Tools for Experiments Manual

Wolfgang Pfaff, Marcos Frenkel

Feb 10, 2023

CONTENTS:

Tools for experiments is a collection of Python packages that are primarily designed for running and analyzing experiments in physics laboratories. The software is meant to provide some additional convenience and capability on top of libraries that provide access to measurement hardware, such as qcodes. While the workflow in mind is electronic measurements of quantum devices (such as superconducting qubits or semiconductor quantum dots) it may well be useful for a much broader range of experiment needs.

Tools for experiments currently consists of the packages listed below. Each tool can be used independently (but they do play well together!).

Todo: we should add a few screenshots or snippets that illustrate how some of the tools work.

CHAPTER

ONE

PLOTTR

Plottr is a GUI tool for inspecting and monitoring data. The primary use case for it is inspecting measurement data (incl live-plotting while data is being acquired). Plottr allows easy graphical inspection of multidimensional data. Plotting can be supplemented with custom data analysis and processing that can be integrated into the GUI.

CHAPTER

TWO

INSTRUMENTSERVER

instrumentserver is a program that runs qcodes instruments in a dedicated server program such that client processes can access them. This allows, for example, accessing the same instrument from multiple processes

CHAPTER

THREE

LABCORE

This package contains a set of convenience tools that make setting up measurements easier.

3.1 Plottr

This is the main documentation for the data analysis tool *plottr*. *plottr* is written in python (version 3.8+) and has been tested to work well on Windows, Linux, and MacOS. This documentation is still very much ongoing work in progress, but should (hopefully!) already give an overview of what it's about, and how to use it.

The aim of *plottr* is to provide a simple but powerful graphical tool that allows efficient inspection of measurement data as frequently found in experimental physics labs (but it is in no way confined to that use). In particular, it allows to define analysis flows that process data (which could come from a file, or some other source) to very easily produce plots that give insight into the data. The goal is not to produce publication-level figures, but rather to be able to quickly 'dissect' or analyze complicated and often multi-dimensional measurement data to determine the best next course of action. For some basic examples, please see *the basic usage page*.

In addition, *plottr* is written explicitly with the goal in mind to be extendable. It is possible to define custom analysis flows that the users can implement themselves. Any kind of data manipulation or analysis (from simple things such as subtracting offsets, to more complex things like automatic fitting) can be added by the user.

Plottr is usable and pretty stable at this moment. However, we're still working on quite a few features, so new things are being improved and changed a lot. Please check back regularly to see what's new!

3.1.1 Basic usage

Installation

There are multiple options:

Recent stable versions can be installed either with pip (`$ pip install plottr`) or conda, using the conda-forge channel (`$ conda install -c conda-forge plottr`).

If you want to play with the most recent versions, or contribute to the development, it makes sense to install from github directly. In that case, clone the github repo, and install into the desired environment using the editable pip install.

Essential tools for inspecting data

There are a few different ways of easy data inspection using tools that come predefined with the basic plottr installation.

- interactive use from IPython
- loading data from a *QCoDeS* database
- loading data from HDF5 files

In the following we briefly introduce all of these. We will use the *autoplot* app that comes with plottr by default.

Note: The examples below are also included in the notebook `doc/examples/Plottr quickstart.ipynb` that comes with the plottr repository.

Interactive use

The easiest way to inspect data with plottr via the *autoplot app* is to use the `plottr.apps.autoplot.autoplot()` function from IPython or Jupyter.

Loading QCoDeS data

TBD.

Loading data from HDF5

To easily load *DataDict* from an HDF5 file we can use the function *datadict_from_hdf5()*. We can also store a *DataDict* with *datadict_to_hdf5()*.

For more information on DataDicts and how to work with them please see *Data Formats*

Live plotting measurement data

The *autoplot app* is able to live plot measurement by changing the "Refresh interval (s)" Option in the top toolbar. For a live monitoring of the entire data directory we can use the *monitr app*. We can start this app by running the following command in a terminal:

```
$ plottr-monitr <general-data-path>
```

All folders containing data will show up in the app and we can open an *autoplot app* for any already or incoming HDF5 data file. For more on how to use the app please see *Plottr Apps*

3.1.2 Data formats

In-memory data

Basic Concept

The main format we're using within plottr is the `DataDict`. While most of the actual numeric data will typically live in numpy arrays (or lists, or similar), they don't typically capture easily arbitrary metadata and relationships between arrays. Say, for example, we have some data `z` that depends on two other variables, `x` and `y`. This information has be stored somewhere, and numpy doesn't offer readily a solution here. There are various extensions, for example xarray or the MetaArray class. Those however typically have a grid format in mind, which we do not want to impose. Instead, we use a wrapper around the python dictionary that contains all the required meta information to infer the relevant relationships, and that uses numpy arrays internally to store the numeric data. Additionally we can store any other arbitrary meta data.

A DataDict container (a *dataset*) can contain multiple *data fields* (or variables), that have values and can contain their own meta information. Importantly, we distinct between independent fields (the *axes*) and dependent fields (the *data*).

Despite the naming, *axes* is not meant to imply that the *data* have to have a certain shape (but the degree to which this is true depends on the class used). A list of classes for different shapes of data can be found below.

The basic structure of data conceptually looks like this (we inherit from *dict*):

```
{
    'data_1' : {
        'axes' : ['ax1', 'ax2'],
        'unit' : 'some unit',
        'values' : [ ... ],
        '__meta__' : 'This is very important data',
        ...
    },
    'ax1' : {
        'axes' : [],
        'unit' : 'some other unit',
        'values' : [ ... ],
        ...,
    },
    'ax2' : {
        'axes' : [],
        'unit' : 'a third unit',
        'values' : [ ... ],
        ...,
    },
    '__globalmeta__' : 'some information about this data set',
    '__moremeta__' : 1234,
    ...
}
```

In this case we have one dependent variable, `data_1`, that depends on two axes, `ax1` and `ax2`. This concept is restricted only in the following way:

- A dependent can depend on any number of independents.
- An independent cannot depend on other fields itself.
- Any field that does not depend on another, is treated as an axis.

Note that meta information is contained in entries whose keys start and end with double underscores. Both the DataDict itself, as well as each field can contain meta information.

In the most basic implementation, the only restriction on the data values is that they need to be contained in a sequence (typically as list, or numpy array), and that the length of all values in the data set (the number of *records*) must be equal. Note that this does not preclude nested sequences!

Relevant data classes

DataDictBase
: The main base class. Only checks for correct dependencies. Any requirements on data structure is left to the inheriting classes. The class contains methods for easy access to data and metadata.

DataDict
: The only requirement for valid data is that the number of records is the same for all data fields. Contains some tools for expansion of data.

MeshgridDataDict
: For data that lives on a grid (not necessarily regular).

DataDict

Note: Because DataDicts are python dictionaries , we highly recommend becoming familiar with them before utilizing DataDicts.

Basic Use

We can start by creating an empty DataDict like any other python object:

```
>>> data_dict = DataDict()
>>> data_dict
{}
```

We can create the structure of the data_dict by creating dictionary items and populating them like a normal python dictionary:

```
>>> data_dict['x'] = dict(unit='m')
>>> data_dict
{'x': {'unit': 'm'}}
```

We can also start by creating a DataDict that has the structure of the data we are going to record:

```
>>> data_dict = DataDict(x=dict(unit='m'), y = dict(unit='m'), z = dict(axes=['x', 'y']))
>>> data_dict
{'x': {'unit': 'm'}, 'y': {'unit': 'm'}, 'z': {'axes': ['x', 'y']}}
```

The DataDict that we just created contains no data yet, only the structure and relationship of the data fields. We have also specified the unit of `x` and `y` and which variables are independent variables (`x`, `y`), or how we will call them from now on, `axes` and dependent variables (`z`), or, `dependents`.

Structure

From the basic and empty DataDict we can already start to inspect its structure. To see the entire structure of a DataDict we can use the *structure* method:

```
>>> data_dict = DataDict(x=dict(unit='m'), y = dict(unit='m'), z = dict(axes=['x', 'y']))
>>> data_dict.structure()
{'x': {'unit': 'm', 'axes': [], 'label': ''},
 'y': {'unit': 'm', 'axes': [], 'label': ''},
 'z': {'axes': ['x', 'y'], 'unit': '', 'label': ''}}
```

We can check for specific things inside the DataDict. We can look at the axes:

```
>>> data_dict.axes()
['x', 'y']
```

We can look at all the dependents:

```
>>> data_dict.dependents()
['z']
```

We can also see the shape of a DataDict by using the *shapes* method:

```
>>> data_dict.shapes()
{'x': (0,), 'y': (0,), 'z': (0,)}
```

Populating the DataDict

One of the only "restrictions" that DataDict implements is that every data field must have the same number of records (items). However, restrictions is in quotes because there is nothing that is stopping you from having different data fields have different number of records, this will only make the DataDict invalid. We will explore what his means later.

There are 2 different ways of safely populating a DataDict, adding data to it or appending 2 different DataDict to each other.

Note: You can always manually update the item `values` any data field like any other item of a python dictionary, however, populating the DataDict this way can result in an invalid DataDict if you are not being careful. Both population methods presented below contains checks to make sure that the new data being added will not create an invalid DataDict.

