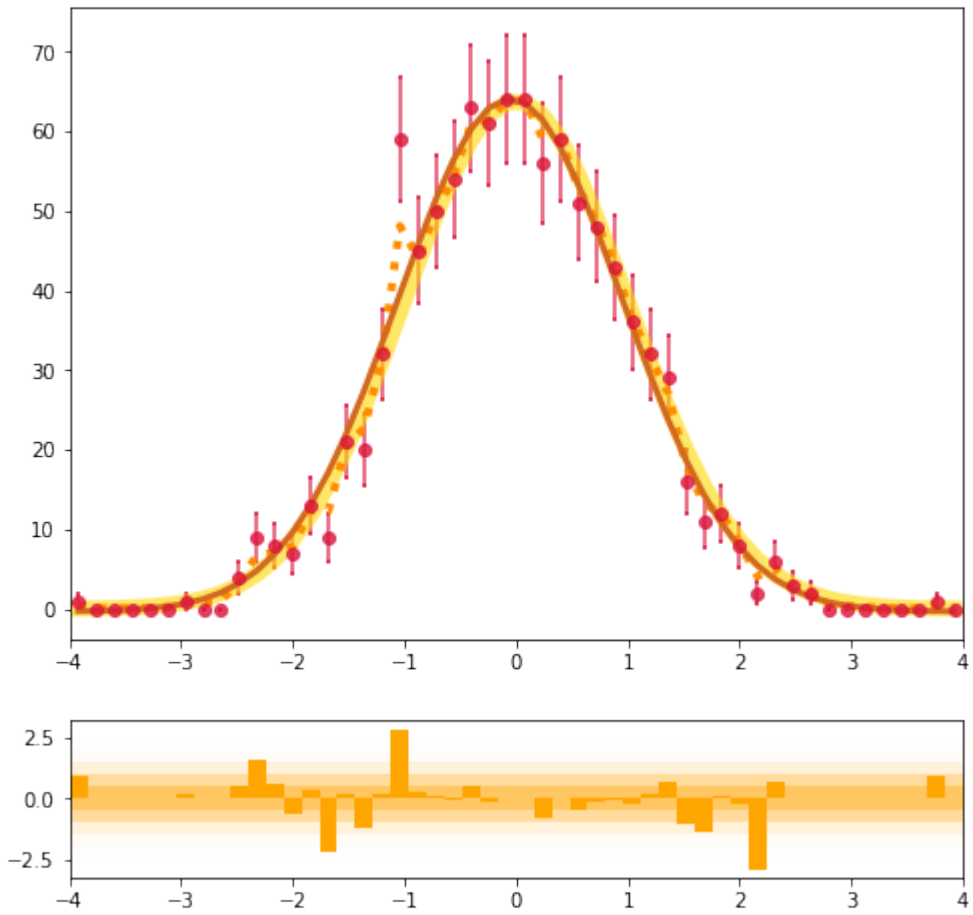
[11]:
```
h.pull_plot_pro(pdf, eb_ecolor='crimson', eb_mfc='crimson', eb_mec='crimson', eb_fmt=
↪'o', eb_ms=6,\
```

(continues on next page)

(continued from previous page)

```
                eb_capsize=1, eb_capthick=2, eb_alpha=.8, vp_c='gold', vp_ls='-', vp_
↪lw=8,\
                vp_alpha=.6, mp_c='darkorange', mp_ls=':', mp_lw=4, mp_alpha=1.,\
                fp_c='chocolate', fp_ls='-', fp_lw=3, fp_alpha=1., bar_fc='orange',\
                pp_num=6, pp_fc='orange', pp_alpha=.618, pp_ec=None)
```

[11]:
```
(<Figure size 576x576 with 2 Axes>,
 <matplotlib.axes._subplots.AxesSubplot at 0x12fec8810>,
 <matplotlib.axes._subplots.AxesSubplot at 0x12ffe34d0>)
```



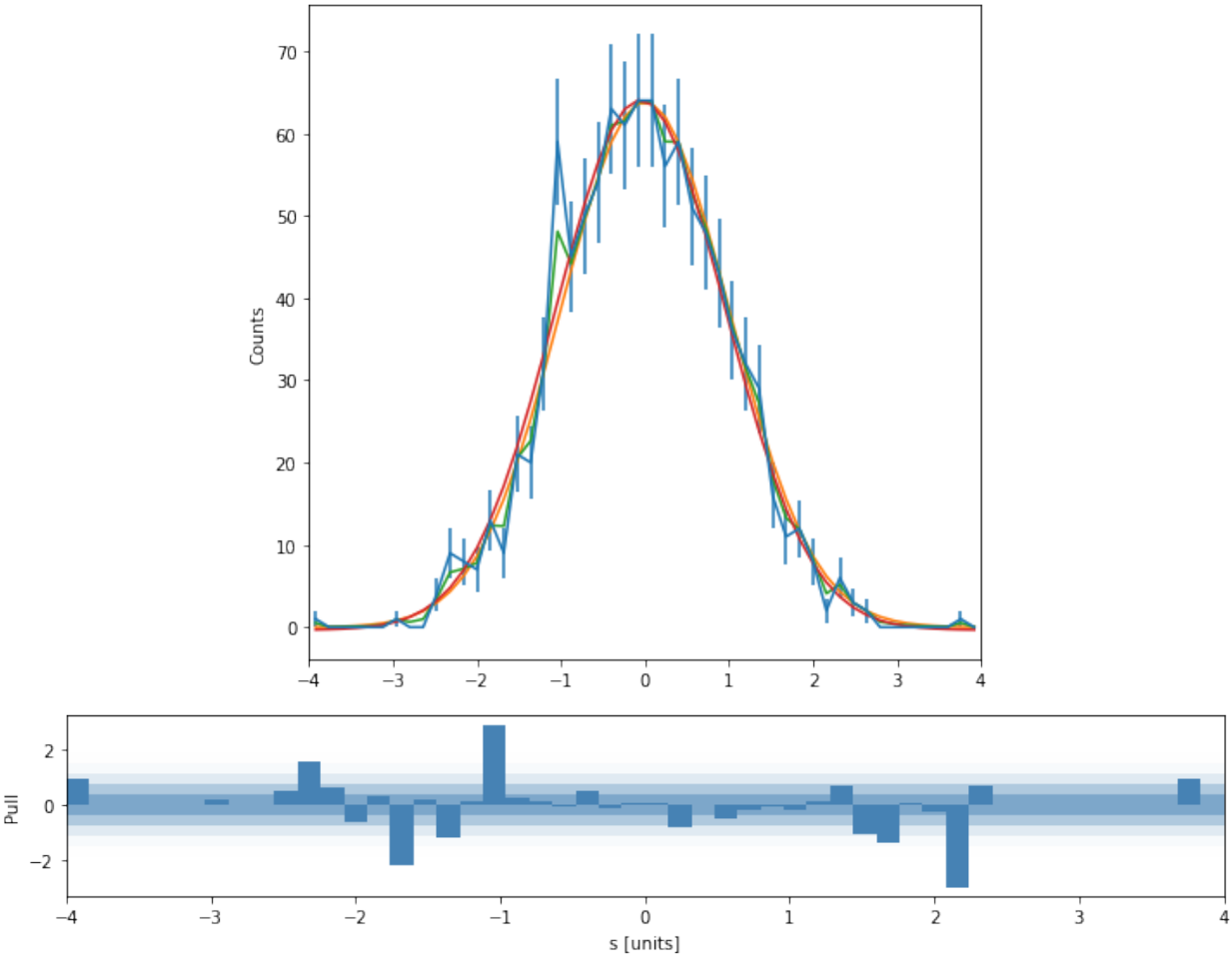
We recommend you to manipulate axes outside pull plot pro like this.

