haven

Release 0.1.0

Mark Wolfman

Apr 03, 2024

CONTENTS:

Warning: This package is intended for use primarily by beamline staff. For user operations at the beamline, consider the *firefly* package.

CHAPTER

ONE

TUTORIALS

These tutorials provide a step-by-step guide to performing specific tasks.

These guides do not cover the individual beamlines thoroughly.

This file is also available as an interactive jupyter notebook.

Download this file as a notebook (hint: right-click -> "save as").

1.1 Demonstration of Knife Scans

This notebook shows the following tasks: - basic of collecting a knife edge scan - loading the scan data from the database - analyzing the knife edge scan to determine beam position and size - saving the data to disk as a csv file

First we have to import haven, the beamline control library. Haven contains most of the tools we will use. Importing it allows us to get to the functions and classes that are defined inside.

Next, the `haven.load_instrument()` function will read the configuration files and scan the hardware for its configuration. It will then **build the devices** that will be used for scans. This function prints out a list of motors that it has discovered.

Then, we create the run engine. The run engine is responsible for executing our scans and will be described in more detail when it is used below.

Lastly we set metadata about who is running the beamline. This value will be saved in every plan executing on this run engine. This step is optional, but will allow database queries for scans taken by a specific person.

[2]:
```
import haven
# Load the motors and detectors
haven.load_instrument()
# The RunEngine will be responsible for executing the plans
RE = haven.RunEngine()
# (Optional) Save the initials of the current beamline operator as metadata
RE.md["operator"] = "MFW"  # <- Put your initials in here
```

```
energy_energy
energy_mono_energy
I0_sensitivity_sens_level
Idk_sensitivity_sens_level
It_sensitivity_sens_level
SLT V Upper
SLT V Lower
SLT H Inb
SLT H Outb
```

(continues on next page)

(continued from previous page)

```
KBH Down
KBH Up
KBV Down
KBV Up
KB Tbl H
KB Tbl V
Focus V
Focus H
Focus In Out
ADC KB Focus
SampleZ
I0 H
I0 V
Optic In Out
Optic V
Optic H
```

```
cannot connect to 25idcVME:m65.DESC
```

```
monochromator_horiz
monochromator_vert
monochromator_bragg
monochromator_gap
monochromator_roll2
monochromator_pitch2
monochromator_roll_int
monochromator_pi_int
monochromator_mode
monochromator_energy
Aerotech_vert
Aerotech_horiz
```

1.1.1 Running the Scan

Running a scan in bluesky is a **two step process**.

First, **create a plan**. A plan generates messages with instructions to do things like move a motor, wait for a motor to arrive at its destination, and trigger and read a detector. To create a plan, you call a function that will generate these messages. **Calling the function doesn't actually execute the scan.** In our case, `haven.knife_scan("Focus V", -200, 200, 50)` will create the plan, but the plan will not do anything unless used with a run engine.

`haven.knife_scan` needs to know which motor to scan over, so in the cell below we provide it with `knife_motor="Focus V"`. The names of motors are the same as those printed out when loading the instrument configuration above. The *start*, *stop* parameters determine the range that the motor will scan. If *relative* is False (default), then *start* and *stop* will be absolute positions, and if *relative* is True, then *start* and *stop* will be relative to motor's position at the start of the run.

The knife scan plan also needs to know **which detectors to measure**. The names for the detectors are the same as the descriptions for the EPICS channels on the scaler. By default it will use `It="It"` and `I0="I0"`, but any scaler channel can be used.

Next, **execute the plan on the run engine**. At the top of this document we created a run engine. Now we will use this run engine to execute the plan. The run engine will read the messages and perform the appropriate tasks.

When the run engine finishes the plan, it will return a unique identifier (UID). This UID is the best way to retrieve the data from the database. We will save the UID to a variable, and also print it to the page in case we want to recall it later.

[ ]:
```
plan = haven.knife_scan(knife_motor="Focus V", start=-200, end=200, num=50,␣
↪relative=True)
uid = RE(plan)
print(uid)
```

1.1.2 Loading the Data

During execution the data are saved to a mongoDB database. Haven has tools to retrieve the data, as well as fit the knife scan to determine the beam position and size.

The `load_data()` function will return a data set, provided we supply the uid that we had previously recorded. It is possible to have multiple experimental runs within a single call to the run engine, and so our variable *uid* from above is actually a list of UIDs. Since there was only one run, we will just use the first (and only) entry: `uid[0]`.

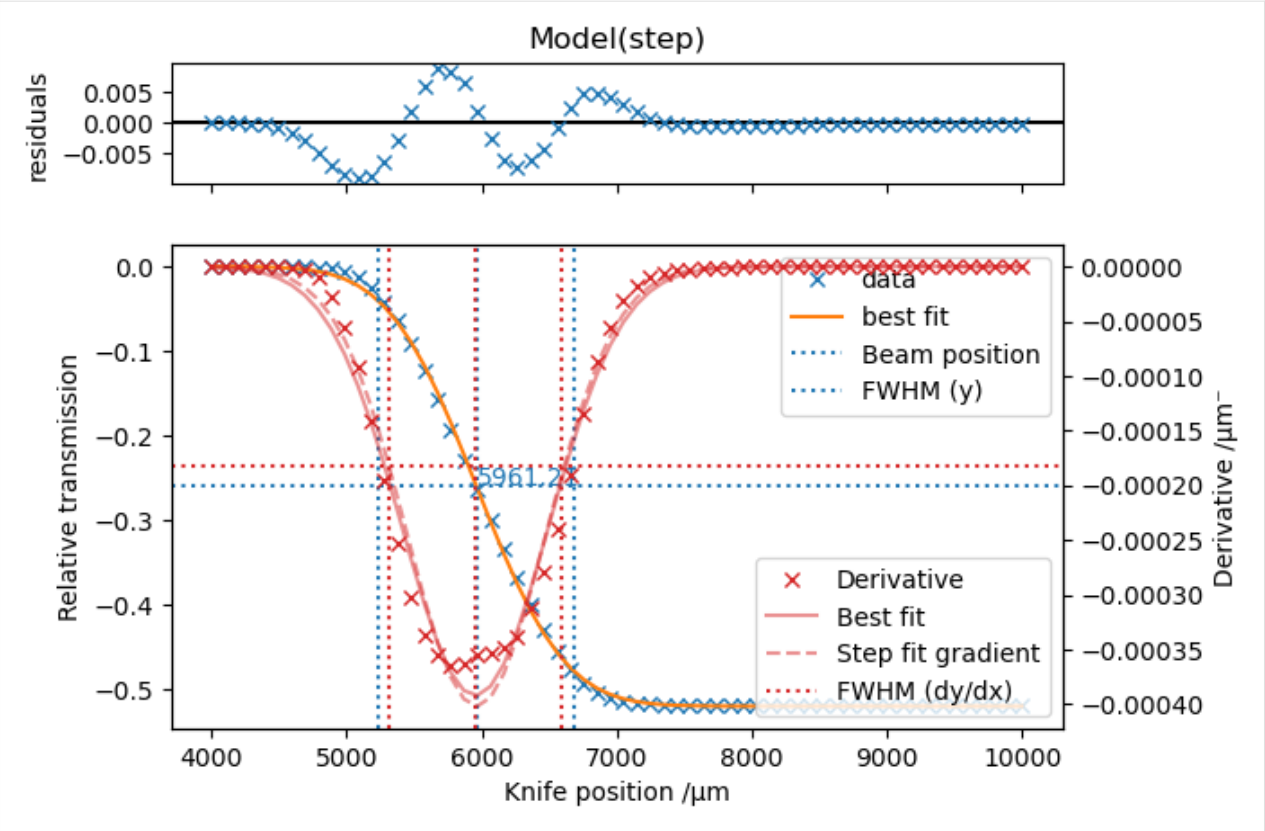
If the analysis is being done at a different time or place from running the scan, then the variable *uid* will probably not be set. In this case, it is possible to provide the UID that was printed above.

The `fit_step()` function is used to **analyze the knife scan data** to determine the beam position and size. Optionally, it will also plot the knife scan, its derivative, and the full-width at half-maximum for each. `fit_step()` needs the x and y data to process, which we can get from the data we loaded from the database. The name of the knife position will be the name of the motor you selected when running the scan. For the y position, we will calculate the transmission ($\frac{I_t}{I_0}$). There are several possible values for the ion chamber, so we must specify one (e.g. "I0_raw_counts", "I0_volts")

Optionally, the **data can be saved to a text CSV file** for additional analysis. First we will convert it to a pandas DataFrame and then use panda's `to_csv()` method. We will append the first segment of the UID to the filename to descrease the likelihood that we will overwrite data.

[9]:
```
# Uncomment this line to manually specify a UID
# uid = ["927fa7dd-e331-45ca-bb9d-3f89d7c65b17"]
# Load the data for the first (and only) UID in the list
data = haven.load_data(uid[0])
# Save the data to a CSV file, with tabs ("\t") instead of commas.
data.to_pandas().to_csv(f"knife_scan_example_{uid[0].split('-')[0]}.csv", sep='\t')
# Do the fitting
transmission = data["It_raw_counts"] / data["I0_raw_counts"]
properties = haven.fit_step(x=data["Focus V "], y=transmission, plot=True, plot_
↪derivative=True)
print(properties)
```

```
Properties(position=5961.212838126833, fwhm=1449.2198918394388)
```