We can add data to an existing DataDict with the *add_data* method:

```
>>> data_dict = DataDict(x=dict(unit='m'), y = dict(unit='m'), z = dict(axes=['x', 'y']))
>>> data_dict.add_data(x=[0,1,2], y=[0,1,2], z=[0,1,4])
>>> data_dict
{'x': {'unit': 'm', 'axes': [], 'label': '', 'values': array([0, 1, 2])},
 'y': {'unit': 'm', 'axes': [], 'label': '', 'values': array([0, 1, 2])},
 'z': {'axes': ['x', 'y'],  'unit': '',  'label': '',  'values': array([0, 1, 4])}}
```

We now have a populated DataDict. It is important to notice that this method will also add any of the missing special keys that a data field doesn't have (*values*, *axes*, *unit*, and *label*). Populating the DataDict with this method will also ensure that every item has the same number of records and the correct shape, either by adding `nan` to the other data fields or by nesting the data arrays so that the outer most dimension of every data field has the same number of records.

We can see this in action if we add a single record to a data field with items but no the rest:

```
>>> data_dict.add_data(x=[9])
>>> data_dict
{'x': {'unit': 'm', 'axes': [], 'label': '', 'values': array([0, 1, 2, 9])},
 'y': {'unit': 'm', 'axes': [], 'label': '', 'values': array([ 0.,  1.,  2., nan])},
 'z': {'axes': ['x', 'y'], 'unit': '', 'label': '', 'values': array([ 0.,  1.,  4.,␣
↪nan])}}
```

As we can see, both `y` and `z` have an extra `nan` record in them. We can observe the change of dimension if we do not add the same number of records to all data fields:

```
>>> data_dict = DataDict(x=dict(unit='m'), y = dict(unit='m'), z = dict(axes=['x', 'y']))
>>> data_dict.add_data(x=[0,1,2], y=[0,1,2],z=[0])
>>> data_dict
{'x': {'unit': 'm', 'axes': [], 'label': '', 'values': array([[0, 1, 2]])},
 'y': {'unit': 'm', 'axes': [], 'label': '', 'values': array([[0, 1, 2]])},
 'z': {'axes': ['x', 'y'], 'unit': '', 'label': '', 'values': array([0])}}
```

If we want to expand our DataDict by appending another one, we need to make sure that both of our DataDicts have the same inner structure. We can check that by utilizing the static method *same_structure*:

```
>>> data_dict_1 = DataDict(x=dict(unit='m'), y=dict(unit='m'), z=dict(axes=['x','y']))
>>> data_dict_2 = DataDict(x=dict(unit='m'), y=dict(unit='m'), z=dict(axes=['x','y']))
>>> data_dict_1.add_data(x=[0,1,2], y=[0,1,2], z=[0,1,4])
>>> data_dict_2.add_data(x=[3,4], y=[3,4], z=[9,16])
>>> DataDict.same_structure(data_dict_1, data_dict_2)
True
```

Note: Make sure that both DataDicts have the exact same structure. This means that every item of every data field that appears when using the method *same_structure* (*unit*, *axes*, and *label*) are identical to one another, except for *values*. Any slight difference will make this method fail due to conflicting structures.

The *append* method will do this check before appending the 2 DataDict, and will only append them if the check returns `True`. Once we know that the structure is the same we can append them:

```
>>> data_dict_1.append(data_dict_2)
>>> data_dict_1
{'x': {'unit': 'm', 'axes': [], 'label': '', 'values': array([0, 1, 2, 3, 4])},
 'y': {'unit': 'm', 'axes': [], 'label': '', 'values': array([0, 1, 2, 3, 4])},
 'z': {'axes': ['x', 'y'], 'unit': '', 'label': '', 'values': array([ 0,  1,  4,  9,␣
↪16])}}
```

Meta Data

One of the advantages DataDicts have over regular python dictionaries is their ability to contain meta data. Meta data can be added to the entire DataDict or to individual data fields. Any object inside a `DataDict` whose key starts and ends with 2 underscores is considered meta data.

We can simply add meta data manually by adding an item with the proper notation:

```
>>> data_dict['__metadata__'] = 'important meta data'
```

Or we can use the *add_meta* method:

```
>>> data_dict.add_meta('sample_temperature', '10mK')
>>> data_dict
{'x': {'unit': 'm', 'axes': [], 'label': '', 'values': array([0, 1, 2])},
 'y': {'unit': 'm', 'axes': [], 'label': '', 'values': array([0, 1, 2])},
 'z': {'axes': ['x', 'y'], 'unit': '', 'label': '', 'values': array([0, 1, 4])},
 '__metadata__': 'important meta data',
 '__sample_temperature__': '10mK'}
```

We can also add meta data to a specific data field by passing its name as the last argument:

```
>>> data_dict.add_meta('extra_metadata', 'important meta data', 'x')
>>> data_dict
{'x': {'unit': 'm', 'axes': [], 'label': '', 'values': array([0, 1, 2]), '__extra_
↪metadata__': 'important meta data'},
 'y': {'unit': 'm', 'axes': [], 'label': '', 'values': array([0, 1, 2])},
 'z': {'axes': ['x', 'y'], 'unit': '', 'label': '', 'values': array([0, 1, 4])},
 '__metadata__': 'important meta data',
 '__sample_temperature__': '10mK'}
```

We can check if a certain meta field exists with the method *has_meta*:

```
>>> data_dict.has_meta('sample_temperature')
True
```

We can retrieve the meta data with the *meta_val* method:

```
>>> data_dict.meta_val('sample_temperature')
'10mK'
```

We can also ask for a meta value from a specific data field by passing the data field as the second argument:

```
>>> data_dict.meta_val('extra_metadata','x')
'important meta data'
```

We can delete a specific meta field by using the *delete_meta* method:

```
>>> data_dict.delete_meta('metadata')
>>> data_dict.has_meta('metadata')
False
```

This also work for meta data in data fields by passing the data field as the last argument:

```
>>> data_dict.delete_meta('extra_metadata', 'x')
>>> data_dict['x']
{'unit': 'm', 'axes': [], 'label': '', 'values': array([0, 1, 2])}
```

We can delete all the meta data present in the DataDict with the *clear_meta* method:

```
>>> data_dict.add_meta('metadata', 'important meta data')
>>> data_dict.add_meta('extra_metadata', 'important meta data', 'x')
>>> data_dict.clear_meta()
>>> data_dict
{'x': {'unit': 'm', 'axes': [], 'label': '', 'values': array([0, 1, 2])},
 'y': {'unit': 'm', 'axes': [], 'label': '', 'values': array([0, 1, 2])},
 'z': {'axes': ['x', 'y'], 'unit': '', 'label': '', 'values': array([0, 1, 4])}}
```

Note: There are 3 helper functions in the datadict module that help converting from meta data name to key. These are: *is_meta_key()*, *meta_key_to_name()* , and *meta_name_to_key()*.

Meshgrid DataDict

A dataset where the axes form a grid on which the dependent values reside.

This is a more special case than DataDict, but a very common scenario. To support flexible grids, this class requires that all axes specify values for each datapoint, rather than a single row/column/dimension.

For example, if we want to specify a 3-dimensional grid with axes x, y, z, the values of x, y, z all need to be 3-dimensional arrays; the same goes for all dependents that live on that grid. Then, say, x[i,j,k] is the x-coordinate of point i,j,k of the grid.

This implies that a MeshgridDataDict can only have a single shape, i.e., all data values share the exact same nesting structure.

For grids where the axes do not depend on each other, the correct values for the axes can be obtained from np.meshgrid (hence the name of the class).

Example: a simple uniform 3x2 grid might look like this; x and y are the coordinates of the grid, and z is a function of the two:

```
x = [[0, 0],
     [1, 1],
     [2, 2]]

y = [[0, 1],
     [0, 1],
     [0, 1]]

z = x * y =
    [[0, 0],
     [0, 1],
     [0, 2]]
```

Note: Internally we will typically assume that the nested axes are ordered from slow to fast, i.e., dimension 1 is the most outer axis, and dimension N of an N-dimensional array the most inner (i.e., the fastest changing one). This guarantees,

for example, that the default implementation of np.reshape has the expected outcome. If, for some reason, the specified axes are not in that order (e.g., we might have `z` with `axes = ['x', 'y']`, but `x` is the fast axis in the data). In such a case, the guideline is that at creation of the meshgrid, the data should be transposed such that it conforms correctly to the order as given in the `axis = [...]` specification of the data. The function *datadict_to_meshgrid* provides options for that.

This implementation of *DataDictBase* consists only of 3 extra methods:

- *MeshgridDataDict.shape*
- *MeshgridDataDict.validate*
- *MeshgridDataDict.reorder_axis*

So the only way of populating it is by manually modifying the `values` object of each data field since the tools for populating the DataDict are specific to the *DataDict* implementation.

DataDict Storage

The datadict_storage.py module offers tools to help with saving DataDicts into disk by storing them in DDH5 files (HDF5 files that contains DataDicts inside).

Description of the HDF5 storage format

We use a simple mapping from DataDict to the HDF5 file. Within the file, a single DataDict is stored in a (top-level) group of the file. The data fields are datasets within that group.

Global meta data of the DataDict are attributes of the group; field meta data are attributes of the dataset (incl., the *unit* and *axes* values). The meta data keys are given exactly like in the DataDict, i.e., includes the double underscore pre- and suffix.