[12]:
```
fig = plt.figure(figsize=(12, 12))
grid = fig.add_gridspec(5, 5, wspace=0.3, hspace=0.3)
ax = fig.add_subplot(grid[0:3, 1:4])
pull_ax = fig.add_subplot(grid[3:4, :], sharex=ax)

fig, ax, pull_ax = h.pull_plot_pro(pdf, fig=fig, ax=ax, pull_ax=pull_ax, bar_fc=
↪'steelblue',\
                pp_fc='steelblue', pp_num=8, pp_alpha=.7)

ax.set_ylabel("Counts")
pull_ax.set_xlabel(h.axes[0].title)
pull_ax.set_ylabel("Pull")
```

(continues on next page)

(continued from previous page)

```
fig.savefig("img/ax-img-pro.png")
```



Pull plot provides themes for users, which make it a convenient tool to draw pull plot.

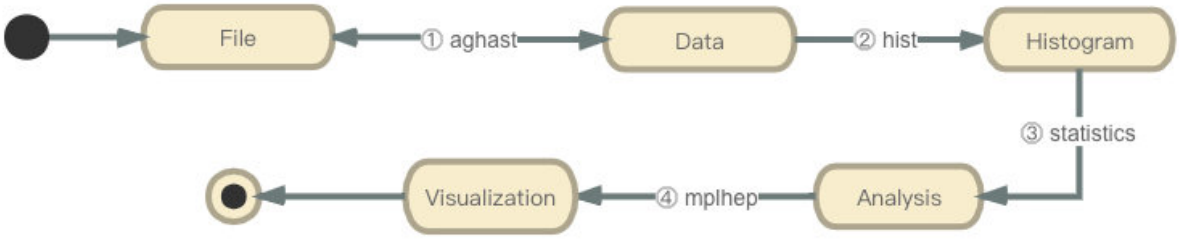
Have some tries to explore it by yourself!

2.4 External Integration

2.4.1 General

Nino-hist is a systematic tool based on boost-histogram. The HEP analysis is a complicated process and relies on many other (Sckit-HEP) tools. Thus, it's intuitive for Nino-hist to support the integration of these external packages and modules. In this part, we will demonstrate how Nino-hist works and its interaction with other (Sckit-HEP) tools.

A brief workflow of HEP analysis is as below:



2.4.2 De/serialization

If we want to analysis data, what do we need to do first? To load the data files, of course! How do we do it? It denpends on the file, of course!

2.4.2.1 pickle Integration

We could use pickle to deal with it like this:

[1]:
```
import hist
import pickle
from pathlib import Path

h1 = hist.Hist(hist.axis.Regular(2, -1, 1, name='h_1'))
h2 = h1.copy()
h2.name = 'h_2'

h1.fill(-0.5)
h2.fill(0.5)

# Arithmetic operators
h3 = h1 + h2
h2.name = 'h_3'
h4 = h3 * 2
h2.name = 'h_4'

print(f"{h4[0]}, {h4[1]}")

h4_saved = Path("pickle_file.pkl")

# Now save the histogram
with h4_saved.open("wb") as f:
    pickle.dump(h4, f, protocol=-1)

# And load
with h4_saved.open("rb") as f:
    h5 = pickle.load(f)

assert h4 == h5
print("Succeeded in pickling a histogram!")

# Delete the file to keep things tidy
# h4_saved.unlink()
```

```
2.0, 2.0
Succeeded in pickling a histogram!
```

2.4.2.2 aghast Integration

Though pickle is capable for serialization and deserialization, we sometimes need to use ROOT files for loading and saving data in HEP analysis. Initially, we expect to use aghast for Nino-hist to interactive with ROOT files. While it's not pure Pythonic and not installable on PyPI, currently. Thus, here we first demonstrate the availability of aghast and then show how to save and load our histograms by aghast's underlying uproot. aghast is a histogramming library that does not fill histograms and does not plot them. Its role is behind the scenes, to provide better communication between histogramming libraries.

As we need to make Nino-hist available to communicate with aghast, that is, to change aghast's histograms to aghast's histograms.