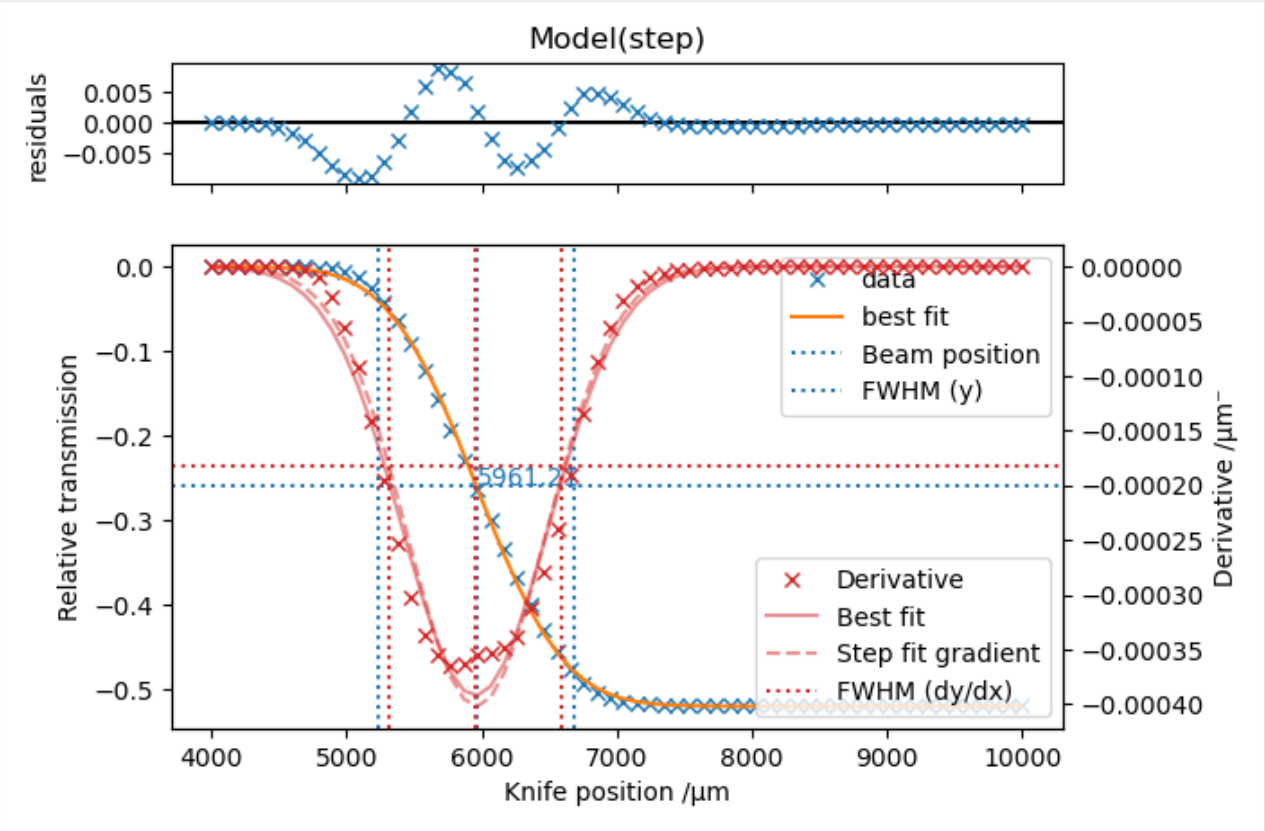
1.1.3 Loading Data From The Other Branch

By default, `haven.load_data()` will retrieve data for the branch corresponding to the computer you are using (e.g. microprobe branch from the microprobe computer and lerix branch for the lerix computer).

The can be changed by using passing the *catalog_name* argument to `haven.load_data()`.

[3]:
```
# This is the UID from a previous knife edge scan at the 25-ID-C microprobe branch.
# Since we explicitly select the s25idc catalog,
#   this cell will work from either the lerix or microprobe computers.
uid_from_previous_scan = "927fa7dd-e331-45ca-bb9d-3f89d7c65b17"
# Load the data by explicitly specifying the catalog to look in
data = haven.load_data(uid_from_previous_scan, catalog_name="s25idc")
# Do the fitting and plotting as normal
haven.fit_step(x=data["knife"], y=data.It_raw_counts/data.I0_raw_counts, plot=True, plot_
↪derivative=True)
```

[3]:
```
Properties(position=5961.212838126833, fwhm=1449.2198918394388)
```



This file is also available as an interactive jupyter notebook.

Download this file as a notebook (hint: right-click -> "save as").

1.2 Tutorial: XAFS Scans and Energy Scans

This notebook shows the following tasks:

- *Running a Single-Segment XANES scan*
- *Loading the Data*
- *Running a Multi-Segment XAFS Scan*
- *Running a Multi-Segment EXAFS Scan in K-space*
- *Modifying the List of Detectors*

First we have to **setup haven**, the beamline control library. Haven contains most of the tools we will use. We can import haven, setup the instrument, and create the run engine with the `start_haven` command. After the required steps are completed, it will deliver us into an ipython terminal.

We will also set metadata about who is running the beamline. This value will be saved in every plan executing on this run engine. This step is optional, but will allow database queries for scans taken by a specific person.

```
>>> RE.md["operator"] = "MFW"  # <- Put your initials in here
```

1.2.1 Running a Single-Segment XANES scan

Running a scan in bluesky is a **two step process**.

First, **create a plan**. A plan generates messages with instructions to do things like move a motor, wait for a motor to arrive at its destination, and trigger and read a detector. To create a plan, you call a function that will generate these messages. **Calling the function doesn't actually execute the scan.** In our case, `haven.xafs_scan(8325, 0.5, 1, 8350)` will create the plan, but the plan will not do anything unless used with a run engine.

The `xafs_scan()` plan requires at least four values: *(start, step, exposure, stop)*. *start* and *stop* mark the boundaries of the energy range, in eV. *step* is the space between energy points, in eV. Unless the range between *start* and *stop* is a whole multiple of *step*, the *stop* energy will not appear in the scan. *exposure* is the time, in seconds, for which to count at each energy.

The optional argument *E0* specifies the energy, in eV, of an x-ray absorbance edge. If given, all other energy values (i.e. *start* and *stop*) will be relative to *E0*.

```
>>> # These two plans will scan from 8323 eV to 8383 eV
>>> #   in 2eV steps with 1 sec exposure
>>> absolute_plan = haven.xafs_scan(8323, 2, 1, 8383)
>>> relative_plan = haven.xafs_scan(-20, 2, 1, 50, E0=8333)
```

Before running either of these plans, we can **verify that it will do what we expect** with the `simulators: summarize_plan()` helper. This function will print a human-readable description of all the steps that will be taken.

```
>>> summarize_plan(relative_plan)
```

Next, **execute the plan on the run engine**. As part of `start_haven`, we created a run engine. Now we will use this run engine to execute the plan. The run engine will read the messages and perform the appropriate tasks. We will also provide some meta-data, which will allow us to determine the purpose of these scans in the future. `simulators: summarize_plan()` consumed the plan so we have to create a new one.

When the run engine finishes the plan, it will return a unique identifier (UID). This UID is the best way to retrieve the data from the database. We will **save the UID** to a variable, and also print it to the page in case we want to recall it later.

```
>>> plan = haven.xafs_scan(-20, 2, 1, 50, E0=8333)
>>> # Run one of the plans with the previously created RunEngine
>>> uid = RE(plan, sample_name="Ni foil", purpose="training")
>>> print(uid)
```

1.2.2 Loading the Data

During execution the data are saved to a mongoDB database. Haven has tools to retrieve the data.

The `load_data()` function will return a data set, provided we supply the uid that we had previously recorded. It is possible to have multiple experimental runs within a single call to the run engine, and so our variable *uid* from above is actually a list of UIDs. Since there was only one run, we will just use the first (and only) entry: `uid[0]`.

If the analysis is being done at a different time or place from running the scan, then the variable *uid* will probably not be set. In this case, it is possible to provide the UID that was printed above.

Optionally, the **data can be saved to a text CSV file** for additional analysis. First we will convert it to a pandas DataFrame and then use panda's `to_csv()` method. We will append the first segment of the UID to the filename to descrease the likelihood that we will overwrite data.

```
>>> # Uncomment this line to manually specify a UID
>>> # uid = ["927fa7dd-e331-45ca-bb9d-3f89d7c65b17"]
>>> # Load the data for the first (and only) UID in the list
>>> data = haven.load_data(uid[0])
>>> # Save the data to a CSV file, with tabs ("\t") instead of commas.
>>> data.to_pandas().to_csv(f"xafs_scan_example_{uid[0].split('-')[0]}.csv", sep='\t')
>>> # Plot the result
>>> data["od"] = data["It_raw_counts"] / data["I0_raw_counts"]
>>> plt.plot(data['energy'], data['od'])
```

1.2.3 Running a Multi-Segment XAFS Scan

The `xafs_scan()` function can accept multiple sets of values to accomodate additional scan regions. After the first set of four parameters (*start*, *step*, *exposure*, *stop*), additional sets of three parameters (*step*, *exposure*, *stop*) can be given and will use the previous *stop* energy as its new *start* energy.

Additionally, Haven will look up the literature energy for a given X-ray absorption edge, in this case the Ni K-edge.

The call below will scan the following energies, relative to 8333 eV:

- -50 to -10 eV (8283 to 8323 eV) in 5 eV steps with 0.5 sec exposure
- -10 to +50 eV (8323 to 8383 eV) in 1 eV steps with 1 sec exposure
- +50 to +200 eV (8383 to 8533 eV) in 10 eV steps with 0.5 sec exposure

```
>>> multisegment_plan = haven.xafs_scan(-50, 5, 0.5,  # start, step, exposure
                                        -10, 1, 1,  # start, step, exposure
                                        50, 10, 0.5,  # start, step, exposure
                                        200,  # stop
                                        E0="Ni_K")
>>> # Run the plan with the previously created RunEngine
>>> uid = RE(multisegment_plan, sample_name="Ni foil", purpose="training")
>>> print(uid)
```

1.2.4 Running a Multi-Segment EXAFS Scan in K-space

The *xafs_scan()* function can also accept one energy segment as X-ray wavenumbers instead of X-ray energy using the *k_step*, *k_exposure* and *k_max* keyword-only parameters. *k_weight* controls the increasing exposure time at higher wavenumbers.

```
>>> exafs_plan = haven.xafs_scan(-200, 5, 1.0,  # start, step, exposure
                                 -20, .3, 1,  # start, step, exposure
                                 30,  # Last non-k energy point (also start of k-region␣
↪in eV)
                                 k_step=0.05, k_exposure=1.0, k_max=13.5, k_weight=0.5,
                                 E0=8331.0)
>>> # Run the plan with the previously created RunEngine
>>> uid = RE(exafs_plan, sample_name="Ni foil", purpose="training")
>>> print(uid)
```

1.2.5 Modifying the List of Detectors

By default, `xafs_scan()` measures all registered ion chambers, most likely those set up during `haven.load_instrument()` called above. This default list can be overridden by using the *detectors* argument. This example records only those scaler channels whose EPICS records' *.DESC* values are "It", "I0", or "Iref". Modify these names to suit your use case.

```
>>> detectors_plan = haven.xafs_scan(8323, 2, 1, 8383, detectors=["It", "I0", "Iref"])
>>> # Run the plan with the previously created RunEngine
>>> uid = RE(detectors_plan, LivePlot('It_raw_counts', 'energy_energy')
>>> print(uid)
```

1.3 Tutorial: Area Detectors

This tutorial covers the basics of using an area detector. Specifically:

- *Loading the Ophyd Device*
- *Verifying the Device Can Be Staged*
- *Running an XAFS Scan*

1.3.1 Loading the Ophyd Device

Note: This tutorial assumes the area detector has already been configured in Haven for use at the beamline. For setup instructions, see *Area Detectors and Cameras*.

First, open a terminal and run `start_haven`. After a brief wait, this will import some basic Haven and Bluesky objects and then present you with an ipython terminal.