For more specific information on how HDF5 works please read the following documentation

Working with DDH5 files

When we are working with data, the first thing we usually want to do is to save it in disk. We can directly save an already existing DataDict into disk by calling the function *datadict_to_hdf5*.

```
>>> data_dict = DataDict(x=dict(values=np.array([0,1,2]), axes=[], __unit__='cm'),␣
↪y=dict(values=np.array([3,4,5]), axes=['x']))
>>> data_dict
{'x': {'values': array([0, 1, 2]), 'axes': [], '__unit__': 'cm'},
 'y': {'values': array([3, 4, 5]), 'axes': ['x']}}
>>> datadict_to_hdf5(data_dict, 'folder\data.ddh5')
```

datadict_to_hdf5 will save data_dict in a file named 'data.ddh5' in whatever directory is passed to it, creating new folders if they don't already exists. The file will contain all of the data fields as well as all the metadata, with some more metadata generated to specify when the DataDict was created.

Note: Meta data is only written during initial writing of the dataset. If we're appending to existing datasets, we're not setting meta data anymore.

Warning: For this method to properly work the objects that are being saved in the `values` key of a data field must by a numpy array, or numpy array like.

Data saved on disk is useless however if we do not have a way of accessing it. To do this we use the *datadict_from_hdf5*:

```
>>> loaded_data_dict = datadict_from_hdf5('folder\data.ddh5')
>>> loaded_data_dict
{'__creation_time_sec__': 1651159636.0,
 '__creation_time_str__': '2022-04-28 10:27:16',
 'x': {'values': array([0, 1, 2]),
  'axes': [],
  '__shape__': (3,),
  '__creation_time_sec__': 1651159636.0,
  '__creation_time_str__': '2022-04-28 10:27:16',
  '__unit__': 'cm',
  'unit': '',
  'label': ''},
 'y': {'values': array([3, 4, 5]),
  'axes': ['x'],
  '__shape__': (3,),
  '__creation_time_sec__': 1651159636.0,
  '__creation_time_str__': '2022-04-28 10:27:16',
  'unit': '',
  'label': ''}}
```

We can see that the DataDict is the same one we saved earlier with the added metadata that indicates the time it was created.

By default both *datadict_to_hdf5* and *datadict_from_hdf5* save and load the datadict in the 'data' group of the DDH5. Both of these can by changed by passing another group to the argument 'groupname'. We can see this if we manually create a second group and save a new DataDict there:

```
>>> data_dict2 = DataDict(a=dict(values=np.array([0,1,2]), axes=[], __unit__='cm'),␣
↪b=dict(values=np.array([3,4,5]), axes=['a']))
>>> with h5py.File('folder\data.ddh5', 'a') as file:
>>>     file.create_group('other_data')
>>> datadict_to_hdf5(data_dict2, 'folder\data.ddh5', groupname='other_data')
```

If we then load the DDH5 file like before we only see the first DataDict:

```
>>> loaded_data_dict = datadict_from_hdf5('folder\data.ddh5', 'data')
>>> loaded_data_dict
{'__creation_time_sec__': 1651159636.0,
 '__creation_time_str__': '2022-04-28 10:27:16',
 'x': {'values': array([0, 1, 2]),
  'axes': [],
  '__shape__': (3,),
  '__creation_time_sec__': 1651159636.0,
  '__creation_time_str__': '2022-04-28 10:27:16',
  '__unit__': 'cm',
  'unit': '',
  'label': ''},
```

(continues on next page)

(continued from previous page)

```
 'y': {'values': array([3, 4, 5]),
  'axes': ['x'],
  '__shape__': (3,),
  '__creation_time_sec__': 1651159636.0,
  '__creation_time_str__': '2022-04-28 10:27:16',
  'unit': '',
  'label': ''}}
```

To see the other DataDict we can specify the group in the argument 'groupname':

```
>>> loaded_data_dict = datadict_from_hdf5('folder\data.ddh5', 'other_data')
>>> loaded_data_dict
{'a': {'values': array([0, 1, 2]),
  'axes': [],
  '__shape__': (3,),
  '__creation_time_sec__': 1651159636.0,
  '__creation_time_str__': '2022-04-28 10:27:16',
  '__unit__': 'cm',
  'unit': '',
  'label': ''},
 'b': {'values': array([3, 4, 5]),
  'axes': ['a'],
  '__shape__': (3,),
  '__creation_time_sec__': 1651159636.0,
  '__creation_time_str__': '2022-04-28 10:27:16',
  'unit': '',
  'label': ''}}
```

We can also use *all_datadicts_from_hdf5* to get a dictionary with all DataDicts in every group inside:

```
>>> all_datadicts = all_datadicts_from_hdf5('folder\data.ddh5')
>>> all_datadicts
{'data': {'__creation_time_sec__': 1651159636.0,
  '__creation_time_str__': '2022-04-28 10:27:16',
  'x': {'values': array([0, 1, 2]),
   'axes': [],
   '__shape__': (3,),
   '__creation_time_sec__': 1651159636.0,
   '__creation_time_str__': '2022-04-28 10:27:16',
   '__unit__': 'cm',
   'unit': '',
   'label': ''},
  'y': {'values': array([3, 4, 5]),
   'axes': ['x'],
   '__shape__': (3,),
   '__creation_time_sec__': 1651159636.0,
   '__creation_time_str__': '2022-04-28 10:27:16',
   'unit': '',
   'label': ''}},
 'other_data': {'a': {'values': array([0, 1, 2]),
   'axes': [],
   '__shape__': (3,),
```

(continues on next page)

(continued from previous page)

```
   '__creation_time_sec__': 1651159636.0,
   '__creation_time_str__': '2022-04-28 10:27:16',
   '__unit__': 'cm',
   'unit': '',
   'label': ''},
  'b': {'values': array([3, 4, 5]),
   'axes': ['a'],
   '__shape__': (3,),
   '__creation_time_sec__': 1651159636.0,
   '__creation_time_str__': '2022-04-28 10:27:16',
   'unit': '',
   'label': ''}}}
```

DDH5 Writer

Most times we want to be saving data to disk as soon as it is generated by an experiment (or iteration), instead of waiting to have a complete DataDict. To do this, Datadict_storage also offers a context manager with which we can safely save our incoming data.

To use it we first need to create an empty DataDict that contains the structure of how the data is going to look like:

```
>>> data_dict = DataDict(
>>> x = dict(unit='x_unit'),
>>> y = dict(unit='y_unit', axes=['x']))
```

With our created DataDict, we can start the *DDH5Writer* context manager and add data to our DataDict utilizing the *add_data*

```
>>> with DDH5Writer(datadict=data_dict, basedir='./data/', name='Test') as writer:
>>>     for x in range(10):
>>>         writer.add_data(x=x, y=x**2)
Data location:  data\2022-04-27\2022-04-27T145308_a986867c-Test\data.ddh5
```

The writer created the folder 'data' (because it did not exist before) and inside that folder, created another new folder for the current day and another new folder inside of it day folder for the the DataDict that we saved with the naming structure of `YYYY-mm-dd_THHMMSS_<ID>-<name>/<filename>.ddh5`, where name is the name parameter passed to the writer. The writer creates this structure such that when we run the writer again with new data, it will create another folder following the naming structure inside the current date folder. This way each new DataDict will be saved in the date it was generated with a time stamp in the name of the folder containing it.

Changing File Extension and Time Format

Finally, datadict_storage contains 2 module variables, 'DATAFILEXT' and 'TIMESTRFORMAT'.

'DATAFILEXT' by default is 'ddh5', and it is used to specify the extension file of all of the module saving functions. Change this variable if you want your HDF5 to have a different extension by default, instead of passing it everytime.

'TIMESTRFORMAT' specifies how the time is formated in the new metadata created when saving a DataDict. The default is: `"%Y-%m-%d %H:%M:%S"`, and it follows the structure of strftime.

Reference

DataDict

DataDictBase

The following is the base class from which both *DataDict* and *MeshgridDataDict* are inheriting.

class plottr.data.datadict.**DataDictBase**(***kw: Any*)

Simple data storage class that is based on a regular dictionary.

This base class does not make assumptions about the structure of the values. This is implemented in inheriting classes.

static to_records(***data: Any*) → Dict[str, ndarray]

Convert data to records that can be added to the `DataDict`. All data is converted to np.array, and reshaped such that the first dimension of all resulting arrays have the same length (chosen to be the smallest possible number that does not alter any shapes beyond adding a length-1 dimension as first dimension, if necessary).

If a data field is given as `None`, it will be converted to `numpy.array([numpy.nan])`.

Parameters
data – keyword arguments for each data field followed by data.

Returns
Dictionary with properly shaped data.

data_items() → Iterator[Tuple[str, Dict[str, Any]]]

Generator for data field items.

Like dict.items(), but ignores meta data.

Returns
Generator yielding first the key of the data field and second its value.

meta_items(*data: Optional[str] = None*, *clean_keys: bool = True*) → Iterator[Tuple[str, Dict[str, Any]]]

Generator for meta items.

Like dict.items(), but yields *only* meta entries. The keys returned do not contain the underscores used internally.

Parameters

- **data** – If `None` iterate over global meta data. If it's the name of a data field, iterate over the meta information of that field.
- **clean_keys** – If *True*, remove the underscore pre/suffix.

Returns
Generator yielding first the key of the data field and second its value.

data_vals(*key: str*) → ndarray

Return the data values of field `key`.

Equivalent to `DataDict['key'].values`.

Parameters
key – Name of the data field.

Returns
Values of the data field.

has_meta(*key: str*) → bool

Check whether meta field exists in the dataset.

Returns
`True` if it exists, `False` if it doesn't.

meta_val(*key: str*, *data: Optional[str] = None*) → Any

Return the value of meta field `key` (given without underscore).

Parameters

- **key** – Name of the meta field.
- **data** – `None` for global meta; name of data field for data meta.