[2]:
```
import aghast
import numpy as np
import matplotlib.pyplot as plt

h = hist.Hist(hist.axis.Regular(50, -3, 3, name='x'))
h.fill(np.random.normal(size=1_000_000))
ghastly_hist = aghast.from_numpy(h.to_numpy())
ghastly_hist.dump()
```

```
Histogram(
  axis=[
    Axis(binning=RegularBinning(num=50, interval=RealInterval(low=-3.0, high=2.
↪9999999999999996)))
  ],
  counts=
    WeightedCounts(
      sumw=
        InterpretedInlineFloat64Buffer(
          buffer=
              [  633.   841.  1273.  1817.  2324.  3077.  4077.  5304.  6775.  8417.
               10582. 12991. 15554. 18490. 21941. 25120. 28839. 31609. 35316. 38458.
               41635. 43466. 45603. 46843. 47655. 47754. 47388. 45594. 43733. 41031.
               38711. 35442. 32205. 28463. 24793. 21814. 18531. 15621. 12792. 10405.
                8297.  6747.  5341.  4026.  3103.  2294.  1776.  1233.   893.   627.
↪]))))
```

See, it works! We are looking forward to aghast's update, and we can use the `ghastly_hist` for better use.

I am sure it will be a shortcut for file serialization and deserialization.

2.4.2.3 uproot Integration

Now that aghast need the more detailed development to support serialization and deserialization for our Nino-hist, we have to use uproot to demonstrate how can we save and load histograms in Nino-hist by ROOT files.

uproot is the most classical ROOT I/O in Python. It is a reader and a writer of the ROOT file format using only Python and Numpy. Unlike the standard C++ ROOT implementation, uproot is only an I/O library, primarily intended to stream data into machine learning libraries in Python. Unlike PyROOT and root_numpy, uproot does not depend on C++ ROOT. Instead, it uses Numpy to cast blocks of data from the ROOT file as Numpy arrays.

The first thing we need to do is to create Nino-hist's histograms as normal.

[3]:
```
import hist
import uproot
import numpy as np
import matplotlib.pyplot as plt

file = uproot.recreate("demo_root_file.root")
h = hist.Hist(hist.axis.Regular(20, -3, 3, name='x'))
h.fill(np.random.normal(size=1_000))

w, data = h.to_numpy()
```

Here, we use a little tricks. First, we convert the Nino-hist's histogram to a Numpy histogram. And then write the histogram as well as a title to the `demo_root_file.root`. We load the demo to another file, printing the title, showing the histogram.

[4]:
```
file['title'] = 'A histogram'
file['h'] = np.histogram(data[:-1], bins=data, weights=w)
file2 = uproot.open("demo_root_file.root")
print(file2['title'])
file2['h'].show()
```

```
A histogram
                 0                                                          134.4
                 +----------------------------------------------------------------+
[-inf, -3)    0  |                                                                |
[-3, -2.7)    3  |*                                                               |
[-2.7, -2.4)  6  |***                                                             |
[-2.4, -2.1)  7  |***                                                             |
[-2.1, -1.8)  19 |*********                                                       |
[-1.8, -1.5)  31 |**************                                                  |
[-1.5, -1.2)  51 |***********************                                         |
[-1.2, -0.9)  57 |************************                                        |
[-0.9, -0.6)  97 |****************************************                        |
[-0.6, -0.3)  104|*****************************************                       |
[-0.3, 0)     117|************************************************              |
[0, 0.3)      128|****************************************************          |
[0.3, 0.6)    108|*********************************************                   |
[0.6, 0.9)    101|******************************************                      |
[0.9, 1.2)    59 |************************                                        |
[1.2, 1.5)    48 |********************                                            |
[1.5, 1.8)    26 |***********                                                     |
[1.8, 2.1)    20 |*********                                                       |
[2.1, 2.4)    8  |****                                                            |
[2.4, 2.7)    4  |**                                                              |
[2.7, 3)      2  |*                                                               |
[3, inf]      0  |                                                                |
                 +----------------------------------------------------------------+
```

Note that uproot is a pretty large package which supports many functionalities. Here, we just demonstate how to save and load ROOT files. You can go to the origin to see more information.

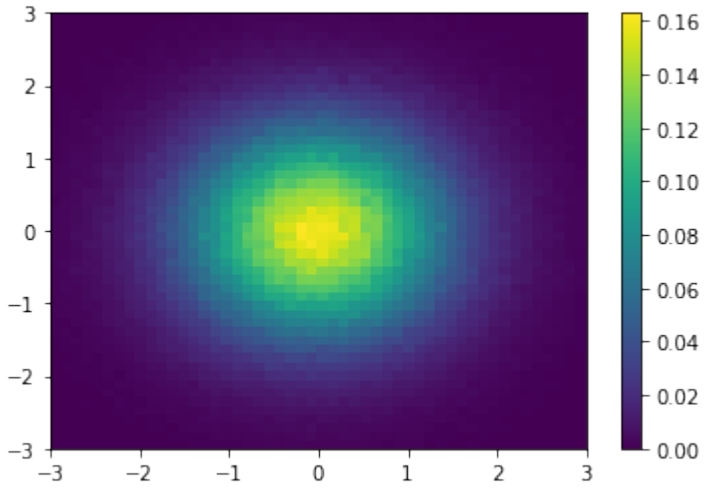
2.4.3 Histogram

2.4.3.1 Nino-hist

Nino-hist acts as the histogramming part, i.e., "Data" to "Histogram", in HEP analysis workflow. It is based on boost-histogram, meaning that you can do whatever you like in boost-histogram with histograms of Nino-hist. Let's see some examples.

First, we will re-use the 2D-density histogram example of boost-histogram.

[5]:
```
import hist
import numpy as np
import matplotlib.pyplot as plt

# Make a 2D histogram
h = hist.Hist(hist.axis.Regular(50, -3, 3, name='x'), hist.axis.Regular(50, -3, 3,␣
↪name='y'))

# Fill with Gaussian random values
h.fill(np.random.normal(size=1_000_000), np.random.normal(size=1_000_000))

# Compute the areas of each bin
areas = np.prod(h.axes.widths, axis=0)

# Compute the density
density = h.view() / h.sum() / areas

# Get the edges
X, Y = h.axes.edges

# Make the plot
fig, ax = plt.subplots()
mesh = ax.pcolormesh(X.T, Y.T, density.T)
fig.colorbar(mesh)
plt.show()
```



Nice! This is good, because this means that you can migrate your boost-histogram codes to Nino-hist directly.