Next we will **retrieve the device from Haven's device registry**. In this tutorial we will be using an Eiger S 500K area detector. We need to know the device name. To find it, we will ask the haven registry for all available devices.

```
In [1]: haven.registry.device_names
Out[1]:
['sim_motor',
 'eigerector',
 'energy',
 'monochromator',
 's25id-gige-A',
 'Shutter A',
 'Aerotech',
 'NHQ01_ch1',
 'NHQ01_ch2',
 'KB_slits',
 'eiger',
 'vortex_me4']
```

The second to last entry is the name of the device we want, so we will now retrieve it from the device registry:

```
In [2]: eiger = haven.registry.find("eiger")
```

1.3.2 Verifying the Device Can Be Staged

If this is the first time the detector has been used since the IOC was started, there may be additional steps required. To test this, we will see if the device can be staged.

```
In [3]: eiger.stage()
```

If the above function **returns without error**, then the device can be unstaged and is ready for use. Before we do that, lets just make sure we can trigger it.

```
In [4]: eiger.trigger().wait()

In [5]: eiger.unstage()
```

However, if **staging causes and exception about an unprimed plugin**, then we need to prime the plugin first. The following steps should prime the plugin:

- open the caQtDM panels (e.g. `start_25idSimDet_caqtdm`)
- open the plugins panel (under *Plugins* click the *All* button
- Ensure the offending plugins are enabled
- In the original camera panel, click *Start* button next to "Acquire" to collect a frame

Now we can stage and trigger the detector.

```
In [4]: eiger.stage()

In [5]: eiger.trigger().wait()

In [6]: eiger.unstage()
```

1.3.3 Running an XAFS Scan

First, we will verify that the detector is going to measure the correct signals for this detector:

```
In [7]: list(eiger.read_attrs)
```

Next, we will prepare the plan. By default, the `xafs_scan()` plan will only measure the ion chambers. To also trigger the area detector, we must include it both as a detector and as a time positioner (for setting exposure time).

```
In [7]: time_positioners = [eiger.cam.acquire_time, ion_chambers[0].exposure_time]

In [8]: detectors = [eiger, *ion_chambers]
```

Now we will **create an XAFS scan plan** with the following energies relative to the N-K edge (8333 eV):

- -200 eV to -30 eV
  - 10 eV steps
  - 1 second exposure
- -30 eV to +30 eV
  - 1 eV steps
  - 1 second exposure

- +30 eV to k=14
  - 0.05 steps
  - 1 second base exposure
  - k_weight = 0.5

```
In [9]: plan = haven.xafs_scan(-200, 10, 1, -30, 1, 1, 30, k_step=0.05, k_max=14, k_
↪exposure=1, k_weight=0.5, E0="Ni_K", time_positioners=time_positioners,␣
↪detectors=detectors)
```

Next we will summarize the plan to ensure that it is performing the steps we expect:

```
In [10]: summarize_plan(plan)
```

Inspect the output to ensure that it is measuring the correct detectors (`Read ['eiger', 'Iref', 'Ipreslit', 'It', 'IpreKB', 'I0dn', 'energy']`) and setting the correct energies (`energy -> 9069.77015484562`) and exposure times (`Iref_exposure_time -> 2.2221354183382798` and `eiger_cam_acquire_time -> 2.2221354183382798`).

Summarizing the plans consumes it, so we will build the plan again, and **run it in the run engine** along with some meta-data describing the sample and the reason we're measuring it:

```
In[12]: plan = haven.xafs_scan(-200, 10, 1, -30, 1, 1, 30, k_step=0.05, k_max=14, k_
↪exposure=1, k_weight=0.5, E0="Ni_K", time_positioners=time_positioners,␣
↪detectors=detectors)

In[13]: RE(plan, sample_name="Ni test sample", purpose="training")
```

This file is also available as an interactive jupyter notebook.

Download this file as a notebook (hint: right-click -> "save as").

1.4 Demonstration of Fly-Scanning

This notebook covers how to perform a fly-scan, where one motor moves continuously while one or more detectors acquire multiple data points.

This includes the following tasks: - preparing an axis on the Aerotech XY stage as a flyer - executing a 1D fly-scan over an Aerotech axis - loading 1D fly-scan data from the database - executing a 2D scan where one axis is flown - loading 2D fly-scan data from the database

First we have to import haven, the beamline control library. Haven contains most of the tools we will use. Importing it allows us to get to the functions and classes that are defined inside.

Next, the `haven.load_instrument()` function will read the configuration files and scan the hardware for its configuration. It will then **build the devices** that will be used for scans. This function prints out a list of motors that it has discovered.

Then, we create the run engine. The run engine is responsible for executing our scans and will be described in more detail when it is used below.

Lastly we set metadata about who is running the beamline. This value will be saved in every plan executing on this run engine. This step is optional, but will allow database queries for scans taken by a specific person.

[21]:
```
# Import support packages
import matplotlib.pyplot as plt
import numpy as np
# Import haven
import haven
# Load the motors and detectors
haven.load_instrument()
# The RunEngine will be responsible for executing the plans
RE = haven.run_engine()
# (Optional) Save the initials of the current beamline operator as metadata
RE.md["operator"] = "MFW"  # <- Put your initials in here
```

```
Could not connect to AravisDetector in 2.27 sec: Hutch A BPM.
Could not connect to AravisDetector in 2.27 sec: s25id-gige-A.
Could not connect to AravisDetector in 2.27 sec: s25id-gige-C.
Could not connect to AravisDetector in 2.27 sec: s25id-gige-D.
Could not connect to AravisDetector in 2.27 sec: s25id-gige-E.
Could not connect to ApsPssShutterWithStatus in 2.27 sec: front_end_shutter.
Could not connect to ApsPssShutterWithStatus in 2.27 sec: hutch_shutter.
Could not connect to ApsMachine in 2.27 sec: APS.
Could not connect to Monochromator in 2.21 sec: monochromator.
Could not connect to ApsUndulator in 2.21 sec: undulator.
Could not connect to energy positioner: energy
Could not connect to fluorescence detector: canberra_Ge7 (20xmap8:)
Could not connect to motor: 25ida:ORM1:m1
Could not connect to motor: 25ida:ORM1:m2
Could not connect to motor: 25ida:ORM1:m3
Could not connect to motor: 25ida:ORM1:m4
Could not connect to motor: 25ida:ORM2:m1
Could not connect to motor: 25ida:ORM2:m2
Could not connect to motor: 25ida:ORM2:m3
Could not connect to motor: 25ida:ORM2:m4
Could not connect to motor: 25ida:ORM2:m5
Could not connect to motor: 25ida:slits:m1
Could not connect to motor: 25ida:slits:m2
Could not connect to motor: 25ida:slits:m3
Could not connect to motor: 25ida:slits:m4
Could not connect to motor: 25ida:slits:m5
Could not connect to motor: 25ida:slits:m6
Could not connect to motor: 25ida:slits:m7
Could not connect to motor: 25ida:slits:m8
Could not connect to motor: 25ida:BPM:m1
APS device not found, suspenders not installed.
```

1.4.1 Preparing the Aerotech Flyer

We need to get the horizontal axis of the aerotech XY stage as an Ophyd device, and the ion chambers to use as detectors.

This can be done easily with the Haven registry.

We will also use a **0.2 sec dwell time** for the rest of this tutorial, which we set on the device now.

[2]:
```
aerotech = haven.registry.find("aerotech")
# Set dwell time here
dwell_time = 0.2  # seconds
aerotech.horiz.dwell_time.set(dwell_time).wait()
# Get ion chamber devices
ion_chambers = haven.registry.findall("ion_chambers")
```

1.4.2 1D Fly-Scan

Now we will **create the plan** to run the fly scan.

We need to provide the *start* and *end* positions for the scan. For easy comparison to regular step scans, the *start* and *stop* positions are the motor position at the center of the first and last bins of the scan.

We also need to inform the plan how often to make a new data bin. *num* tells the plan how many bins to create. The example code will produce 41 bins between 1000µm and 1000µm, which means each bin will cover 50µm.

[3]:
```
start, stop, num = (-1000, 1000, 41)
plan = haven.fly_scan(ion_chambers, aerotech.horiz, start, stop, num)
uid, = RE(plan, purpose="fly scanning tutorial", sample="")
print(f"Scan complete. UID: {uid}")
```

```
/home/beams0/S25IDCUSER/micromamba/envs/haven-dev/lib/python3.9/site-packages/event_
↪model/__init__.py:208: UserWarning: The document type 'bulk_events' has been␣
↪deprecated in favor of 'event_page', whose structure is a transpose of 'bulk_events'.
  warnings.warn(
```

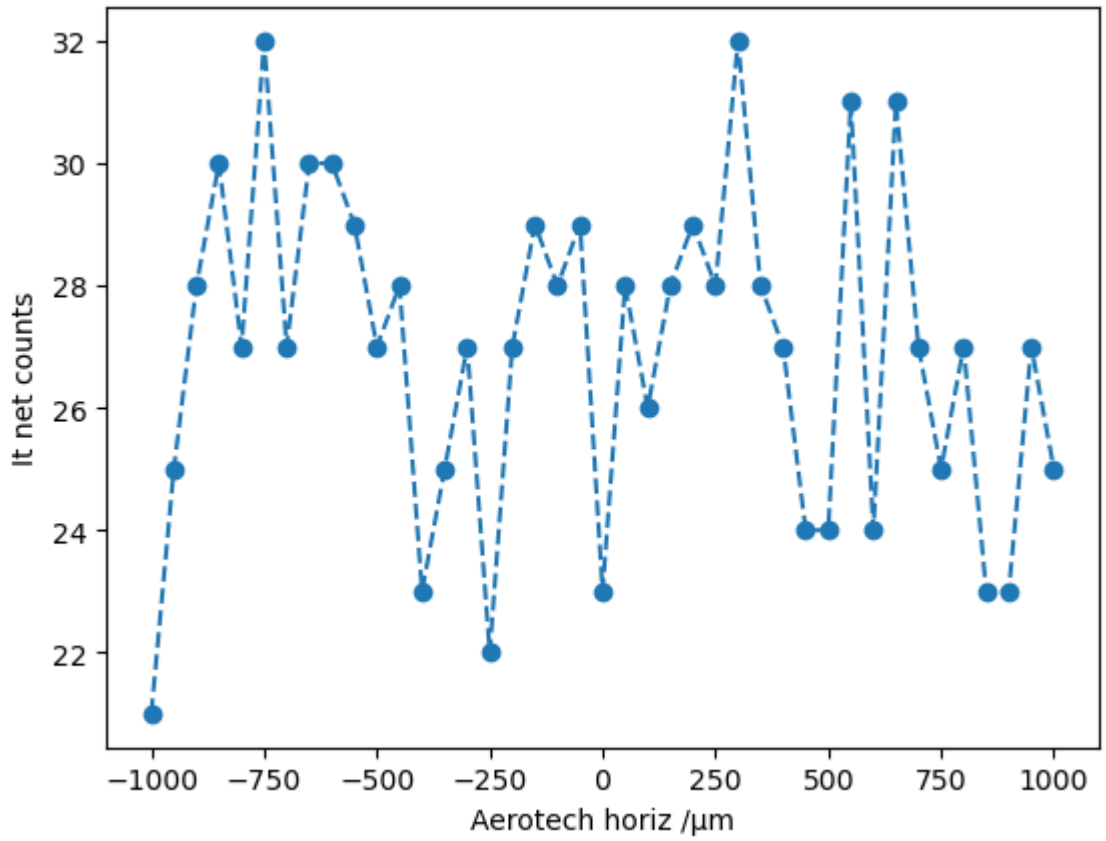
```
Scan complete. UID: 0ebb79bd-eea1-4ff2-8a2e-5c4915cd13fc
```

Viewing 1D Fly-Scan Results

[41]:
```
# Load the data from the mongodb database
client = haven.tiled_client()
uid = "0ebb79bd-eea1-4ff2-8a2e-5c4915cd13fc"
data = client[uid]['primary']['data'].read()

# Create a new set of axes for plotting
plt.figure()
ax = plt.gca()
# Plot It versus motor position (w/o offset for now)
ax.plot(data.aerotech_horiz, data.Ipre_KB_net_counts, marker='o', linestyle="--")
ax.set_xlabel("Aerotech horiz /µm")
ax.set_ylabel("It net counts")
```

[41]:
```
Text(0, 0.5, 'It net counts')
```



1.4.3 2D Map Fly-Scanning

In this section we will fly the horizontal axis and step the vertical axis.