Returns
The value of the meta information.

add_meta(*key: str*, *value: Any*, *data: Optional[str] = None*) → None

Add meta info to the dataset.

If the key already exists, meta info will be overwritten.

Parameters

- **key** – Name of the meta field (without underscores).
- **value** – Value of the meta information.
- **data** – If `None`, meta will be global; otherwise assigned to data field `data`.

delete_meta(*key: str*, *data: Optional[str] = None*) → None

Deletes specific meta data.

Parameters

- **key** – Name of the meta field to remove.
- **data** – If `None`, this affects global meta; otherwise remove from data field `data`.

clear_meta(*data: Optional[str] = None*) → None

Deletes all meta data.

Parameters
data – If not `None`, delete all meta only from specified data field `data`. Else, deletes all top-level meta, as well as meta for all data fields.

extract(*data: List[str]*, *include_meta: bool = True*, *copy: bool = True*, *sanitize: bool = True*) → T

Extract data from a dataset.

Return a new datadict with all fields specified in `data` included. Will also take any axes fields along that have not been explicitly specified. Will return empty if `data` consists of only axes fields.

Parameters

- **data** – Data field or list of data fields to be extracted.
- **include_meta** – If `True`, include the global meta data. data meta will always be included.
- **copy** – If `True`, data fields will be deep copies of the original.
- **sanitize** – If `True`, will run DataDictBase.sanitize before returning.

Returns
New DataDictBase containing only requested fields.

static same_structure(**data: T*, *check_shape: bool = False*) → bool

Check if all supplied DataDicts share the same data structure (i.e., dependents and axes).

Ignores meta data and values. Checks also for matching shapes if *check_shape* is *True*.

Parameters

- **data** – The data sets to compare.
- **check_shape** – Whether to include shape check in the comparison.

Returns

`True` if the structure matches for all, else `False`.

structure(*add_shape: bool = False*, *include_meta: bool = True*, *same_type: bool = False*) → Optional[T]

Get the structure of the DataDict.

Return the datadict without values (*value* omitted in the dict).

Parameters

- **add_shape** – Deprecated – ignored.
- **include_meta** – If *True*, include the meta information in the returned dict.
- **same_type** – If *True*, return type will be the one of the object this is called on. Else, DataDictBase.

Returns

The DataDict containing the structure only. The exact type is the same as the type of `self`.

label(*name: str*) → Optional[str]

Get the label for a data field. If no label is present returns the name of the data field as the label. If a unit is present, it will be appended at the end in brackets: "label (unit)".

Parameters

name – Name of the data field.

Returns

Labelled name.

axes_are_compatible() → bool

Check if all dependent data fields have the same axes.

This includes axes order.

Returns

`True` or `False`.

axes(*data: Optional[Union[Sequence[str], str]] = None*) → List[str]

Return a list of axes.

Parameters

data – if `None`, return all axes present in the dataset, otherwise only the axes of the dependent `data`.

Returns

The list of axes.

dependents() → List[str]

Get all dependents in the dataset.

Returns

A list of the names of dependents.

shapes() → Dict[str, Tuple[int, ...]]

Get the shapes of all data fields.

Returns
A dictionary of the form `{key : shape}`, where shape is the np.shape-tuple of the data with name `key`.

validate() → bool

Check the validity of the dataset.

Checks performed:

- All axes specified with dependents must exist as data fields.

Other tasks performed:

- `unit` keys are created if omitted.
- `label` keys are created if omitted.
- `shape` meta information is updated with the correct values (only if present already).

Returns
`True` if valid, `False` if invalid.

Raises
`ValueError` if invalid.

remove_unused_axes() → T

Removes axes not associated with dependents.

Returns
Cleaned dataset.

sanitize() → T

Clean-up tasks:

- Removes unused axes.

Returns
Sanitized dataset.

reorder_axes_indices(*name: str*, ***pos: int*) → Tuple[Tuple[int, ...], List[str]]

Get the indices that can reorder axes in a given way.

Parameters

- **name** – Name of the data field of which we want to reorder axes.
- **pos** – New axes position in the form `axis_name = new_position`. Non-specified axes positions are adjusted automatically.

Returns
The tuple of new indices, and the list of axes names in the new order.

reorder_axes(*data_names: Optional[Union[Sequence[str], str]] = None*, ***pos: int*) → T

Reorder data axes.

Parameters

- **data_names** – Data name(s) for which to reorder the axes. If None, apply to all dependents.

- **pos** – New axes position in the form `axis_name = new_position`. Non-specified axes positions are adjusted automatically.

Returns
Dataset with re-ordered axes.

copy() → T

Make a copy of the dataset.

Returns
A copy of the dataset.

astype(*dtype: dtype*) → T

Convert all data values to given dtype.

Parameters
dtype – np dtype.

Returns
Copy of the dataset, with values as given type.

mask_invalid() → T

Mask all invalid data in all values. :return: Copy of the dataset with invalid entries (nan/None) masked.

DataDict

class plottr.data.datadict.**DataDict**(***kw: Any*)

The most basic implementation of the DataDict class.

It only enforces that the number of *records* per data field must be equal for all fields. This refers to the most outer dimension in case of nested arrays.

The class further implements simple appending of datadicts through the `DataDict.append` method, as well as allowing addition of DataDict instances.

append(*newdata:* DataDict) → None

Append a datadict to this one by appending data values.

Parameters
newdata – DataDict to append.

Raises
`ValueError`, if the structures are incompatible.

add_data(***kw: Any*) → None

Add data to all values. new data must be valid in itself.

This method is useful to easily add data without needing to specify meta data or dependencies, etc.

Parameters
kw – one array per data field (none can be omitted).

nrecords() → Optional[int]

Gets the number of records in the dataset.

Returns
The number of records in the dataset.

`is_expanded`() → bool

Determine if the DataDict is expanded.

Returns
`True` if expanded. `False` if not.

`is_expandable`() → bool

Determine if the DataDict can be expanded.

Expansion flattens all nested data values to a 1D array. For doing so, we require that all data fields that have nested/inner dimensions (i.e, inside the *records* level) shape the inner shape. In other words, all data fields must be of shape (N,) or (N, (shape)), where shape is common to all that have a shape not equal to (N,).

Returns
`True` if expandable. `False` otherwise.

`expand`() → *DataDict*

Expand nested values in the data fields.

Flattens all value arrays. If nested dimensions are present, all data with non-nested dims will be repeated accordingly – each record is repeated to match the size of the nested dims.

Returns
The flattened dataset.

Raises
`ValueError` if data is not expandable.

`validate`() → bool

Check dataset validity.

Beyond the checks performed in the base class `DataDictBase`, check whether the number of records is the same for all data fields.

Returns
`True` if valid.

Raises
`ValueError` if invalid.

`sanitize`() → *DataDict*

Clean-up.

Beyond the tasks of the base class `DataDictBase`:

- remove invalid entries as far as reasonable.

Returns
sanitized DataDict.

`remove_invalid_entries`() → *DataDict*

Remove all rows that are `None` or `np.nan` in *all* dependents.

Returns
The cleaned DataDict.

Meshgrid DataDict

class plottr.data.datadict.**MeshgridDataDict**(***kw: Any*)

Implementation of DataDictBase meant to be used for when the axes form a grid on which the dependent values reside.

It enforces that all dependents have the same axes and all shapes need to be identical.

shape() → Union[None, Tuple[int, ...]]

Return the shape of the meshgrid.

Returns
The shape as tuple. `None` if no data in the set.

validate() → bool

Validation of the dataset.

Performs the following checks: * All dependents must have the same axes. * All shapes need to be identical.

Returns
`True` if valid.

Raises
`ValueError` if invalid.

reorder_axes(*data_names: Optional[Union[Sequence[str], str]] = None*, ***pos: int*) → *MeshgridDataDict*

Reorder the axes for all data.

This includes transposing the data, since we're on a grid.

Parameters
pos – New axes position in the form `axis_name = new_position`. non-specified axes positions are adjusted automatically.

Returns
Dataset with re-ordered axes.

Extra Module Functions

datadict.py :

Data classes we use throughout the plottr package, and tools to work on them.

plottr.data.datadict.**is_meta_key**(*key: str*) → bool

Checks if `key` is meta information.

Parameters
key – The `key` we are checking.

Returns
`True` if it is, `False` if it isn't.

plottr.data.datadict.**meta_key_to_name**(*key: str*) → str

Converts a meta data key to just the name. E.g: for `key`: "__meta__" returns "meta"

Parameters
key – The key that is being converted

Returns
The name of the key.

Raises
`ValueError` if the `key` is not a meta key.

`plottr.data.datadict.`**`meta_name_to_key`**(*name: str*) → str

Converts `name` into a meta data key. E.g: "meta" gets converted to "__meta__"

Parameters
`name` – The name that is being converted.

Returns
The meta data key based on `name`.

`plottr.data.datadict.`**`guess_shape_from_datadict`**(*data:* DataDict) → Dict[str, Union[None, Tuple[List[str], Tuple[int, ...]]]]

Try to guess the shape of the datadict dependents from the axes values.

Parameters
`data` – Dataset to examine.

Returns
A dictionary with the dependents as keys, and inferred shapes as values. Value is `None`, if the shape could not be inferred.

`plottr.data.datadict.`**`datadict_to_meshgrid`**(*data:* DataDict, *target_shape: Optional[Tuple[int, ...]] = None*, *inner_axis_order: Union[None, Sequence[str]] = None*, *use_existing_shape: bool = False*) → *MeshgridDataDict*

Try to make a meshgrid from a dataset.