Another example:

[6]:
```
import hist
import boost_histogram as bh

# Make 1-d histogram with 5 logarithmic bins from 1e0 to 1e5
h = hist.Hist(
    hist.axis.Regular(5, 1e0, 1e5, name="x", transform=bh.axis.transform.log),
    storage=bh.storage.Weight(),
)

# Fill histogram with numbers
x = (2e0, 2e1, 2e2, 2e3, 2e4)

# Doing this several times so the variance is more interesting
h.fill(x, weight=1)
h.fill(x, weight=1)
h.fill(x, weight=1)
h.fill(x, weight=1)

# Iterate over bins and access bin counter
for idx, (lower, upper) in enumerate(h.axes[0]):
    val = h[idx]
    print(val)
    print(f"bin {idx} in [{lower:g}, {upper:g}): {val.value} +/- {val.variance**.5}")
```

```
WeightedSum(value=4, variance=4)
bin 0 in [1, 10): 4.0 +/- 2.0
WeightedSum(value=4, variance=4)
bin 1 in [10, 100): 4.0 +/- 2.0
WeightedSum(value=4, variance=4)
bin 2 in [100, 1000): 4.0 +/- 2.0
WeightedSum(value=4, variance=4)
bin 3 in [1000, 10000): 4.0 +/- 2.0
WeightedSum(value=4, variance=4)
bin 4 in [10000, 100000): 4.0 +/- 2.0
```

We haven't define `Storage` and `transform` for Nino-hist yet, so you cannot directly use them in our package, but Nino-hist's compatibility with boost-histogram is good, so you can use whatever you need by citing boost-histogram's corresponding modules.

To sum up, Nino-hist acts as the core part in HEP analysis workflow, allowing us to histogram data. This is valuable for "Data" to "Histogram" in HEP analyse workflow!

2.4.4 Analysis

Here we are going to show how some (Sckit-HEP) analysis tools are integrated to our package.

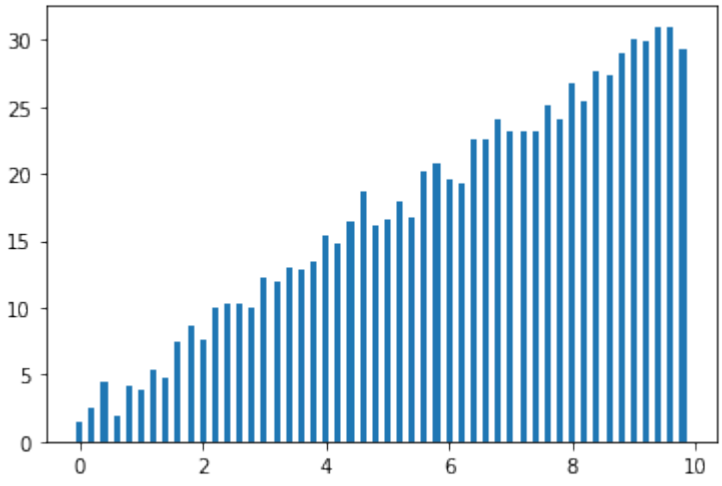
2.4.4.1 iminuit Integration

iminuit is most commonly used for likelihood fits of models to data, and to get model parameter error estimates from likelihood profile analysis. It's a numerical minimizer and error calculator. You provide it an analytical function, which accepts one or several parameters, and an initial guess of the parameter values. It will then find a local minimum of this function starting from the initial guess. In that regard, iminuit is similar to other minimizers, like those in `scipy.optimize`.

Let's use an example to see how can iminuit fits the data in Nino-hist.

[7]:
```
import hist
import numpy as np
import matplotlib.pyplot as plt

k = 3.
b = 2.
data_x = np.arange(0, 10, .2)
offset = np.random.normal(size=50)
data_y = k * data_x + b + offset

plt.bar(data_x, data_y, width=0.1)

h = hist.Hist(
    hist.axis.Regular(20, 0, 10, name='iminuit', underflow=False, overflow=False)
)

h.fill(data_x, weight=data_y)
```

[7]:
```
Hist(Regular(20, 0, 10, underflow=False, overflow=False, metadata={'name': 'iminuit',
↪'title': None}), storage=Double()) # Sum: 844.3786162013363
```



By this way, we create and fill a histogram with the linear distribution, i.e., the weight of each bin is the multiple of its index. Of course you can also use a histogram loading by serialization. Back to the point, it's a linear distribution, and we want to use a fitter to get its slope `k` and intercept `b`.

First, we need to access the data (the weight of histogram) and define the error func to be minimized.

[8]:
```
y, x = h.to_numpy()

def least_squares_np(par):  # par is a numpy array here
    mu = np.polyval(par, data_x)  # for par = (a, b) this is a line
```

(continues on next page)

(continued from previous page)

```
    yvar = 0.01
    return np.sum((data_y - mu) ** 2 / yvar)
```

Then, we will initialized the parameters in an array (two elements in this linear case), the step size, and the error definition (`errordef = 0.5` for negative log-likelihood functions, `errordef = 1` for least-squares functions). You are able to see the minuit initial states use `get_param_states()`.

[9]:
```
from iminuit import Minuit

m = Minuit.from_array_func(least_squares_np, (5, 5), error=(0.1, 0.1), errordef=1,␣
↪name=('k', 'b'))
m.get_param_states()
```

[9]:
```
------------------------------------------------------------------------------------
↪----
|   | Name |   Value   | Hesse Err | Minos Err- | Minos Err+ | Limit-  | Limit+  |␣
↪Fixed |
------------------------------------------------------------------------------------
↪----
| 0 | k    |   5.00    |   0.10    |            |            |         |         |   ␣
↪   |
| 1 | b    |   5.00    |   0.10    |            |            |         |         |   ␣
↪   |
------------------------------------------------------------------------------------
↪----
```

And it's time for fitting! It's pretty handy in iminuit, just `m.migrad()` is ok. Of course you can use more specific methods like fixed, limits, fast fitting methods, etc. This is just the simplest way to fit.