Instead of `fly_scan()` we will use `grid_fly_scan()`.

[7]:
```
# Set parameters for fly scan here
step_params = (-1500, 500, 21)  # (start, stop, num)
fly_params = (-1000, 1000, 41)  # (start, stop, num)
# Create the plan, slow axis listed first
plan = haven.grid_fly_scan(ion_chambers, aerotech.vert, *step_params, aerotech.horiz,␣
↪*fly_params, snake_axes=True)
# Execute the plan
uid, = RE(plan, purpose="fly scanning tutorial", sample="")
print(f"Scan complete. UID: {uid}")
```

```
/home/beams0/S25IDCUSER/micromamba/envs/haven-dev/lib/python3.9/site-packages/event_
↪model/__init__.py:208: UserWarning: The document type 'bulk_events' has been␣
↪deprecated in favor of 'event_page', whose structure is a transpose of 'bulk_events'.
  warnings.warn(
```

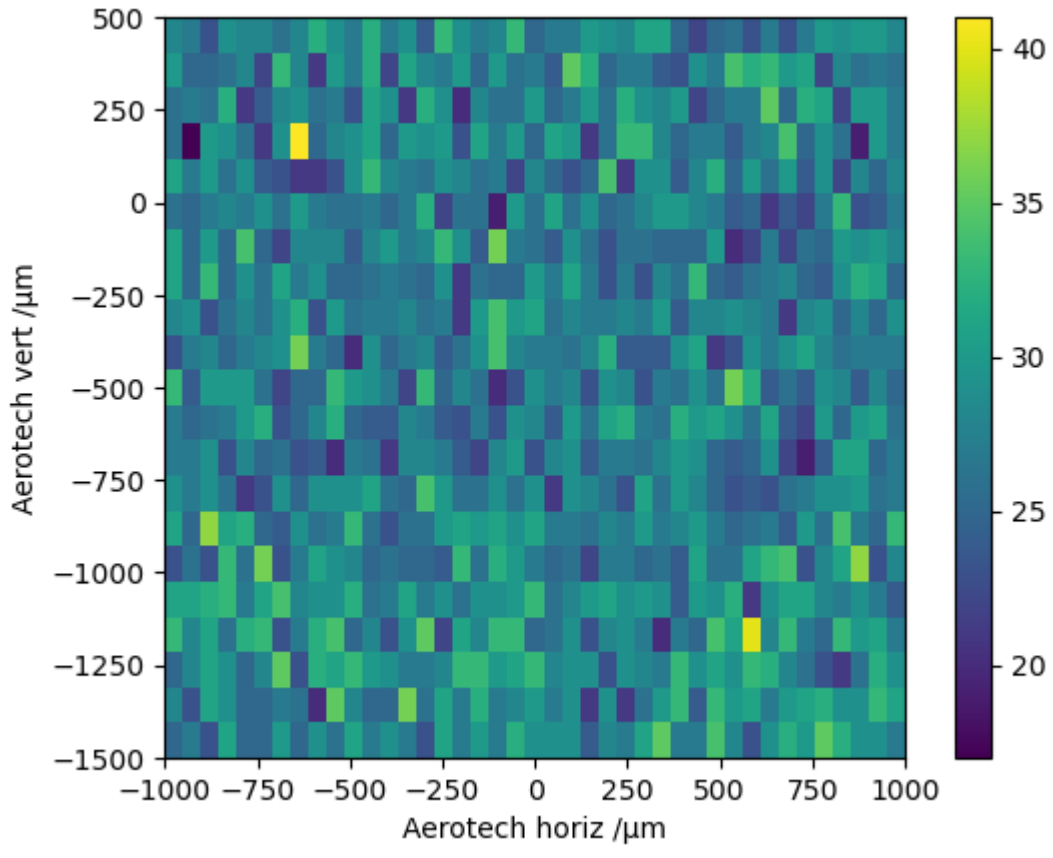
```
Scan complete. UID: 5e00f671-7b4b-4215-8ed9-224566a6fa35
```

Viewing 2D Fly-Scan Results

[39]:
```
# Load the data from the mongodb database
client = haven.tiled_client()
uid = "5e00f671-7b4b-4215-8ed9-224566a6fa35"
# Get the shape of the map from the metadata
metadata = client[uid].metadata
extent = (*metadata['start']['extents'][1], *metadata['start']['extents'][0])
shape = metadata['start']['shape']
# Read and re-shape the data
data = client[uid]['primary']['data'].read()
```

[40]:
```
# Create a new set of axes for plotting
plt.figure()
ax = plt.gca()
# Plot a map of It measurements (w/o offset for now)
I0 = np.reshape(np.asarray(data.Ipre_KB_net_counts), shape)
im = ax.imshow(I0, extent=extent, origin="lower")
plt.colorbar(im, ax=ax)
ax.set_xlabel("Aerotech horiz /µm")
ax.set_ylabel("Aerotech vert /µm")
```

[40]:
```
Text(0, 0.5, 'Aerotech vert /µm')
```



This file is also available as an interactive jupyter notebook.

Download this file as a notebook (hint: right-click -> "save as").

CHAPTER

TWO

HOW-TO GUIDES

These guides cover a particular topic in depth, coving material useful to users and beamline staff.

These guides do not cover the individual beamlines thoroughly.

2.1 Area Detectors and Cameras

Area detectors are all largely the same but with small variations from device-to-device. All the device definitions for area detectors are in the `haven.instrument.area_detector` module.

Currently supported detectors:

- Eiger 500K (`Eiger500K`)
- Lambda (`Lambda250K`)
- Simulated detector (`SimDetector`)

EPICS and Ophyd do not make a distinction between area detectors and cameras. After all, a camera is just an area detector for visible light.

In Haven, the device classes are largely the same. The only substantive difference is that cameras have the ophyd label "cameras", whereas non-camera area detectors (e.g. Eiger 500K), have the ophyd label "area_detectors". They can be used interchangeably in plans.

> **Warning:** Currently, cameras are not properly implemented in Haven. This will be fixed soon.

2.1.1 Using Devices in Haven

If the device you are using already has a device class created for it, then using the device only requires a suitable entry in the iconfig file (`~/bluesky/instrument/iconfig.toml`). The iconfig section name should begin with "area_detector", and end with the device name (e.g. "area_detector.eiger"). The device name will be used to retrieve the device later from the instrument registry.

The key "prefix" should list the IOC prefix, minus the trailing ":". The key "device_class" should point to a subclass of ophyd's `DetectorBase` class that is defined in `haven.instrument.area_detector`.

```
[area_detector.eiger]

prefix = "dp_eiger_xrd91"
device_class = "Eiger500K"
```

Once this section has been added to `iconfig.toml`, then the device can be loaded from the instrument registry. No special configuration is generally necessary.

```
>>> import haven
>>> haven.load_instrument()
>>> det = haven.registry.find("eiger")
>>> plan = haven.xafs_scan(..., detectors=[det])
```

Usually, no special configuration is needed for area detectors. By default it will save HDF5 and TIFF files for each frame. The filenames for these TIFF and HDF5 files will be stored automatically to the database. The outputs of the stats plugins will also be saved.

Warning: It is up you to make sure the file path settings are correct for the HDF5 and TIFF NDplugins. Also, ensure that the routing is correct for the ROI and STATS NDplugins.

Warning: The first time you stage the device after the IOC has been restarted, you may receive an error about the **plugin not being primed**. The means that the plugin does not know the size of the image to expect since it has not seen one yet. The solution is to open the caQtDM panels for the detector, ensure the corresponding plugins are enabled, and then manually acquire a frame.

2.2 Making Changes to Haven and Contributing

Two Scenarios are likely when proposing changes to Haven:

- New feature or bugfix written in a development environment (preferred)
- Troubleshooting the beamline during beamtime

2.2.1 From a Development Environment

The preferred way to modify Haven is to fork the main repository on github, make changes on a new branch, and then submit a pull request back to the main repository. This section assumes you **have an active github account** (if not, sign up for one first).

The following steps are **only required the first time** you work on Haven. Once done, the forked repository and local environment can be reused.

1. Install a git client on your local computer (e.g. git or Github Desktop)
2. Create a fork of the main Haven repository
3. Clone the forked repository to your local computer (e.g. `git clone git@github.com:canismarko/haven.git`)
4. Install an anaconda-like distribution environment (mamba-forge is recommended)
5. Create a new conda environment from *environment.yml* (e.g. `mamba env create -n haven -f haven/environment.yml`)
6. Activate the newly created conda environment (e.g. `mamba activate haven`)
7. Install haven in the environment (`pip install -e "haven[dev]"`)
8. Verify that the *test-suite passes*

The following steps should then be performed every time a new feature is being added or bug is being fixed.

9. Sync your github fork with the main github repository
10. Pull changes to your local repository (`git pull`)
11. Create a new git branch for the task you are doing (e.g. `git checkout -b area_detector_support`)
12. Make changes to the Haven source code as needed
13. Ensure all tests pass (`pytest`)
14. Commit changes to your local branch (`git add file1.py file2.py ...` and `git commit`)
15. Push changes back to github (`git push`)
16. Create a pull request on github to send changes back to the main repository.