Parameters

- **`data`** – Input DataDict.
- **`target_shape`** – Target shape. If `None` we use `guess_shape_from_datadict` to infer.
- **`inner_axis_order`** – If axes of the datadict are not specified in the 'C' order (1st the slowest, last the fastest axis) then the 'true' inner order can be specified as a list of axes names, which has to match the specified axes in all but order. The data is then transposed to conform to the specified order.

  Note: If this is given, then `target_shape` needs to be given in in the order of this inner_axis_order. The output data will keep the axis ordering specified in the *axes* property.

- **`use_existing_shape`** – if `True`, simply use the shape that the data already has. For numpy-array data, this might already be present. If `False`, flatten and reshape.

Raises
GriddingError (subclass of ValueError) if the data cannot be gridded.

Returns
The generated `MeshgridDataDict`.

`plottr.data.datadict.`**`meshgrid_to_datadict`**(*data:* MeshgridDataDict) → *DataDict*

Make a DataDict from a MeshgridDataDict by reshaping the data.

Parameters
`data` – Input `MeshgridDataDict`.

Returns
Flattened `DataDict`.

plottr.data.datadict.**combine_datadicts**(**dicts:* DataDict) → Union[*DataDictBase*, *DataDict*]

Try to make one datadict out of multiple.

Basic rules:

- We try to maintain the input type.
- Return type is 'downgraded' to DataDictBase if the contents are not compatible (i.e., different numbers of records in the inputs).

Returns
Combined data.

plottr.data.datadict.**datastructure_from_string**(*description: str*) → *DataDict*

Construct a DataDict from a string description.

Examples:

- `"data[mV](x, y)"` results in a datadict with one dependent `data` with unit `mV` and two independents, `x` and `y`, that do not have units.
- `"data_1[mV](x, y); data_2[mA](x); x[mV]; y[nT]"` results in two dependents, one of them depening on `x` and `y`, the other only on `x`. Note that `x` and `y` have units. We can (but do not have to) omit them when specifying the dependencies.
- `"data_1[mV](x[mV], y[nT]); data_2[mA](x[mV])"`. Same result as the previous example.

Rules:

We recognize descriptions of the form `field1[unit1](ax1, ax2, ...); field1[unit2](...); ...`

- Field names (like `field1` and `field2` above) have to start with a letter, and may contain word characters.
- Field descriptors consist of the name, optional unit (presence signified by square brackets), and optional dependencies (presence signified by round brackets).
- Dependencies (axes) are implicitly recognized as fields (and thus have the same naming restrictions as field names).
- Axes are separated by commas.
- Axes may have a unit when specified as dependency, but besides the name, square brackets, and commas no other characters are recognized within the round brackets that specify the dependency.
- In addition to being specified as dependency for a field, axes may be specified also as additional field without dependency, for instance to specify the unit (may simplify the string). For example, `z1[x, y]; z2[x, y]; x[V]; y[V]`.
- Units may only consist of word characters.
- Use of unexpected characters will result in the ignoring the part that contains the symbol.
- The regular expression used to find field descriptors is: `((?<=\A)|(?<=\;))[a-zA-Z]+\w*(\[\w*\])?(\(([a-zA-Z]+\w*(\[\w*\])?\,?)*\))?`

plottr.data.datadict.**str2dd**(*description: str*) → *DataDict*

shortcut to *datastructure_from_string()*.

plottr.data.datadict.**datasets_are_equal**(*a:* DataDictBase, *b:* DataDictBase, *ignore_meta: bool = False*) → bool

Check whether two datasets are equal.

Compares type, structure, and content of all fields.

Parameters

- **a** – First dataset.
- **b** – Second dataset.
- **ignore_meta** – If `True`, do not verify if metadata matches.

Returns
`True` or `False`.

DataDict Storage

plottr.data.datadict_storage

Provides file-storage tools for the DataDict class.

Note: Any function in this module that interacts with a ddh5 file, will create a lock file while it is using the file. The lock file has the following format: ~<file_name>.lock. The file lock will get deleted even if the program crashes. If the process is suddenly stopped however, we cannot guarantee that the file lock will be deleted.

`class plottr.data.datadict_storage.AppendMode(value)`

How/Whether to append data to existing data.

new = 0

Data that is additional compared to already existing data is appended.

all = 1

All data is appended to existing data.

none = 2

Data is overwritten.

`plottr.data.datadict_storage.h5ify(obj: Any) → Any`

Convert an object into something that we can assign to an HDF5 attribute.

Performs the following conversions: - list/array of strings -> numpy chararray of unicode type

Parameters
obj – Input object.

Returns
Object, converted if necessary.

`plottr.data.datadict_storage.deh5ify(obj: Any) → Any`

Convert slightly mangled types back to more handy ones.

Parameters
obj – Input object.

Returns
Object

plottr.data.datadict_storage.**set_attr**(*h5obj: Any*, *name: str*, *val: Any*) → None

Set attribute *name* of object *h5obj* to *val*

Use *h5ify()* to convert the object, then try to set the attribute to the returned value. If that does not succeed due to a HDF5 typing restriction, set the attribute to the string representation of the value.

plottr.data.datadict_storage.**add_cur_time_attr**(*h5obj: Any*, *name: str = 'creation'*, *prefix: str = '__'*, *suffix: str = '__'*) → None

Add current time information to the given HDF5 object, following the format of: <prefix><name>_time_sec<suffix>.

Parameters

- **h5obj** – The HDF5 object.
- **name** – The name of the attribute.
- **prefix** – Prefix of the attribute.
- **suffix** – Suffix of the attribute.

plottr.data.datadict_storage.**datadict_to_hdf5**(*datadict: DataDict*, *path: Union[str, Path]*, *groupname: str = 'data'*, *append_mode: AppendMode = AppendMode.new*, *file_timeout: Optional[float] = None*) → None

Write a DataDict to DDH5

Note: Meta data is only written during initial writing of the dataset. If we're appending to existing datasets, we're not setting meta data anymore.

Parameters

- **datadict** – Datadict to write to disk.
- **path** – Path of the file (extension may be omitted).
- **groupname** – Name of the top level group to store the data in.
- **append_mode** –
  - *AppendMode.none* : Delete and re-create group.
  - *AppendMode.new* : Append rows in the datadict that exceed the number of existing rows in the dataset already stored. Note: we're not checking for content, only length!
  - *AppendMode.all* : Append all data in datadict to file data sets.
- **file_timeout** – How long the function will wait for the ddh5 file to unlock. Only relevant if you are writing to a file that already exists and some other program is trying to read it at the same time. If none uses the default value from the *FileOpener*.

plottr.data.datadict_storage.**datadict_from_hdf5**(*path: Union[str, Path]*, *groupname: str = 'data'*, *startidx: Optional[int] = None*, *stopidx: Optional[int] = None*, *structure_only: bool = False*, *ignore_unequal_lengths: bool = True*, *file_timeout: Optional[float] = None*) → *DataDict*

Load a DataDict from file.

Parameters

- **path** – Full filepath without the file extension.
- **groupname** – Name of hdf5 group.

- **startidx** – Start row.
- **stopidx** – End row + 1.
- **structure_only** – If *True*, don't load the data values.
- **ignore_unequal_lengths** – If *True*, don't fail when the rows have unequal length; will return the longest consistent DataDict possible.
- **file_timeout** – How long the function will wait for the ddh5 file to unlock. If none uses the default value from the *FileOpener*.

Returns
Validated DataDict.

plottr.data.datadict_storage.**all_datadicts_from_hdf5**(*path: Union[str, Path]*, *file_timeout: Optional[float] = None*, ***kwargs: Any*) → Dict[str, Any]

Loads all the DataDicts contained on a single HDF5 file. Returns a dictionary with the group names as keys and the DataDicts as the values of that key.

Parameters

- **path** – The path of the HDF5 file.
- **file_timeout** – How long the function will wait for the ddh5 file to unlock. If none uses the default value from the *FileOpener*.

Returns
Dictionary with group names as key, and the DataDicts inside them as values.

class plottr.data.datadict_storage.**FileOpener**(*path: Union[Path, str]*, *mode: str = 'r'*, *timeout: Optional[float] = None*, *test_delay: float = 0.1*)

Context manager for opening files, creates its own file lock to indicate other programs that the file is being used. The lock file follows the following structure: "~<file_name>.lock".

Parameters

- **path** – The file path.
- **mode** – The opening file mode. Only the following modes are supported: 'r', 'w', 'w-', 'a'. Defaults to 'r'.
- **timeout** – Time, in seconds, the context manager waits for the file to unlock. Defaults to 30.
- **test_delay** – Length of time in between checks. I.e. how long the FileOpener waits to see if a file got unlocked again

class plottr.data.datadict_storage.**DDH5Loader**(*name: str*)

nodeName = 'DDH5Loader'

Name of the node. used in the flowchart node library.

uiClass

alias of DDH5LoaderWidget

useUi = True

Whether or not to automatically set up a UI widget.

process(*dataIn: Optional[DataDictBase] = None*) → Optional[Dict[str, Any]]

Process data through this node. This method is called any time the flowchart wants the node to process data. It will be called with one keyword argument corresponding to each input terminal, and must return a dict mapping the name of each output terminal to its new value.