[10]:
```
m.migrad()
# m.get_param_states()
```

[10]:
```
------------------------------------------------------------------
| FCN = 5340                    |      Ncalls=34 (34 total)      |
| EDM = 1.47E-19 (Goal: 1E-05)  |            up = 1.0            |
------------------------------------------------------------------
|  Valid Min.   | Valid Param.  | Above EDM | Reached call limit |
------------------------------------------------------------------
|     True      |     True      |   False   |       False        |
------------------------------------------------------------------
| Hesse failed  |   Has cov.    | Accurate  | Pos. def. | Forced |
------------------------------------------------------------------
|     False     |     True      |   True    |   True    | False  |
------------------------------------------------------------------
------------------------------------------------------------------------------------
↪----
|   | Name |   Value   | Hesse Err | Minos Err- | Minos Err+ | Limit-  | Limit+  |␣
↪Fixed |
------------------------------------------------------------------------------------
↪----
| 0 | k    |   3.000   |   0.005   |            |            |         |         |   ␣
↪   |
| 1 | b    |   2.189   |   0.028   |            |            |         |         |   ␣
↪   |
------------------------------------------------------------------------------------
↪----
```

Nice! It's almost true! Next, we want to visualized our fitting (the correlationship of two params and the params' scopes).

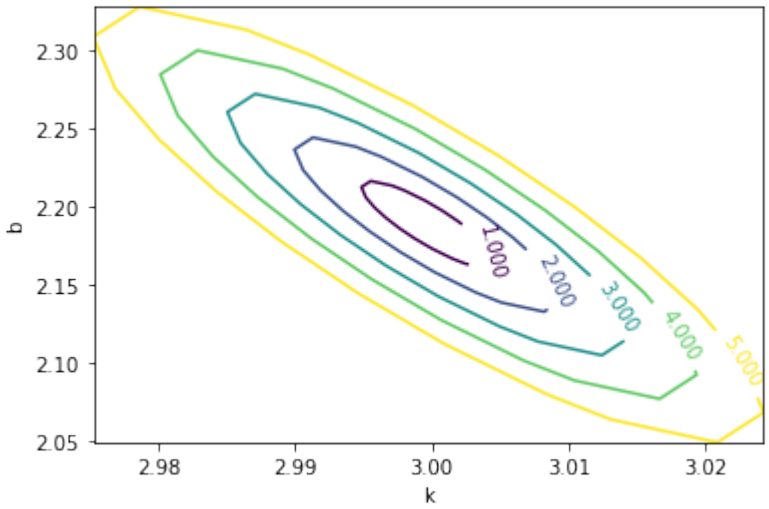
We can analysis the correlationship of the params to eliminate some of them.

[11]:
```
m.matrix(correlation=True)
```

[11]:
```
------------------
|   |    k     b |
------------------
| k |  1.0 -0.9 |
| b | -0.9  1.0 |
------------------
```

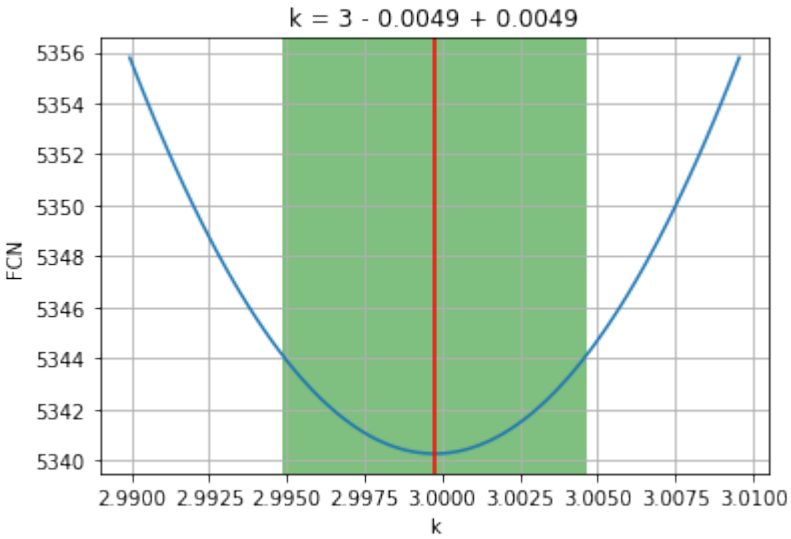
We can also use contour to represent it.

[12]:
```
m.draw_mncontour('k','b', nsigma=5);  # draw five contours from sigma=1 to 5
# m.draw_contour('k','b');
```



We can also view the scope of those params.

[13]:
```
m.draw_profile('k');
# m.draw_profile('b');
```



To sum up, Nino-hist allow user to fit data by interacting with iminuit. Sepcifically, you need to convert your histogram to numpy format, and then iminuit will deal with that. With numpy arrays as the media, it's flexible. You can do whatever you want in iminuit with the histograms of Nino-hist. This is valuable for "Histogram" to "Analysis" in HEP analyse workflow!

2.4.5 Visualization

You can use many tool for Nino-hist's visualization, because it can be converted to numpy arrays and there are many visualization tools for numpy arrays I firmly believe. But there, our primary focus is its interation with mplhep.

2.4.5.1 mplhep Integration

Although we can use Matplotlib to visualize our histograms, it should be a better way to do the same thing in a, umm… (Sckit-HEP) manner. mplhep is a set of helpers for matplotlib to more easily produce plots typically needed in HEP as well as style them in way that's compatible with current collaboration requirements (ROOT).

Let's use an example to see how mplhep visualizes the histograms in Nino-hist.

First, create 1-D and 2-D histograms.