Running Tests

Pytest is the recommended runner for Haven. Once the environment is properly setup, the tests can be run using:

```
$ pytest
```

`pytest` should not report any errors or failures, though skipped, xfailed, and warnings are expected.

While running the tests, devices created using `make_device()` will be replaced with simulated devices using Ophyd's *sim* module. This means that `load_instrument()` can be called without hardware being present, and the corresponding fake devices can be found in the `haven.registry`.

Additionally, some pytest fixtures are provided that create simulated devices, (e.g. ion chambers) and can be used directly in your tests.

More details can be found in the file *haven/tests/conftest.py*.

2.2.2 From the Beamline

> **Warning:** This section is intended for qualified beamline staff. **Users are not authorized** to make changes to the beamline software without staff involvement.
>
> If at all possible, changes should be made through a development environment as described above.

User support often requires changes to be made quickly from the beamline computers.

Git is our version control software. It interacts with github, and allows changes to the source code to be tracked and managed.

Before modifying Haven, create a new branch using git. This will allow changes to be undone or pushed to github for use at other beamlines. First we will create the new branch, then we will check it out to begin working on it.

```
$ cd ~/haven
$ git branch broken_shutter_workaround
$ git checkout broken_shutter_workaround
```

Now modify the Haven scripts as needed to get the beamline running. Once the changes are complete, **commit them to version control**. If **new files have been added**, then we have to inform git that they should be included, for examples:

```
$ git add haven/shutter_workaround.py
```

Then **commit the changes**:

```
$ git commit -a -m "Workaround for the shutter not also closing when requested."
```

If you see `black...Failed`, then you need to run the command again. Black is an add-on that enforces its own code format so that we can focus on the important stuff, and it runs every time changes are committed. If code needs to be reformatted, it stops the commit and fixes the formatting. Attempting the commit again with the reformatted code usually works.

The `-a` option tells git to automatically include all files that have been changed. The `-m` option lets us include a short message describing the commit. Please **write descriptive commit messages**. For longer messages, omit the -m option (just `git commit -a`) and a text editor will appear.

Now the new branch can be pushed to github with

```
$ git push -u origin delete_me
```

The `-u` option is only needed the first time: it tells git to connect the new branch to github (origin).

2.2.3 Design Defense

An important consideration is how to manage changes to the code-base in a way that satisfies several goals:

1. maximize reuse of code between beamlines (9-BM, 20-BM, and 25-ID)
2. support rapid troubleshooting at the beamline
3. control deployment of new features among the beamlines
4. encourage documentation and testing

Rapid troubleshooting necessarily leads to the code-base being in an untested state, and so these changes should not automatically apply to the code-base in use at another beamline.

The idea presented here is to have each beamline own a local copy of the haven repository. Changes made at the beamline should ideally be made to a separate branch. If the change is worth keeping it can be committed along with documentation and tests, and the new branch can be merged into the main branch.

Getting those changes to the other beamlines can be done whenever no experiments are taking place there. We can pull the changes from github, and run the system tests.

Using a common network folder for the scripts would satisfy requirements 1 and 2, but not 3 and 4. Having entirely separate sets of scripts would satisfy requirement 2, but not 1, 3, or 4. The approach described here aims to strike a balance between the 4 requirements.

2.3 Configuration Files

Table of Contents

2.3.1 Motivation

Haven's goal is to **provide support for all of the spectroscopy beamlines**. However, each beamline is different, and these differences are **managed by a set of configuration files**, similar to the .ini files used in the old LabView solution. To keep the complexity of these configuration files manageable, Haven gets much of the needed information from the IOCs directly.

Haven/Firefly should always load without a specific configuration file, but will probably not do anything useful.

2.3.2 Checking Configuration

If Haven is installed with pip, the command `haven_config` can be used to read configuration variables as they will be seen by Haven:

```
$ haven_config beamline
{'is_connected': False, 'name': 'SPC Beamline (sector unknown)'}
$ haven_config beamline.is_connected
False
```

2.3.3 Configuration File Priority

There are several sources of configuration files, described in detail below. They are loaded in the following order, with lower numbers taking precedence over higher numbers.

1. Files listed in the `$HAVEN_CONFIG_FILES`
2. `~/bluesky/instrument/iconfig.toml` (for backwards compatibility)
3. `~/bluesky/iconfig.toml` (best place)
4. `iconfig_default.toml` packaged with Haven

Unless there's a good reason to do otherwise, **most beamline configuration belongs in ~/bluesky/iconfig.toml**.

For example, to enable support for our Universal Robotics robot *Austin* to 25-ID-C, open the file `~/bluesky/iconfig.toml` and add the following:

```
[robot.Austin]
prefix = "25idAustin"
```

Note: The prevent accidental changes, the bluesky configuration files may not be writable by the user accounts at the beamline. For example, at 25-ID, the user account does not have permission to write to `~/bluesky/iconfig.toml`

so **changes must be made as the staff account**.

HAVEN_CONFIG_FILES Environment

If the environmental variable HAVEN_CONFIG_FILES is set to a *comma-separated* list of file path, then these files will take priority, with later entries superseding earlier entries.

~/bluesky/iconfig.toml

The file ~/bluesky/iconfig.toml will be read if it is present. **This is the best place to put beamline-specific configuration.**

The file ~/bluesky/instrument/iconfig.toml is also read for backwards compatibility. It should not be used for new deployments, and support for it may be removed without warning.

iconfig_default.toml

Haven includes an set of default configuration values in src/haven/iconfig_default.toml. This is mainly so that Haven and Firefly can still run during development without a dedicated configuration file. It also serves as a starting point for deploying Haven to a new beamline. See the section on testing below for suggestions on how to add default configuration.

2.3.4 Development and Testing

While adding features and tests to Haven, it is often necessary to read a configuration file, for example when testing functions that load devices through load_instrument(). However, the configuration that is loaded should not come from a real beamline configuration or else there is a risk of controlling real hardware while running tests.

To avoid this problem, **pytest modifies the configuration file loading** when running tests with pytest:

1. Ignore any config files besides iconfig_default.toml.
2. Add iconfig_testing.toml to the configuration

Additionally, all load_motors() style functions should accept an optional *config* argument, that will determine the configuration instead of using the above-mentioned priority.

If a feature is added to Haven that would benefit from beamline-specific configuration, it can be added in one of two places.

src/haven/iconfig_default.toml
: This is the best choice if the device or feature is critical to the operation of Haven and/or Firefly, such as the beamline scheduling system. The values listed should still not point at real hardware, but should be sensible defaults or dummy values to allow Haven to function.

src/haven/iconfig_testing.toml
: This is the best choice if the device or hardware is optional, and may or may not be present at any given beamline, for example, fluorescence detectors. This configuration should not point to real hardware.

2.3.5 Example Configuration

Below are some examples of configuration that can be re-used for new devices support or beamline setup.

Listing 1: iconfig_default.toml

```
# Defaults go here, then get updated by toml loader
[beamline]

name = "SPC Beamline (sector unknown)"
is_connected = false

[facility]

name = "Advanced Photon Source"
xray_source = "insertion device"

[database.databroker]

catalog = "bluesky"

# Cameras
# =======

[camera]

imagej_command = "imagej"
```

Listing 2: iconfig_testing.toml

```
[bss]

prefix = "255idc:bss"
beamline = "255-ID-C"

[xray_source]

type = "undulator"
prefix = "ID255:"

[queueserver]
kafka_topic = "s255idc_queueserver"
control_host = "localhost"
control_port = "60615"
info_host = "localhost"
info_port = "60625"

[database.tiled]

uri = "http://localhost:8337/"
entry_node = "255id_testing"

[shutter]
```

(continues on next page)

(continued from previous page)

```
prefix = "PSS:99ID"

[shutter.front_end_shutter]

hutch = "A"
# open_pv = "PSS:99ID:FES_OPEN_EPICS.VAL"
# close_pv = "PSS:99ID:FES_CLOSE_EPICS.VAL"
# status_pv = "PSS:99ID:A_BEAM_PRESENT"

[shutter.hutch_shutter]

hutch = "C"
# open_pv = "PSS:99ID:SCS_OPEN_EPICS.VAL"
# close_pv = "PSS:99ID:SCS_CLOSE_EPICS.VAL"
# status_pv = "PSS:25ID:C_BEAM_PRESENT"

[undulator]

ioc = "id_ioc"

[monochromator]

ioc = "mono_ioc"
ioc_branch = "UP"  # For caQtDM macros


[ion_chamber]

[ion_chamber.scaler]
prefix = "scaler_ioc"
channels = [2]

[ion_chamber.preamp]
prefix = "preamp_ioc"

[ion_chamber.voltmeter]
prefix = "255idc:LabjackT7_"  # Don't include the labjack number

# Motors
# ======
#
# Add a new section for each IOC (or IOC prefix) that has motors
# matching the format {prefix}:m1. The heading of the subsection
# ("VME_crate" in the example below) is a human-readable name that
# will become a label on the Ophyd Device. *num_motors* determines how
# many motors will be read. The example below will load three motors
# with PVs: "vme_crate_ioc:m1", "vme_crate_ioc:m2", and
# "vme_crate_ioc:m3".

[motor.VME_crate]
prefix = "255idVME"
num_motors = 3
```

(continues on next page)

(continued from previous page)

```
# Keys for camera definitions must begin with "cam" (e.g. "camA", "camB")
[camera.camA]

name = "s25id-gige-A"
description = "GigE Vision A"
prefix = "255idgigeA"

[aerotech_stage.aerotech]

prefix = "255idc"
delay_prefix = "255idc:DG645"
pv_vert = ":m1"
pv_horiz = ":m2"

[power_supply.NHQ01]

prefix = "ps_ioc:NHQ01"
n_channels = 2

[slits.KB_slits]

prefix = "vme_crate_ioc:KB"
device_class = "BladeSlits"

[slits.whitebeam_slits]
# A single rotating aperture slit, like the 25-ID white/pinkbeam slits
prefix = "255ida:slits:US:"
device_class = "ApertureSlits"
pitch_motor = "m33"
yaw_motor = "m34"
horizontal_motor = "m35"
diagonal_motor = "m36"

# A bendable mirror, like the long KB at 25-ID-C
[kb_mirrors.LongKB_Cdn]
prefix = "255idcVME:LongKB_Cdn:"
horiz_upstream_motor = "m33"
horiz_downstream_motor = "m34"
horiz_upstream_bender = "m21"
horiz_downstream_bender = "m22"
vert_upstream_motor = "m46"
vert_downstream_motor = "m47"
vert_upstream_bender = "m56"
vert_downstream_bender = "m57"

# A non-bendable mirror, like the KB at 25-ID-C
[kb_mirrors.KB]
prefix = "255idcVME:KB:"
horiz_upstream_motor = "m35"
horiz_downstream_motor = "m36"
vert_upstream_motor = "m48"
```