This method is also called with a 'display' keyword argument, which indicates whether the node should update its display (if it implements any) while processing this data. This is primarily used to disable expensive display operations during batch processing.

class plottr.data.datadict_storage.**DDH5Writer**(*datadict: DataDict*, *basedir: Union[str, Path] = '.'*, *groupname: str = 'data'*, *name: Optional[str] = None*, *filename: str = 'data'*, *filepath: Optional[Union[str, Path]] = None*, *file_timeout: Optional[float] = None*)

Context manager for writing data to DDH5. Based on typical needs in taking data in an experimental physics lab.

Creates lock file when writing data.

Parameters

- **basedir** – The root directory in which data is stored. `create_file_structure()` is creating the structure inside this root and determines the file name of the data. The default structure implemented here is `<root>/YYYY-MM-DD/YYYY-mm-dd_THHMMSS_<ID>-<name>/<filename>.ddh5`, where `<ID>` is a short identifier string and `<name>` is the value of parameter *name*. To change this, re-implement *data_folder()* and/or `create_file_structure()`.
- **datadict** – Initial data object. Must contain at least the structure of the data to be able to use *add_data()* to add data.
- **groupname** – Name of the top-level group in the file container. An existing group of that name will be deleted.
- **name** – Name of this dataset. Used in path/file creation and added as meta data.
- **filename** – Filename to use. Defaults to 'data.ddh5'.
- **file_timeout** – How long the function will wait for the ddh5 file to unlock. If none uses the default value from the *FileOpener*.

data_folder() → Path

Return the folder, relative to the data root path, in which data will be saved.

Default format: `<basedir>/YYYY-MM-DD/YYYY-mm-ddTHHMMSS_<ID>-<name>`. In this implementation we use the first 8 characters of a UUID as ID.

Returns
The folder path.

data_file_path() → Path

Determine the filepath of the data file.

Returns
The filepath of the data file.

add_data(***kwargs: Any*) → None

Add data to the file (and the internal *DataDict*).

Requires one keyword argument per data field in the *DataDict*, with the key being the name, and value the data to add. It is required that all added data has the same number of 'rows', i.e., the most outer dimension has to match for data to be inserted faithfully. If some data is scalar and others are not, then the data should

be reshaped to (1,) for the scalar data, and (1, ...) for the others; in other words, an outer dimension with length 1 is added for all.

3.1.3 Plottr apps

Predefined apps

Autoplot

Monitr

Monitr is a file monitoring app that updates in real time showing every measurement in the monitoring folder. In monitr you can directly open plots to see your data, select favorite items, filter items and add images and comments to your data.

How to Open Monitr

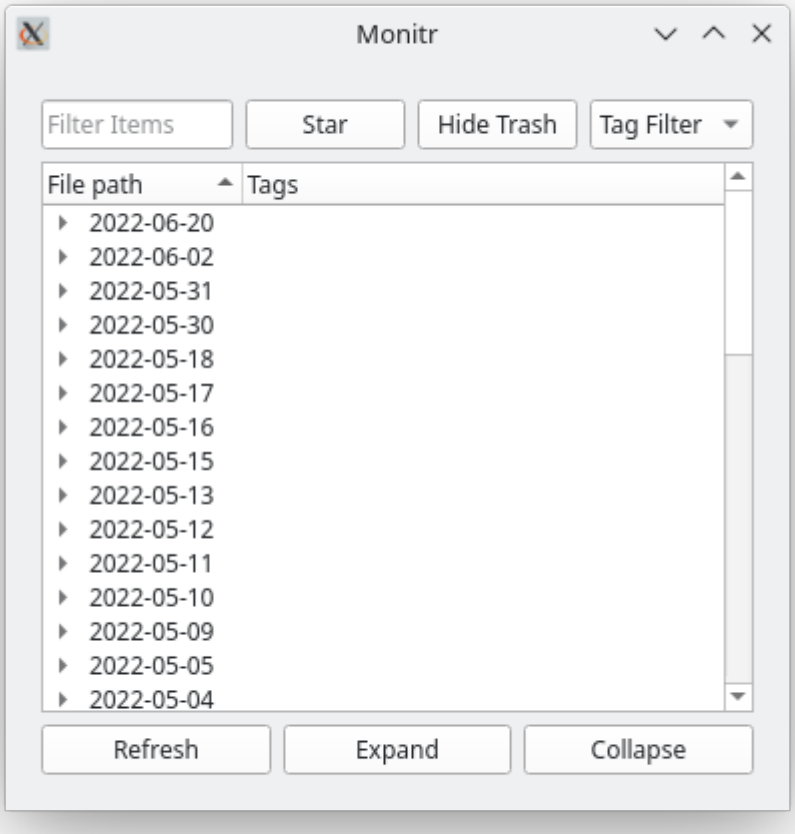
Monitr is designed to monitor all subdirectories created inside the monitoring directory. We open Monitr from a Linux terminal with the command:

```
$ plottr-monitr <monitoring-directory>
```

From a Windows terminal:

```
$ plottr-monitr.exe <monitoring-directory>
```

The command will start the file monitoring and the main window will open:



The main window will display all *Datasets*, or directories containing datasets, that lives inside the monitoring directory and will keep updating itself in real time, showing any new datasets, deleting them when the files are deleted, and changing the names of them when necessary.

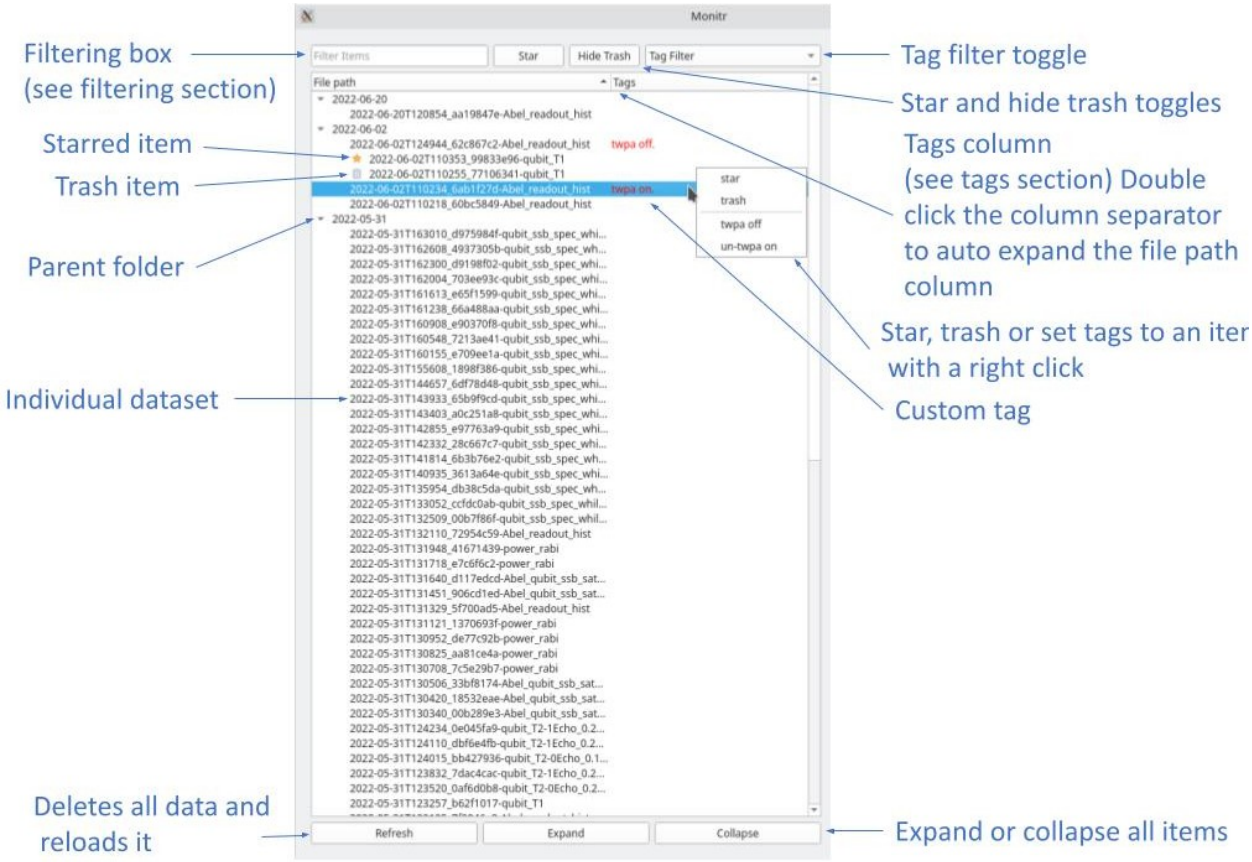
Datasets

At the moment we define a folder as a dataset if it contains either a ddh5, markdown or json file inside of it. This is defined in the `SupportedDataTypes` object. If any file directory passes a regex matching with these file types, its parent folder is considered a dataset.

Note: If a directory lives inside of the monitoring directory but does not contain any of the three file types, or it doesn't have a folder that has them, the folder will not show in the main file tree.