[14]:
```
import hist
import numpy as np
import matplotlib.pyplot as plt

data_x = np.random.normal(loc=0., scale=1., size=1_000_000)
data_y = np.random.normal(loc=0., scale=1., size=1_000_000)

h_1d = hist.Hist(
    hist.axis.Regular(500, -3, 3, name='x_1d', underflow=False, overflow=False)
)

h_1d.fill(data_x)

h_2d = hist.Hist(
    hist.axis.Regular(500, -3, 3, name='x_2d', underflow=False, overflow=False),
    hist.axis.Regular(500, -3, 3, name='y_2d', underflow=False, overflow=False)
)

h_2d.fill(data_x, data_y)
```

[14]:
```
Hist(
  Regular(500, -3, 3, underflow=False, overflow=False, metadata={'name': 'x_2d',
↪'title': None}),
  Regular(500, -3, 3, underflow=False, overflow=False, metadata={'name': 'y_2d',
↪'title': None}),
  storage=Double()) # Sum: 994660.0
```

Then, we can plot our histogram. (mplhep is in its infancy, with a few very single features, but I'm sure there will be better interfaces later.)

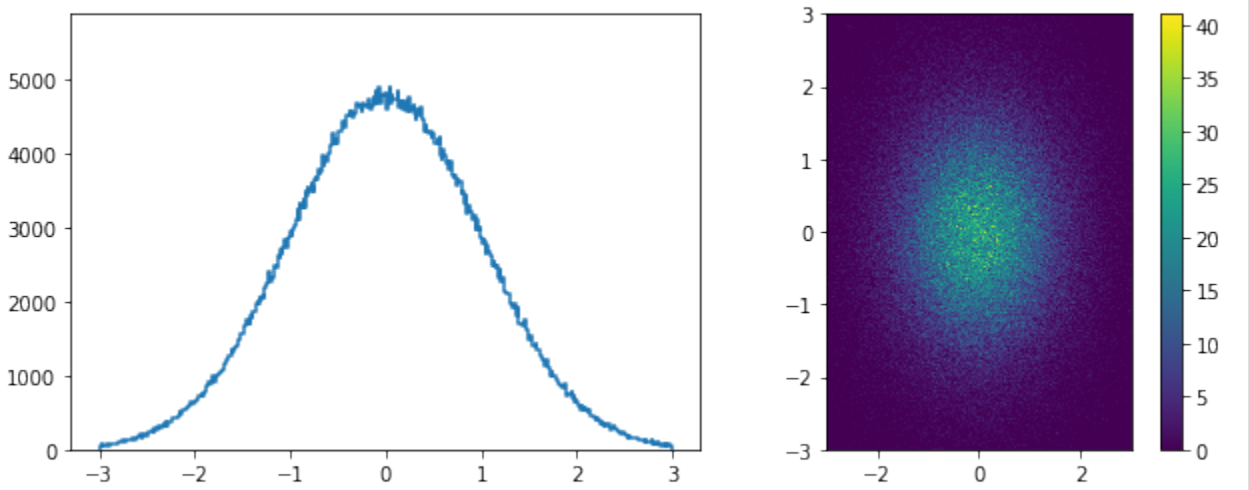
[15]:
```
import mplhep

# plt.style.use(mplhep.style.LHCb)
plt.subplot(121)
w, xbins = h_1d.to_numpy()
mplhep.histplot(w.T, xbins)
```

(continues on next page)

(continued from previous page)

```
plt.subplot(122)
w, xbins, ybins = h_2d.to_numpy()
mplhep.hist2dplot(w.T, xbins, ybins)
```

[15]: `<matplotlib.axes._subplots.AxesSubplot at 0x12d2a5790>`



To sum up, Nino-hist allow user to visualize data by interacting with mplhep. Sepcifically, you need to convert your histogram to numpy format, and then mplhep will deal with that. With numpy arrays as the media, it's flexible. You can do whatever you want in mplhep with the histograms of Nino-hist. This is valuable for "Analysis" to "Visualization" in HEP analyse workflow!

CHAPTER

THREE

EXAMPLES

3.1 Axes

3.1.1 Axes Name

```
import hist
import numpy as np
import matplotlib.pyplot as plt

h = hist.NamedHist(
    hist.axis.Regular(10, 0, 1, name='myRegular'),
    hist.axis.Integer(-1, 1, name='myInteger')
)

regular = [.15, .15, .25, .35, .55, .55]
integer = [-1, -1, 0, 0, 0, 0]

h.fill(myRegular=regular, myInteger=integer)

fig, ax = plt.subplots(figsize=(8,5))
w, x, y = h.to_numpy()
mesh = ax.pcolormesh(x, y, w.T)
ax.set_xlabel(h.axes[0].metadata["name"])
ax.set_ylabel(h.axes[1].metadata["name"])
fig.colorbar(mesh)
fig.show()
```

3.1.2 Axes Title

```
import hist
import numpy as np
import matplotlib.pyplot as plt

h = hist.NamedHist(
    hist.axis.Regular(10, 0, 1, name='myRegular', title='Regular'),
    hist.axis.Integer(-1, 1, name='myInteger', title='Regular')
)

regular = [.15, .15, .25, .35, .55, .55]
integer = [-1, -1, 0, 0, 0, 0]
```

(continues on next page)

(continued from previous page)

```
h.fill(myRegular=regular, myInteger=integer)

fig, ax = plt.subplots(figsize=(8,5))
w, x, y = h.to_numpy()
mesh = ax.pcolormesh(x, y, w.T)
ax.set_xlabel(h.axes[0].title)
ax.set_ylabel(h.axes[1].title)
fig.colorbar(mesh)
fig.show()
```

3.1.3 Bool Axes

Use boost_histogram imitate `Bool` axes:

```
import boost_histogram as bh
import numpy as np
import matplotlib.pyplot as plt

h = bh.Histogram(
    bh.axis.Regular(50, -1, 1),
    bh.axis.Regular(50, -1, 1),
    bh.axis.Integer(0, 2, underflow=False, overflow=False),
)

x, y = np.random.random_sample([2, 1_000_000])*2 - 1
valid = (x**2 + y**2) < .5

h.fill(x, y, valid)
valid_only = h[:, :, bh.loc(True)]

fig, ax = plt.subplots(figsize=(8,5))
W, X, Y = valid_only.to_numpy()
mesh = ax.pcolormesh(X, Y, W.T)
ax.set_xlabel('x')
ax.set_ylabel('y')
fig.colorbar(mesh)
fig.show()
```