(continues on next page)

(continued from previous page)

```
vert_downstream_motor = "m49"

# A single bounch, high heat load mirror
[mirrors.ORM1]
prefix = "25ida:ORM1:"
device_class = "HighHeatLoadMirror"
bendable = false

[mirrors.ORM2]

prefix = "25ida:ORM2:"
device_class = "HighHeatLoadMirror"
bendable = true

# An optical table with two vertical motors
[table.downstream_table]
prefix = "255idcVME:"
transforms = "table_ds_trans:"
pseudo_motors = "table_ds:"
upstream_motor = "m21"
downstream_motor = "m22"

# An optical table with one vertical motor and one horizontal motor
[table.upstream_table]
prefix = "255idcVME:"
vertical_motor = "m26"
horizontal_motor = "m25"

[area_detector.sim_det]

prefix = "255idSimDet"
device_class = "SimDetector"

[lerix.lerix.rowland]

x_motor_pv = "255idVME:m1"
y_motor_pv = "255idVME:m2"
z_motor_pv = "255idVME:m3"
z1_motor_pv = "255idVME:m4"

[heater.capillary_heater]

prefix = "255idptc10"
device_class = "CapillaryHeater"

[robot.A]
prefix = "255idAustin"

# Managed IOC control PVs
[iocs]
255idb = "glados:ioc255idb"
255idc = "glados:ioc255idc"
```

(continues on next page)

(continued from previous page)

```
[fluorescence_detector]

[dxp.vortex_me4]

prefix = "vortex_me4"
num_elements = 4

[dxp.canberra_Ge7]

prefix = "20xmap8"
num_elements = 4

[xspress.vortex_me4_xsp]

prefix = "vortex_me4_xsp"
num_elements = 4
```

2.4 Defining a New Motor

2.5 Energy Scans (XAFS)

2.5.1 `xafs_scan()` for Straight-Forward XAFS Scans

The `xafs_scan()` is a bluesky plan meant for scanning energy over a a number of energy ranges, for example the pre-edge, edge, and EXAFS signal of a K-edge.

The function accepts an arbitrary number of parameters for defining the ranges. The parameters are expected to provide energy step sizes (in eV) and exposure times (in sec) between the boundaries of the ranges. They should be passed following the pattern:

`energy, step, exposure, energy, step, exposure, energy, ...`

An example across the Nickel K-edge at 8333 eV could be:

```
RE(xafs_scan(8313, 2, 1, 8325, 0.5, 2, 8365, 10, 1.5, 8533))
```

`RE` is the bluesky `RunEngine`, which should already be imported for you in the ipython environment.

Absolute vs. Relative Scans

In many cases, it is more intuitive to describe the energy ranges relative to some absorption edge (*E0*). If this *E0* energy is given directly to `xafs_scan()` via the *E0* argument, then all energy points will be interpreted as relative to this energy. The same scan from above would be:

```
RE(xafs_scan(-20, 2, 1, -8, 0.5, 2, 32, 10, 1.5, 200, E0=8333))
```

2.5.2 Defining Scans in K-Space

For extended structure scans (EXAFS), it may be more helpful to define the EXAFS region in terms of the excited electron's wavenumber (k-space). This can be done with the keyword arguments *k_step*, *k_exposure*, and *k_max*. Providing *E0* is necessary, since otherwise wavenumbers will be calculated relative to 0 eV, and will not produce sensible results.

```
RE(xafs_scan(-20, 2, 1, -8, 0.5, 2, 32, k_step=0.02, k_max=12, k_exposure=1., E0=8333))
```

Better quality results can sometimes be achieved by setting longer exposure times at higher *k*. The *k_weight* parameter will scale the exposure time geometrically with *k*. `k_weight=0` will produce constant exposure times, and if `k_weight=1` then exposure will scale linearly with *k*.

```
RE(xafs_scan(-20, 2, 1, -8, 0.5, 2, 32, k_step=0.02, k_max=12, k_exposure=1., k_weight=1,
↪ E0=8333))
```

2.5.3 `energy_scan()` for More Sophisticated Scans

For extra flexibility, use the `energy_scan()` plan, which accepts a sequence of energies to scan. For example, to scan from 8325 to 8375 eV in 1 eV steps:

```
energies = range(8325, 8376, step=1)
RE(energy_scan(energies))
```

Notice the range ends at 8376 eV instead of 8375 eV, since the last value is not included when using a `range`.

The *exposure* time can also be given. *exposure* can either be a single number to be used for all energies, or a sequence of numbers with the same length as *energies*, and each energy will use the corresponding exposure:

```
import numpy as np
energies = range(8325, 8376, step=1)
exposures = np.linspace(0.5, 5, num=len(energies))
RE(energy_scan(energies), exposure=exposures)
```

Building a more complicated set of energies can be made simpler using the `ERange` helper class:

```
energies = ERange(8325, 8375, E_step=1).energies()
RE(energy_scan(energies))
```

To make things even easier, `energy_scan()` can accept energy range objects directly:

```
energies = [
    8300, 8320,  # Individual energies are okay too, you can mix and match
    ERange(8325, 8375, E_step=0.5),
    ERange(8375, 8533, E_step=5),
]
RE(energy_scan(energies))
```

Other than including the ending energy in the list, this usage does not provide considerable value. However, the inclusion of multiple energies with different exposure times makes the value more clear, since *energy_scan* will automatically replace an `ERange` instance with the result of the instance's `energies()` method, and add equivalent entries into *exposure* based on the instance's `exposures()` method.

```
energies = [
    ERange(8325, 8375, E_step=0.5, exposure=1.5),
    ERange(8375, 8533, E_step=5, exposure=0.5),
]
RE(energy_scan(energies))
```

There is also a similar `KRange` that works similarly except using electron wavenumbers (k) instead of X-ray energy. This allows the energies to be given in a more intuitive way for EXAFS:

```
energies = [
    ERange(-50, 50, E_step=0.5, exposure=1.5),
    ERange(50, 200, E_step=5, exposure=0.5),
    KRange(200, 14, k_step=0.05, , k_weight=1., exposure=1.),
]
RE(energy_scan(energies, E0=8333))
```

Notice that the energies are now given relative to the edge energy *E0* (the nickel K-edge in this case). This is almost always necessary when using a `KRange` instance, since otherwise the corresponding energies would be relative to a free, zero-energy electron, instead of core electrons. *E0* can also be given as a string, in this case `E0="Ni_K"`.

At this point, we have largely replicated the behavior of `xafs_scan()` described above. In fact, `xafs_scan()` is a wrapper around `energy_scan()` whose main purpose is to take the parameters in the form of `(energy, step, exposure, energy, ...)`, and convert them to `ERange` and `KRange` instances.

2.5.4 Changing Detectors or Positioners

For more sophisticated scans, it may be necessary to include additional detectors. By default, `xafs_scan()` and `energy_scan()` will measure the ion chambers as detectors (those returned by `haven.registry.findall("ion_chambers")`). Both plans accept the *detectors* argument, which can be any of the following:

1. A list of devices.
2. A list of names/labels of devices.
3. A single name/label for devices.

Options 1 and 2 can be intermingled. For example:

```
eiger = haven.registry.find("eiger")
detectors = [eiger, "ion_chambers"]
plan = haven.xafs_scan(..., detectors=detectors)
```

Supplying the *detectors* argument will ensure that the detectors are captured in the data streams, but it may still be necessary to **specify positioners for setting the exposure time**. By default, only the ion chambers will receive have their exposure time set. This is especially important when using the *k_weight* parameter to `xafs_scan()` or the *exposure* parameter to `energy_scan()`.

Both plans accept a *time_positioners* argument for this purpose, which should be a list of entries similar to those accepted for *detectors* described above but with positioners for the various detectors. Extending the above example:

```
eiger = haven.registry.find("eiger")
detectors = [eiger, "ion_chambers"]
time_positioners = [eiger.cam.acquire_time, "ion_chambers.exposure_time"]
plan = haven.xafs_scan(..., detectors=detectors, time_positioners=time_positioners)
```

The above example actually uses all of the ion chambers' exposure times as separate positioners. This will work but produces extra messages and may be confusing. Since counting is handled by the scaler, any of the ion chambers on the same scaler can be used as a time positioner:

```
ion_chambers = haven.registry.findall("ion_chambers")
time_positioners = [eiger.cam.acquire_time, ion_chambers[0].exposure_time]
plan = haven.xafs_scan(..., time_positioners=time_positioners)
```

Lastly, we may want to **specify a different energy position** for example when using a secondary monocrhomator. By default the "energy" positioner is used, which is a pseudo-positioner that controls both the monochromator and the insertion device (if present). This positioner temporariy **disables the EPICS-based pseudo-motor** in use at sector 25-ID since the done status is not properly reported for the insertion device when using the EPICS implementation.

The *energy_positioners* argument accepts similar types as the previous options just discussed, and each one will be set to the energy in electron-volts at each point. For example, to scan only the monochromator energy we could do:

```
mono_energy = haven.registry.find("monochromator.energy")
plan = haven.energy_scan(..., energy_positioners=[mono_energy])
```

or equivalently:

```
plan = haven.energy_scan(..., energy_positioners="monochromator.energy")
```

2.6 Fluorescence Detectors

Table of Contents

2.6.1 Specifying Detectors in Configuration

To add new detectors to the beamline, new sections should be added the *iconfig.toml* file. Each section should be labeled `[<class>.<name>]`, where `<class>` specifies which interface is present (`"dxp"` for XIA DXP or `"xspress"` for Xspress3), and `<name>` becomes the device name. *prefix* is the PV prefix for the EPICS IOC, and *num_elements* specifies the number of detector elements.

```
[dxp.vortex_me4]

prefix = "20xmap4b"
num_elements = 4
```

(continues on next page)

(continued from previous page)

```
[xspress.vortex_ex]

prefix = "dp_xsp3_2"
num_elements = 1
```

The device can then be retrieved from the instrument registry for use in bluesky plans:

```
import haven

# Get individual fluorescence detectors
my_detector = haven.registry.find(name="vortex_me4")
another_detector = haven.registry.find(name="vortex_ex")

# Get all fluorescence detectors of any kind (e.g. DXP, Xspress3, etc.)
detectors = haven.registry.findall(label="fluorescence_detectors")
```

2.6.2 Common Behavior

Fluorescence detectors are implemented as `Xspress3Detector` and `DxpDetector` Ophyd device classes. They are written to have a common Ophyd interface so that clients (e.g. Firefly) can use fluorescence detectors interchangeably.