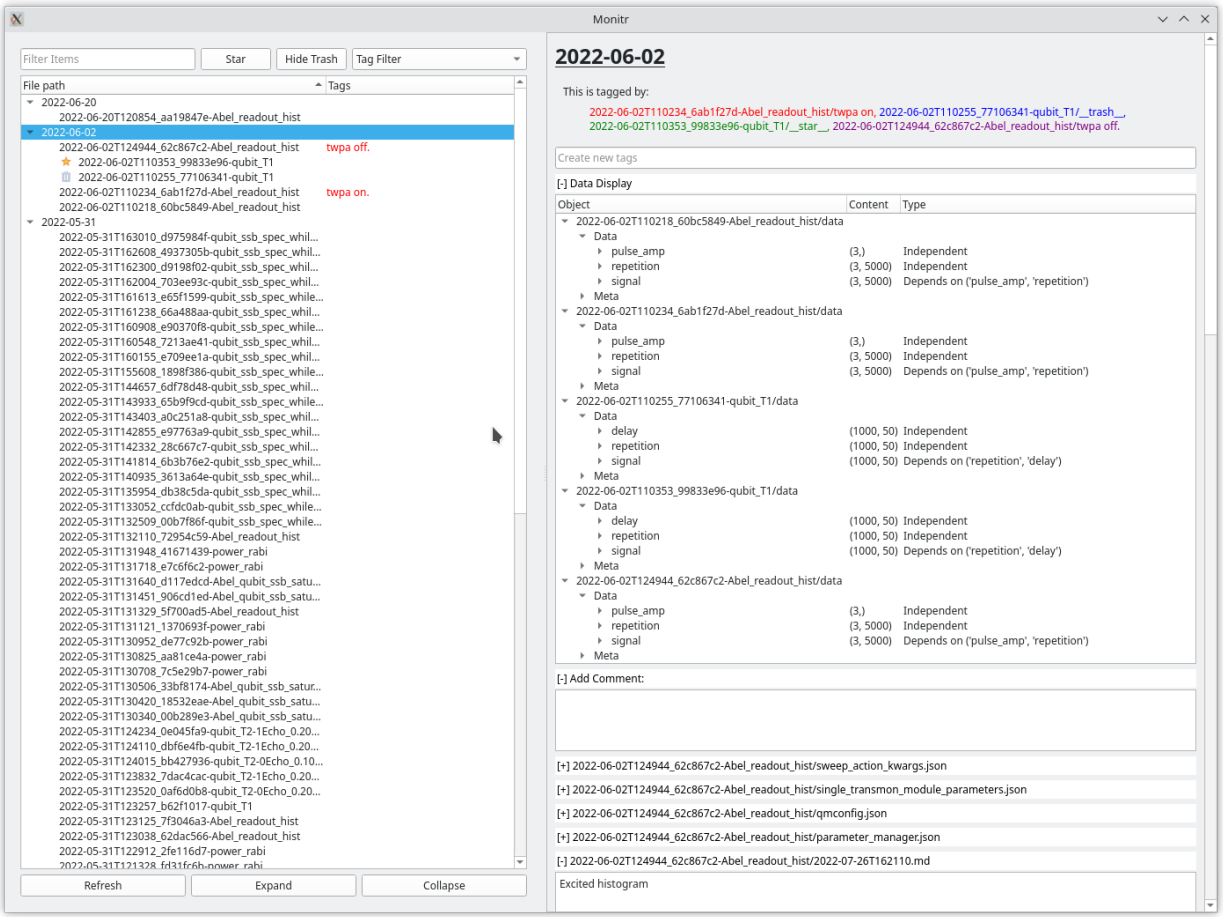
Main Window (File Explorer)

The following is an example taken from a real data folder.



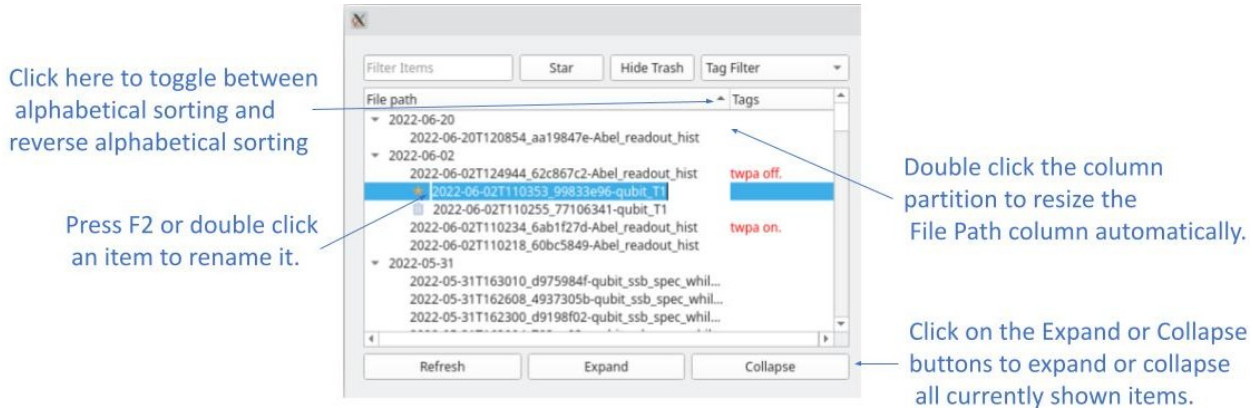
Basic Control And Usability

Selecting an item, either by clicking on it or selecting it with the arrow keys, will open the *Right Side Window*. If you select an item that contains children, the right side window will contain all of the information of the selected item and all of its children:



The following are a few things that the window can do:

- You can resize the File Path column to show the complete name of all shown items, double click in the column separator.
- You can rename any selected item by double clicking on its name or pressing F2 to change the currently selected item
- Clicking on the collapse or extend bottom buttons will collapse or extend all currently showing items in the tree.
- You can change the order in which the items are sorted (alphabetically or reverse alphabetically by clicking in the File Path column header.



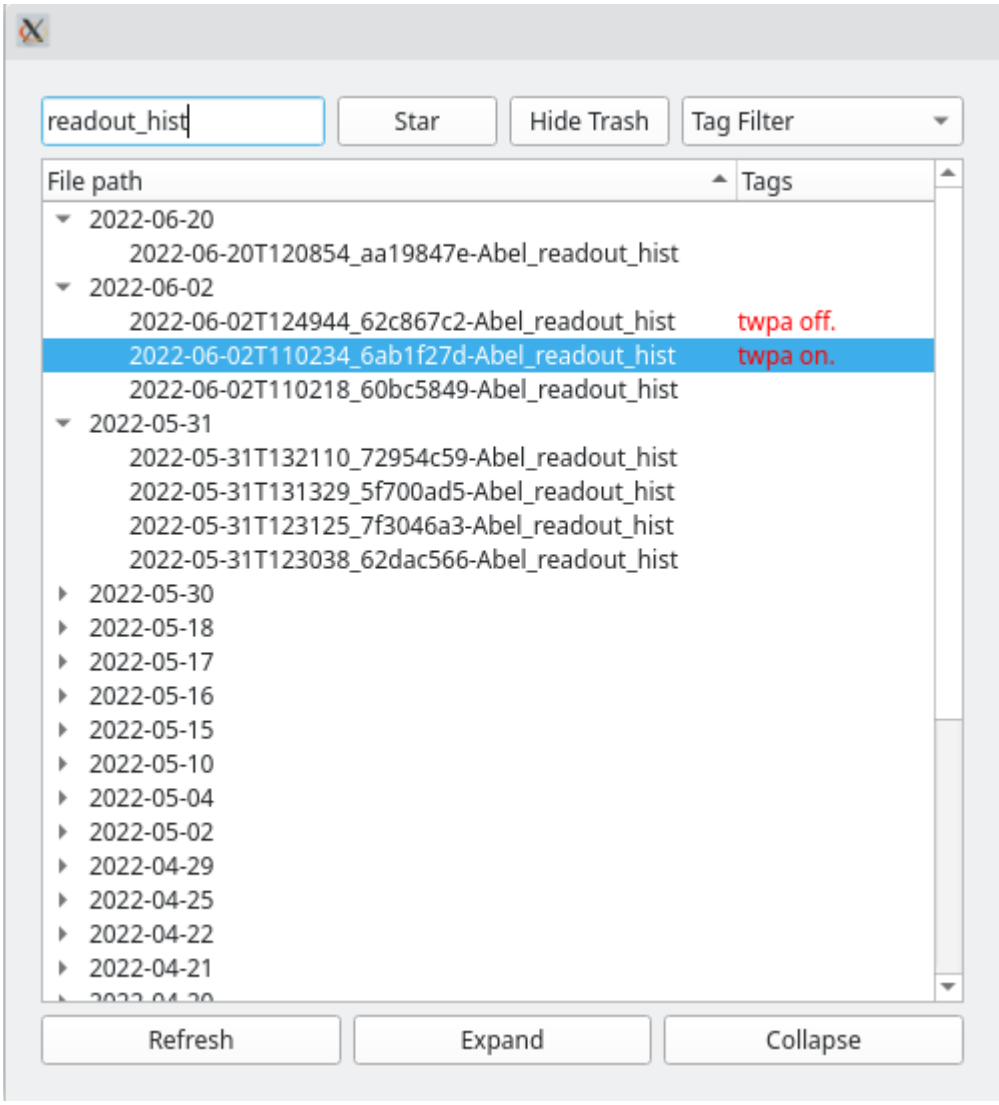
Tags

Monitr supports user created tags. Any file whose name end with ".tag" is considered a tag, and its file name becomes the text of the tag. In the overview "twpa off" is a custom tag, the folder 2022-06-02T124944_62c867c2-Abel_readout_hist contains the file "twpa off.tag". All custom tags will appear in the tags column and they can be used for custom *Filtering*. Y ou can create new tags with the tag creator (see *Tags Display and Tag Creator* or by simply adding a .tag file in a dataset folder). Two special tags exists: "__star__.tag" and "__trash__.tag", these will add icons to your items and have dedicated filtering buttons. The trash tag is the only tag that can be used to hide items.