Nino-hist `Bool` axes:

```
import boost_histogram as bh
import hist
import numpy as np
import matplotlib.pyplot as plt


h = hist.NamedHist(
    hist.axis.Regular(50, -1, 1, name="X"),
    hist.axis.Regular(50, -1, 1, name="Y"),
    hist.axis.Bool(name="V"),
)

x, y = np.random.random_sample([2, 1_000_000])*2 - 1
valid = (x**2 + y**2) < .5
h.fill(Y=y, X=x, V=valid)
```

(continues on next page)

(continued from previous page)

```
valid_only = h[:, :, bh.loc(True)]

fig, ax = plt.subplots(figsize=(8,5))
W, X, Y = valid_only.to_numpy()
mesh = ax.pcolormesh(X, Y, W.T, cmap='spring')
ax.set_xlabel('X')
ax.set_ylabel('Y')
fig.colorbar(mesh)
fig.show()
```

3.2 NamedHist

3.2.1 NamedHist Simple

```
import boost_histogram as bh
import hist
import numpy as np
import matplotlib.pyplot as plt


h = hist.NamedHist(
    hist.axis.Regular(50, -3, 3, name="x"),
    hist.axis.Regular(50, -3, 3, name="y"),
)

x = np.random.randn(1_000_000)
y = np.random.randn(1_000_000)
h.fill(y=y, x=x)

fig, ax = plt.subplots(figsize=(8,5))
w, x, y = h.to_numpy()
mesh = ax.pcolormesh(x, y, w.T, cmap='autumn')
ax.set_xlabel('x')
ax.set_ylabel('y')
fig.colorbar(mesh)
fig.show()
```

3.2.2 NamedHist Coin Games

```
import hist
import numpy as np
import matplotlib.pyplot as plt

h = hist.NamedHist(
    hist.axis.Bool(name='USD', title='usd font'),
    hist.axis.Bool(name='EUR', title='eur font')
)
usd = np.random.rand(1_000_000) > 0.5
eur = np.random.rand(1_000_000) > 0.5
```

(continues on next page)

(continued from previous page)

```
h.fill(USD=usd, EUR=eur)

bar = [h[0, 0], h[0, 1], h[1, 0], h[1, 1]]
x = range(len(bar))
bar_color = ['steelblue', 'indianred', 'violet', 'orange']
fig, ax = plt.subplots(figsize=(8,5))
ax.bar(x, bar, color=bar_color)
plt.xticks(x, ("usd+, eur+", "usd+, eur-", "usd-, eur+",\
                                "usd-, eur-"), size=16)
plt.ylabel("Counts", size=16)


fig.show()
```

3.2.3 NamedHist Descartes

```
import boost_histogram as bh
import hist
import numpy as np
import matplotlib.pyplot as plt


h = hist.NamedHist(
    hist.axis.Regular(100, -1, 1, name="X"),
    hist.axis.Regular(100, -1, 1, name="Y"),
    hist.axis.Bool(name="V"),
)

x, y = np.random.random_sample([2, 1_000_000])*2 - 1
valid = np.abs(x)**2 + (y + .2 - np.power(np.abs(x), 2/3))**2 < .5
h.fill(Y=y, X=x, V=valid)

valid_only = h[:, :, bh.loc(True)]

fig, ax = plt.subplots(figsize=(8,5))
W, X, Y = valid_only.to_numpy()
mesh = ax.pcolormesh(X, Y, W.T, cmap='autumn')
ax.set_xlabel('X')
ax.set_ylabel('Y')
fig.colorbar(mesh)
fig.show()
```

3.3 Hist

3.3.1 Hist Simple

```
import hist
import numpy as np
import matplotlib.pyplot as plt


```

(continues on next page)

(continued from previous page)

```
def pdf(x, a=1/np.sqrt(2*np.pi), x0=0, sigma=1, offset=0):
    return a * np.exp(-(x-x0)**2/(2*sigma**2)) + offset


h = hist.Hist(
    hist.axis.Regular(50, -4, 4, name="S", title="s [units]", underflow=False,␣
↪overflow=False)
)

data = np.random.normal(size=1_000)

h.fill(data)

fig, ax, pull_ax = h.pull_plot(pdf, size='m', theme='Chrome')
```

3.3.2 Hist Customized Figure

```
def pdf(x, a=1/np.sqrt(2*np.pi), x0=0, sigma=1, offset=0):
    return a * np.exp(-(x-x0)**2/(2*sigma**2)) + offset


h = hist.Hist(
    hist.axis.Regular(50, -4, 4, name="S", title="s [units]", underflow=False,␣
↪overflow=False)
)

data = np.random.normal(size=1_000)

h.fill(data)

fig = plt.figure(figsize=(10, 10))
grid = fig.add_gridspec(5, 5, wspace=0.3, hspace=0.3)
ax = fig.add_subplot(grid[0:3, 2:])
pull_ax = fig.add_subplot(grid[3:4, :], sharex=ax)

h.pull_plot(pdf, size="m", fig=fig, ax=ax, pull_ax=pull_ax, theme='winter')
fig.savefig("img/ax-img.png")
```

3.3.3 Hist Pro

```
import hist
import numpy as np
import matplotlib.pyplot as plt

def pdf(x, a=1/np.sqrt(2*np.pi), x0=0, sigma=1, offset=0):
    return a * np.exp(-(x-x0)**2/(2*sigma**2)) + offset

h = hist.Hist(
    hist.axis.Regular(50, -4, 4, name="S", title="s [units]", underflow=False,␣
↪overflow=False)
)
```

(continues on next page)