Creating Devices

By default, devices created from these device classes include one MCA element, available on the `mcas` attribute. The **recommended way to create a fluorescence detector** device directly is with the `load_xspress()` and `load_dxp()` factory functions:

```
from haven import load_xspress

det = load_xspress(name="vortex_me4",
                   prefix="20xmap4b",
                   num_elements=4)
det.wait_for_connection()
```

Alternately, to make a dedicated subclass with a specific number of elements, override the `mcas` attributes:

```
from haven.instrument import xspress

class Xspress4Element(xspress.Xspress3Detector):
    mcas = xspress.DDC(
        xspress.add_mcas(range_=range(4)),
        kind=(Kind.normal | Kind.config),
        default_read_attrs=["mca0", "mca1", "mca2", "mca3"],
        default_configuration_attrs=["mca0", "mca1", "mca2", "mca3"],
    )
```

Managing Elements and ROIs

Note: Not all fluorescence detector IOCs agree on how to number MCAs and ROIs. To maintain a unified interface, Haven uses the convention to start counting from 0 regardless of the IOC. As such, the haven device signals may be misaligned with the PVs they map to.

For example on a DXP-based IOC, an ophyd signal `det.mcas.mca1.rois.roi1` will have a PV like `xmap_4b:MCA1.R0`.

By default all elements (MCAs) will collect spectra, and **all ROIs will save aggregated values**. While this setup ensures that no data are lost, it also creates a large number of signals in the database and may make analysis tedious. Most likely, only some ROIs are meaningful, so those signals can be identified by giving them the `hinted` kind.

https://blueskyproject.io/ophyd/user/reference/signals.html#kind

During the staging phase (in its `stage()` method), each ROI will check this signal and if it is true, then it **will change its kind** to `hinted`. When unstaging, the signal is reset to its original value.

Individual **ROIs can be marked for hinting** by setting the `use` signal:

```
from haven import load_xspress

# Create a Xspress3-based fluorescence detector
det = load_xspress(name="vortex_me4",
                   prefix="20xmap4b",
                   num_elements=4)

# Mark the 3rd element, 2nd ROI (0-indexed)
det.mcas.mca2.rois.roi1.use.set(1)
```

Behind the scenes, to track the state of `use` we add a "~" to the start of the value in the `label()` signal if `use()` is false.

Marking multiple ROIs on multiple elements is possible using the following methods on the `XRFMixin` object:

- `enable_rois()`
- `disable_rois()`

These methods accepts an optional sequence of integers for the indices of the elements or ROIs to enable/disable. If not ROIs or elements are specified, the methods will operate on all ROIs or elements (e.g. `det.disables_rois()` will disable all ROIs on all elements.

```
from haven import load_xspress

# Create a Xspress3-based fluorescence detector
det = load_xspress(name="vortex_me4",
                   prefix="20xmap4b",
                   num_elements=4)

# Mark all ROIs on the third and fifth elements
det.enable_rois(elements=[2, 4])

# Unmark the first, eight, and fifteeth elements
det.enable_rois(rois=[0, 7, 14])
```

(continues on next page)

(continued from previous page)

```
# Unmark the third ROI on the second element
det.enable_rois(rois=[2], elements=[1])
```

2.6.3 Xspress 3

Support for Quantum Detectors' Xspress3 Family of detectors is provided by the `Xspress3Detector` base class. The EPICS support for Xspress3 detectors is based on the EPICS area detector module, and so the `Xspress3Detector` is a customized `ophyd.DetectorBase`.

2.6.4 XIA DXP (XMAP)

DXP (XMAP, Mercury, Saturn) electronics use the bluesky multi-channel analyzer (MCA) device, packaged in Haven as the `DxpDetector` class.

The DXP electronics are **not yet compatible** with *fly-scanning*. The `DxpDetector` does implement the `kickoff()` and `complete()` methods, but does not yet handle data collection. This is because the data are reported as a byte stream that must first be decoded. The DXP manual describes the structure of this byte-stream, so in principle it is possible to parse this in the `collect()` method.

2.7 Fly Scanning

Table of Contents

Fly scanning is when detectors take measuments from a sample while in motion. Creating a range of measurements based on user specified points. This method is generally faster than traditional step scanning.

Flyscanning with Bluesky follows a general three method process

- **Kickoff:** Initializes flyable Ophyd devices to set themselves up and start scanning
- **Complete:** Continously checks whether flight is occuring until it is finished
- **Collect:** Retrieves data from fly scan as proto-events

Most of the work that is done for fly scanning is done with Ophyd. Bluesky's way of fly scanning requires the Ophyd flyer device to have the `kickoff()`, `complete()`, `collect()`, and `collect_describe()` methods. Any calculation or configuration for fly scanning is done inside the Ophyd device.

2.7.1 Plans for Fly-Scanning

Haven provides several fly-scanning plans. Each one assumes that flyers implement Ophyd's `FlyerInterface`. Flyer's must also have component signals for defining the parameters of the fly scan. These signals do not need to have EPICS PVs; they can just be regular `Signal` components:

- **start_position:** center of the first bin to be measured, in motor engineering units
- **end_position:** center of the last bin to be measured, in motor engineering units
- **step_size:** width of each bin, in engineering units

fly_scan()

Haven's `fly_scan()` mimics the Bluesky `scan()` plan, except that it only accepts one motor and accompanying arguments. Both *detectors* and *motor* must implement Ophyd's `FlyerInterface`. Notice that `dwell_time` is set separately.

```
import bluesky.plan_stubs as bps
import haven
haven.load_instrument()
RE = haven.run_engine()
# Prepare devices
aerotech = haven.registry.find("aerotech")
ion_chambers = haven.registry.findall("ion_chambers")
RE(bps.mv(aerotech.horiz.dwell_time, 0.2))
# Execute the fly scan
plan = haven.fly_scan(ion_chambers, aerotech.horiz, -1000, 1000, num=101)
RE(plan, sample_name="...", purpose="...")
```

This plan only works for one flyer motor since flying two motors from Bluesky does not ensure consistent timing between the flyers. If multiple motors should be flown following the inner_product pattern, they should be wrapped in a new Flyer object that can coordinate both motor trajectories.

grid_fly_scan()

Haven's `grid_fly_scan()` provides an N-dimension scan over all combinations of multiple axes, mimicing Bluesky's `grid_scan()` plan. The first motor listed will be the slow scanning axis, and the last motor listed will be the flyer. Each motor must have an accompanying *start*, *stop*, and *num* arguments:

```
from bluesky import plans as bp, plan_stubs as bps
import haven

# (start, stop, num)
fly_params = (-100, 100, 21)
step_params = (-100, 100, 5)
dwell_time = 0.1

haven.load_instrument()

# Find the devices
ion_chambers = list(haven.registry.findall("ion_chambers"))
aerotech = haven.registry.find("aerotech")
# Create the run engine
```

(continues on next page)

(continued from previous page)

```
RE = haven.run_engine()
# Set the dwell time per pixel separately
RE(bps.mv(aerotech.horiz.dwell_time, dwell_time))
# Set up the plan
plan = haven.grid_fly_scan(ion_chambers,
                           aerotech.vert, *step_params,
                           aerotech.horiz, *fly_params,
                           snake_axes=True)
# Run the plan
RE(plan, purpose="testing fly scanning", sample="None")
```

Note: The flyer's `dwell_time` component is set outside of `grid_fly_scan()`. This is in keeping with Bluesky's approach on setting acquisition times, where each device has its own concept of acquisition time and so these need to be explicitly set as determined by the hardware.

2.7.2 Aerotech-Stage

The Aerotech stage has a number of axes, for example, `.horiz` and `.vert`. Each is a sub-class of `EpicsMotor`, adding the `FlyerInterface`. Each of these axes can be used as a flyer in the *plans for fly-scanning*.

Position-Synchronized Output (PSO)

The Ensemble controller can be configured to emit voltage pulses at fixed distance intervals. These position-synchronized output (PSO) pulses are used to trigger hardware to begin a new bin of measurements. The Ophyd flyer device sends comands to the ensemble controller to configure its settings. PSO pulses are sent in the form of a 10us on pulse. These pulses are then set to only happen every multiple integer of encoder step counts, corresponding to the Flyer device's `step_size` signal. When possible, the pulses are set to only ocur within the range of scanning.

Fig. 1: Diagram of PSO pulse timing. Encoder counts are an integer number of the smallest unit the controller can measure (e.g. nanometers). The distance from one pulse to the next equates to new bin on the scaler. Encoder window gives a range outside of which PSO pulses will be suppressed. Bottom line shows relative positions of key calculated and supplied parameters.

While the scaler can use these raw pules to create a bin, other detectors have other requirements. A DG645 delay generator is used to transform the pulses to match the various detectors. The trigger signal going to the scaler also goes through the delay generator, but the length of the delay matches the duration of the PSO pulse, so effectively output *AB* from the delay generator repeats the PSO pulses.

Fig. 2: Control flow diagram of how hardware is connected for fly scanning. The *trigger* output mimics the trigger input on the DG645 delay generator, while the length of the delay for the falling edge of the *gate* signal is based on the dwell time of the scan.

Calculated Components Before Scan

The aerotech flyer calculates the following components: slew speed, a taxi start and end position, a PSO start and end position, the window start and end in encoder counts, and the step size in encoder count.

Because step size and dwell time are input parameters, that means points must be captured while the stage moves at a constant velocity otherwise the measurments will have distorted lengths.

The Taxi start and end are the physical start and end positons of the sample stage. This is to allow the stage to accelerate to target velocity needed during scan.

The encoder window start/end is set to create a range for pulses during the scan. As well as the encoder step size which tells the PSO when to send pulses.

The PSO start/end determines the start of the first measument and the end of the last.

An array of PSO positions is also created to provide the location of each measured point.

Physical Fly scan process

1. Moves to PSO start
2. Arms PSO and starts encoder count
3. Moves to taxi start
4. Begins accelerating until reaching speed at PSO start and starts flying
5. PSO triggers detectors to take measurments until reaching a step
6. Continues flight taking measurments until reaching the end of the last measument at PSO end
7. Finally comes to a stop at taxi end after deccelerating

2.7.3 Notes

If a scan crashes the velocity will need to be changed back to its previous value in the setup caQtDM, otherwise the velocity will likely be very slow.

2.8 Hardware Triggering

For simple devices, it is enough to let bluesky and ophyd handle triggering the detector. In our case, though, many detectors are to be triggered simultaneously using one piece of hardware.

An example is **using the scaler to trigger multiple pieces of hardware**. The SIS3820 multi-channel scaler can measure multiple channels of input with one trigger. If each detector is an ophyd `Device` object, then running a bluesky plan with multiple of these devices on the same scaler will result in the scaler being triggered multiple times (once for each device in the plan).

Additionally, the scaler presents the counting gate on one of its control output lines. This can be fed into the Xspress3 electronics that power many of our Vortex detectors. Bluesky by default will try to trigger the Xspress3 directly. The `Device` definition for the Vortex detector could trigger the scaler itself, but this creates yet another trigger signal to the scaler, as described above.

The **solution** is to use the `ScalerTriggered` mixin class. This adds a *scaler_prefix* argument to `__init__` that expects a channel access PV path and points to the scaler that should be used to trigger this device. If multiple instances of `ScalerTriggered` with the same *scaler_prefix* are present in a bluesky plan, then the scaler is only triggered once for all the devices.

scaler_prefix can be omitted, in which case the *prefix* argument will be used for the scaler prefix.

```
from haven.instrument.scaler_triggered import ScalerTriggered
from ophyd import Device

class VortexDetector(ScalerTriggered, Device):
    ...

vortex = VortexDetector(prefix="vortex1ioc:vortex", scaler_prefix="25idcVME:scaler1")
```

2.9 Instrument Registry for Looking Up Components

The **instrument registry** in Haven provides a way to keep track of the devices (including components, motors, signals, etc.) that have been defined across the package. In order for the registry to know of a device, that device must first be registered. Unless you are defining your own devices or components, this will have already been done.

It is a goal of this project that **executing simple scans will not require you to know about or interact directly with the registry.** However, more advanced scans, like using area detectors from the command line, may require you to look up devices in the registry prior to building the scan.

This documentation is provided primarily for developers who are planning to register their own devices and components.

2.9.1 Looking Up Registered Devices/Components

In many cases, Haven will look up devices behind the scenes when executing a plan. However, it is possible to look up devices directly using the registry.

The registry uses the built-in concept of device labels in Ophyd. The registry's `find()` and `findall()` methods allows devices to be looked up by label or device name. For example, assuming four devices exist with the label "ion_chamber", then these devices can be retrieved using the registry:

```
from haven import registry

ion_chambers = registry.find(label="ion_chambers")
assert len(ion_chambers) == 4
```

Many plans in Haven accept lists of detectors and positioners. In some cases, it is possible to pass a string as these parameters as well, in which case the plan will assume that the string is a device name or label and find all registered devices that match. The following will execute the `energy_scan()` plan using any device initialized with `labels={"ion_chambers"}` and known to the registry.

```
from haven import energy_scan

RE(energy_scan(..., detectors="ion_chambers"))
```

2.9.2 Looking Up Sub-Components by Dot-Notation

For simple devices, the full name of the sub-component should be enough to retrieve the device. For example, to find the signal *preset_time* on the device named "vortex_me4", the following may work fine:

```
preset_time = haven.registry.find("vortex_me4_preset_time")
```

However, **if the component is lazy** and has not been accessed prior to being registered, then **it will not be available in the registry**. Sub-components can instead be accessed by dot notation. Unlike the full device name, dot-notation names only resolve when the component is requested from the registry, at which point the lazy components can be accessed.

For example, area detectors use many lazy components. If `sim_det` is an area detector with a camera component `sim_det.cam`, then the name of the gain channel is "sim_det_cam_gain", however this is a lazy component so is not available. Instead, retrieving the device by `haven.registry.find("sim_det.cam.gain")` will first find the area detector ("sim_det"), then access the *cam* attribute, and then cam's *gain* attribute. This has the side-effect of instantiating the lazy components.

2.9.3 Registering Individual Devices

Before looking up a device in the registry, it is necessary to inform the registry about the device. The simplest way to do this is using the `register()` method on the registry.

```
from ophyd import Device
from haven import registry

# Create the device instance
I0 = Device("PV_PREFIX", name="I0", labels={"ion_chamber"})
# Register the device with the registry
registry.register(I0)

# Or more concisely in 1 line
It = registry.register(Device("PV_PREFIX", name="It", labels={"ion_chamber"}))
```

2.9.4 Registering Device Classes

If you are creating many instances of a custom Device subclass, registering each instance individually can be repetitive. Haven allows you to modify the class itself so that each instance is automatically registered. This is accomplished using the `register()` method as a decorator on the class:

```
from ophyd import Device
from haven import registry

@registry.register
class IonChamber(Device):
    ...

I0 = IonChamber(..., labels={"ion_chamber"})
```

This is equivalent to the examples for registering individual devices above.

2.9.5 Creating Your Own Registry

There is nothing special about `haven.instrument.instrument_registry.registry`; it is simply an instance of `haven.instrument.instrument_registry.InstrumentRegistry` created during module import as a default. Most of the devices and components defined in Haven register themselves with this default registry. However, there's nothing to prevent you from creating your own registry:

```
from haven import InstrumentRegistry
from ophyd import Device

# Create an empty registry
my_registry = InstrumentRegistry()

# Create a new registered device object
my_device = my_registry.register(Device("PV_PREFIX", name="My Device", labels={"custom"}
↪))

# Now look for this device in the registry
my_devices = my_registry.find(label="custom")
```

2.9.6 Design Defense

This pattern touches on behavior already present in bluesky and apstools. However, there are some quirks that make these implementations unsuitable for use in Haven.

Bluesky provides the `%wa` IPython magic to list devices (apstools has a similar `listobjects()` function). While convenient when working in an IPython environment, this comes with a number of drawbacks for Haven. First, `%wa` only knows about devices listed in the local context of the IPython interpreter. If a device is defined in the file *devices.py*, the method of importing will determine whether the device is visible or not:

Listing 3: devices.py

```
from ophyd import Device

I0 = Device("PV_PREFIX", name="I0", labels={"ion_chamber"})
```

Listing 4: IPython shell

```
>>> import devices
>>> print(devices.I0)
>>> %wa  # This will not include I0
>>> from devices import I0
>>> print(I0)
>>> %wa  # Now I0 is included
```

This detail makes it impossible to run plans without knowing about all the devices and importing them individually, or else using star imports (e.g. `from devices import *`) which make tracing imports difficult and leads to cluttered namespaces.

Furthermore, this approach is tightly coupled to IPython, since it relies on the IPython shell's namespace to find devices. The above approach is not possible with vanilla CPython.

It may be possible to use `locals()` instead of the IPython shell namespace, solving the reliance on IPython. This still leaves the issue of only having access to devices imported directly into the shell's namespace, however. This could be solved by recursively descending into imported modules looking for devices. Here, PEP 20 steers us towards the

registry-based solution, where we must explicitely define a device as being included in the registry ("explicit is better than implicit").

2.10 Motor Positions

Haven is able to save the positions of one of more motors in a database; the saved positions can then be recalled later. The following functions are related to motor positions:

- `save_motor_position()`
- `list_motor_positions()`
- `recall_motor_position()`

Contents

2.10.1 Saving a Motor Position

To save the position of one or more motors, call `save_motor_position()` with the motors to be saved as arguments. These arguments can either be the name of a previously instantiated `ophyd.Device` object, or the `Device` itself. A keyword-only *name* argument is also necessary, which should be a short, human-readable description of the motor position.

```
import haven
# An example of using the motor names to save the position
uid = haven.save_motor_position("Aerotech_vert", "Aerotech_horiz", name="CuO A")
```

```
import ophyd
import haven
# An example of using the ophyd Devices to save the position
aerotech_vert = ophyd.EpicsMotor("25idd:m1")
aerotech_horiz = ophyd.EpicsMotor("25idd:m2")
uid = haven.save_motor_position(aerotech_vert, aerotech_horiz, name="CuO A")
```

`save_motor_position()` returns the database ID of the document that was created. This **ID is the best way to retrieve a motor position** from the database later, though it can be also be retrieved using the *name* argument provided it is unique.

Saving All Motor Positions

It may be convenient to save all motor positions to the database as a sort of checkpoint before performing some non-routine operation. This can be done with the following line. Future work will provide a shorted version. **Remember to call `load_instrument()` first**.

```
haven.save_motor_position(*haven.registry.findall(label="motors"), name="checkpoint␣
↪before replacing monochromator")
```

2.10.2 Viewing Saved Motor Positions

The function `list_motor_positions()` will print out a list of all the saved motor positions. This list also contains the database ID for each position, in case that information was not retained when saving the motor position originally.

2.10.3 Recalling a Saved Motor Position

The beamline can be set back a previously saved motor position using the `haven.motor_position.recall_motor_position()` function. **This function is a bluesky-style plan**, and so the plan **must be passed to a RunEngine** to be effective.

The saved motor position can be retrieved using either the ID generated when the position was saved (the *uid* argument), or by the *name* argument that was chosen when the position was saved. **If the *name* is not unique**, no guarantee is made regarding which motor position is restored.

```
import haven
RE = haven.RunEngine()

# Save the motor position
uid = haven.save_motor_position("Aerotech_vert", name="start position")

# Restore the motor position
plan = haven.recall_motor_position(uid=uid)
RE(plan)
```

2.10.4 The MotorPosition Data Model

`haven.motor_position.MotorPosition` is a pydantic model that represents a set of motor positions in the database. Any attribute that has a type definition (e.g. `offset: float = None`) is a data attribute and can be saved to the database.

To **add a new database value**, add the appropriate attribute to the pydantic model, and modify the `save()` and `load()` methods to accomodate the new database value.

2.11 Saving Data to XDI Files

Note: This page is intended for beamline staff. If you are a user at a beamline using Haven, this is most likely already set up for you.

XAFS Data Interchange (XDI) is a standard file format for storing data from individual XAFS scans in a plain-text file. Currently, Haven supports **automatic saving of energy scans** using either the `energy_scan()` or `xafs_scan()` functions. The filename used for saving will be generated from metadata. For more refined control, see below for how to create `XDIWriter` objects, or even creating a customized subclass of `XDIWriter`.

The XDI file is opened at the start of the scan, and **data are written in real time** during data acquisition, so aborted plans will still have data saved. Halted plans will still have data saved, but **the file may remain open** with write intent until the python interpreter running Haven is closed. This was a deliberate design choice to ensure the XDI writer keeps an exclusive lock on the file during execution of the plan.

2.11.1 Using the XDIWriter

If you want to save the XDI file to specific place or pass in other arguments, you can create your own instance of the `XDIWriter` class. The first argument to `XDIWriter()` should be either a file name, a `pathlib.Path` object, or an open file like those return by python's built-in `open()`. The following 3 invocations are all valid:

```
from haven import XDIWriter
from pathlib import Path

# Provide a regular string...
writer = XDIWriter("/path/to/my/xafs_data.xdi")

# ...or provide a Path object...
root = Path("/path/to/my/")
writer = XDIWriter(root / "xafs_data.xdi")
```

The *filename* can contain placeholders that will be filled in once the plan starts. This works similarly to python's format string syntax. For example:

```
from haven import XDIWriter

plan = energy_scan(..., E0="Ni_K", md=dict(sample_name="nickel oxide"))
writer_callback = XDIWriter(fd="./{year}{month}{day}_{sample_name}_{edge}.xdi")
RE(plan, writer)
```

Assuming the date is 2022-08-19, then the filename will become "20220819_nickel-oxide_Ni_K.xdi".

2.11.2 Custom Subclasses of XDIWriter

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search