(continued from previous page)

```
data = np.random.normal(size=1_000)
h.fill(data)

h.pull_plot_pro(pdf, eb_ecolor='crimson', eb_mfc='crimson', eb_mec='crimson', eb_fmt=
↪'o', eb_ms=6,\
                eb_capsize=1, eb_capthick=2, eb_alpha=.8, vp_c='gold', vp_ls='-', vp_
↪lw=8,\
                vp_alpha=.6, mp_c='darkorange', mp_ls=':', mp_lw=4, mp_alpha=1.,\
                fp_c='chocolate', fp_ls='-', fp_lw=3, fp_alpha=1., bar_fc='orange',\
                pp_num=6, pp_fc='orange', pp_alpha=.618, pp_ec=None)
```

3.3.4 Hist Customized Figure Pro

```
import hist
import numpy as np
import matplotlib.pyplot as plt

def pdf(x, a=1/np.sqrt(2*np.pi), x0=0, sigma=1, offset=0):
    return a * np.exp(-(x-x0)**2/(2*sigma**2)) + offset

h = hist.Hist(
    hist.axis.Regular(50, -4, 4, name="S", title="s [units]", underflow=False,␣
↪overflow=False)
)

data = np.random.normal(size=1_000)
h.fill(data)

fig = plt.figure(figsize=(12, 12))
grid = fig.add_gridspec(5, 5, wspace=0.3, hspace=0.3)
ax = fig.add_subplot(grid[0:3, 1:4])
pull_ax = fig.add_subplot(grid[3:4, :], sharex=ax)

fig, ax, pull_ax = h.pull_plot_pro(pdf, fig=fig, ax=ax, pull_ax=pull_ax, bar_fc=
↪'steelblue',\
                pp_fc='steelblue', pp_num=8, pp_alpha=.7)

ax.set_ylabel("Counts")
pull_ax.set_xlabel(h.axes[0].title)
pull_ax.set_ylabel("Pull")

fig.savefig("img/ax-img-pro.png")
```

CHAPTER

FOUR

DEVELOPMENT

4.1 Development Guideline

Nino-hist development is done in a git repository on github. Continuous integration testing is provided by travis CI, code coverage is measured with coveralls.io.

4.2 Repository structure

Nino-hist's repository structure is very simple, as we are normally not supporting older versions with bugfixes or other complicated things. The *master* branch of the repository is the basis for releases, a release is nothing more than adding a tag to the branch, creating the tarball, etc. The *master* branch should always be in a deployable state, i.e. one should be able to use it as the base for everyday work without worrying about random breakages due to updates. To ensure this, no commit ever goes into the *master* branch without passing the test suite before (see below). The only exception to this rule is if a commit not touches any code files, e.g. additions to the README file or to the documentation (but even in this case, care should be taken that the documentation is still built correctly).

For every feature that a developer works on, a new branch should be opened (normally based on the *master* branch), with a descriptive name (e.g. `add-matplotlib-support`). Developers should fork the repository and work in their own repository (if working on multiple issues/features, also using branches).

4.3 Implementing a feature/fixing a bug

Every new feature or bug fix should be done in a dedicated branch and have an issue in the issue database. For bugs, it is important to not only fix the bug but also to introduce a new test case that makes sure that the bug will not ever be reintroduced by other changes. It is often a good idea to first define the test cases (that should fail) and then work on the fix so that the tests pass. As soon as the feature/fix is complete *or* as soon as specific feedback on the code is needed, open a "pull request" to merge the changes from your branch into *master*. In this pull request, others can comment on the code and make suggestions for improvements. New commits to the respective branch automatically appear in the pull request which makes it a great tool for iterative code review. Even more useful, travis will automatically run the test suite on the result of the merge. As a reviewer, always wait for the result of this test (it can take up to 30 minutes or so until it appears) before doing the merge and never merge when a test fails. As soon as the reviewer decides that the branch is ready to merge, he/she can merge the pull request and optionally delete the corresponding branch (but it will be hidden by default, anyway).

4.4 Idea list

We have some preliminary ideas about what is possible or doable. Some concentrations are listed as follows.

4.4.1 Pull-plot pro implementation

- **Project description:** Release a more professional pull-plot version (i.e., pull-plot-pro) for Hist's pull-plot method
- **Objective:**
  - Allow user to customize pull-plot
  - Enable user to select type of plots in the pull-plot
  - Support passing more specific parameters
  - Specify the same properties for different plots, e.g. color, linewidth, linestyle, etc.
- **Possible tools:** Matplotlib.Axes, Matplotlib.Figure
- **Difficulty:** Easy

4.4.2 GUI design

- **Project description:** Design a graphic user interface for Nino-hist to simplify users' operations
- **Objective:**
  - Create GUI playground
  - Support Nino-hist's functionality
- **Possible tools:** Tkinter, wxPython, Jython
- **Difficulty:** Hard

4.4.3 Bool axis transformation

- **Project description:** Expand transformation to Bool axis
- **Objective:**
  - All transformation operations for Bool axis
  - Allow functional transformation by Callable
- **Possible tools:** Boost_histogram
- **Difficulty:** Medium

CHAPTER

FIVE

SUPPORT

If you are stuck with a problem using Nino-hist, please do get in touch at our issues.

You can save time by following this procedure when reporting a problem:

1. Do try to solve the problem on your own first. Read the documentation, including using the search feature, index and reference documentation.
2. Search the issue archives to see if someone else already had the same problem.
3. Before writing, try to create a minimal example that reproduces the problem. You'll get the fastest response if you can send just a handful of lines of code that show what isn't working.

CHAPTER

SIX

NINO-HIST API

6.1 Nino-hist package

6.1.1 Submodules

6.1.1.1 hist.axis module

6.1.1.2 hist.core module

class hist.core.**BaseHist**(**args*, ***kwargs*)
Bases: boost_histogram.Histogram

6.1.1.3 hist.general module

6.1.1.4 hist.named module

6.1.1.5 hist.theme module

class hist.theme.**Theme**(*theme*)
Bases: object

to_param(*size*)
Change theme to params.

6.1.1.6 hist.version module

6.1.2 Module contents

PYTHON MODULE INDEX

h

INDEX