Filtering

You can filter utilizing the "Filter Items" entry. All our filtering occurs by regex matching. When filtering, Monitr will show parent items if they contain children that match the filter requirements (even if the parent itself does not match). You can stack multiple searches using commas and only items that match with all of the queries will be shown.

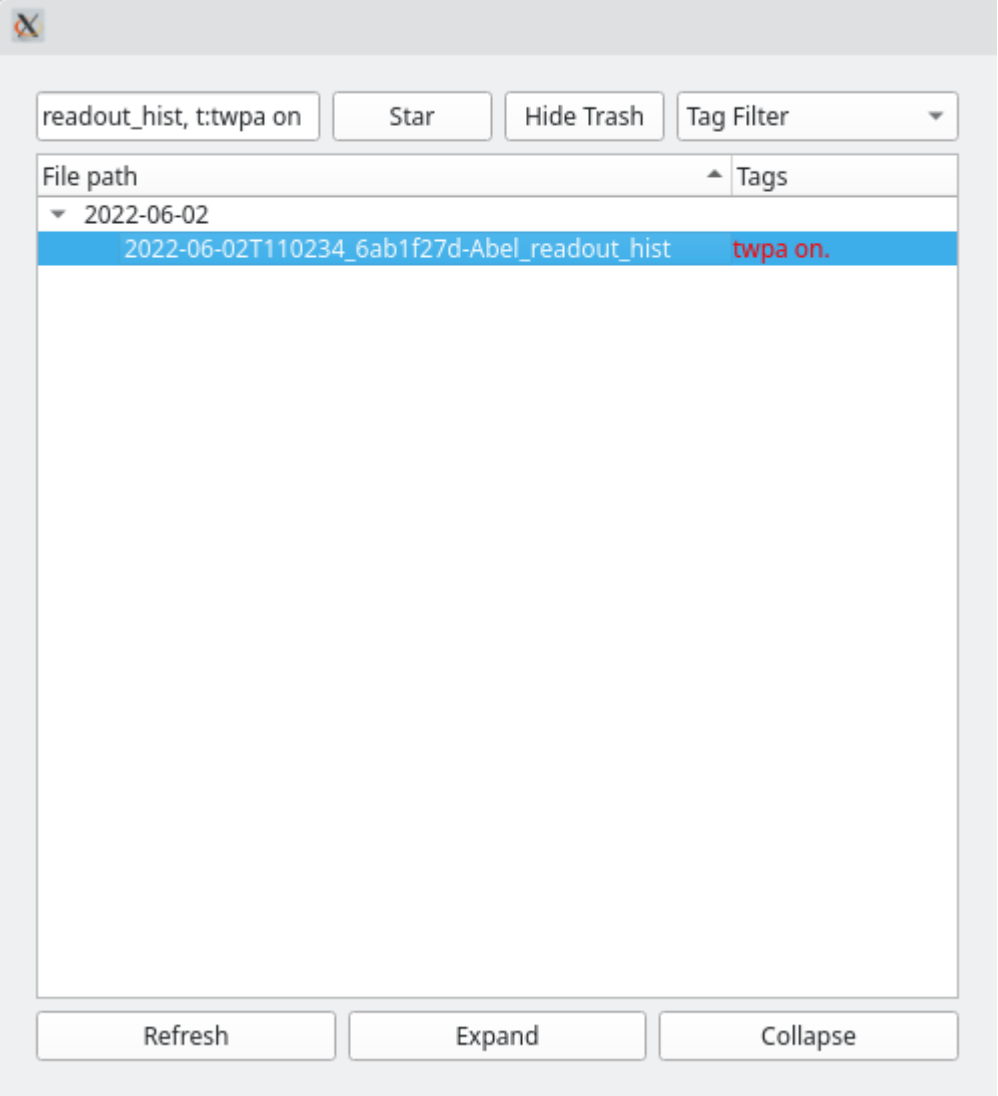
For example, if looking for "readout_hist" in our example, only items whose path match with that text will show up:



Filtering by tags can also be done through the tag filtering widget. Monitr keeps track of all the existing custom tags

and lets you select them in the tag filtering widget.

We can also filter through files inside of the item folder, like their tag names or image names:



Like this we can combine as many searches for all five of our currently supported search types.

Monitr supports filtering for:

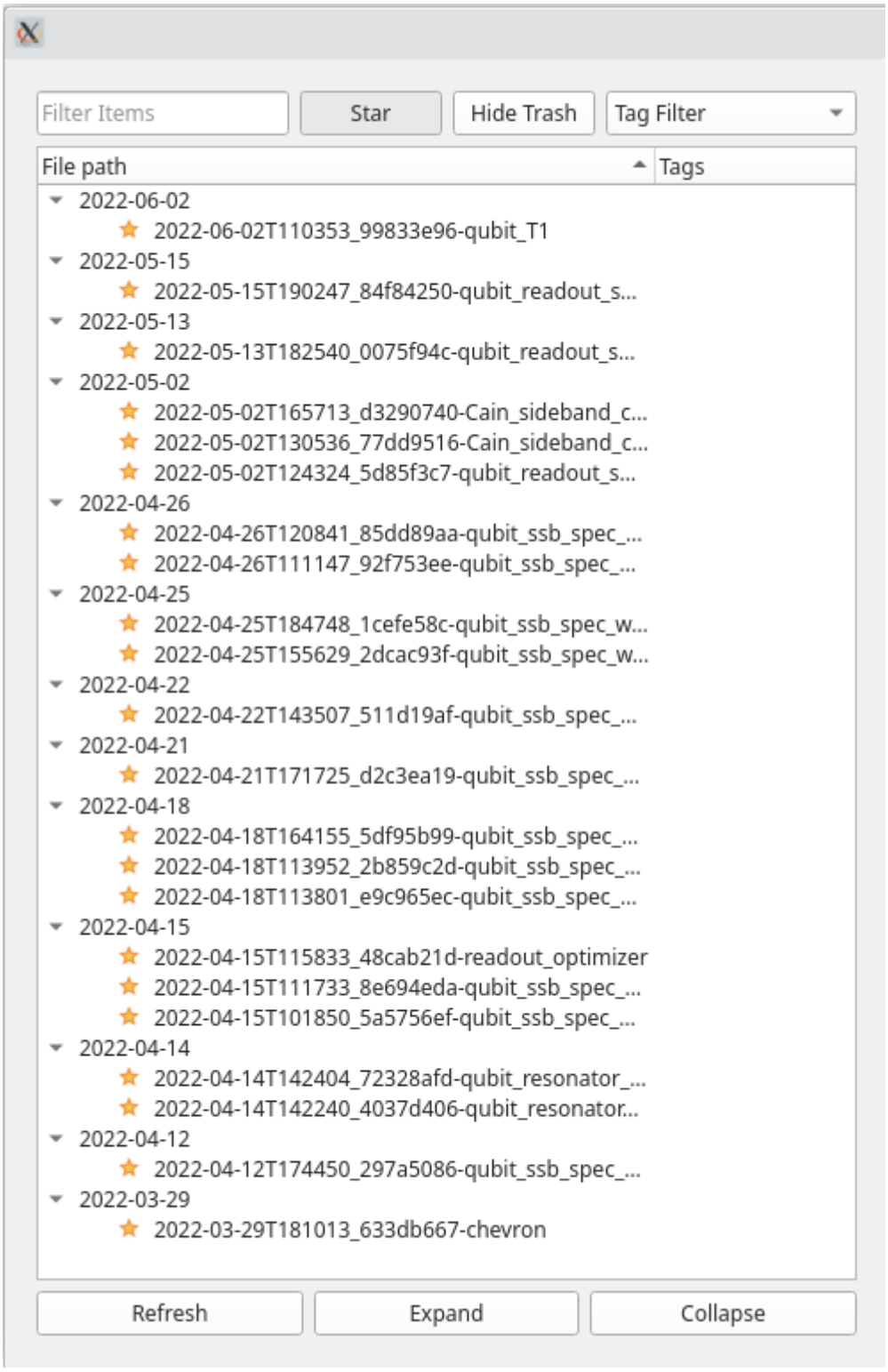
- Paths of datasets: Just write your query and all items whose entire paths match with the query will be shown.
- Tags: Start the queries with: “t:”, “T:”, or “tag:” directly followed by the tag you want to filter.
- Markdown files: Start the queries with: “m:”, “M:”, or “md:” directly followed by the text you want to filter. This will look inside of all folders inside of the monitoring directory and show only the datasets that contain a markdown file whose name matches with the query.
- Images: Start the queries with: “i:”, “I:”, or “image:” directly followed by the text you want to filter. This will look inside of all folders inside of the monitoring directory and show only the datasets that contain an image file whose name matches with the query.
- Json: Start the queries with: “j:”, “J:”, or “json:” directly followed by the text you want to filter. This will look inside of all folders inside of the monitoring directory and show only the datasets that contain a markdown file whose name matches with the query.

Star and Trash Items

You can star or trash items by right clicking on them and clicking on the star or trash button. This will create a tag file inside of that folder ("__star__.tag" or "__trash__.tag"), which indicates that that item is either a starred item or a trashed item. If you right click on an already starred/trashed item, an option to un-star or un-trash will appear instead.

The star and hide trash buttons (top right buttons of the main window picture) allow to toggle the two special filters corresponding to these special tags:

- Star: If this button is toggled, only starred items (or parents of starred) will be shown:



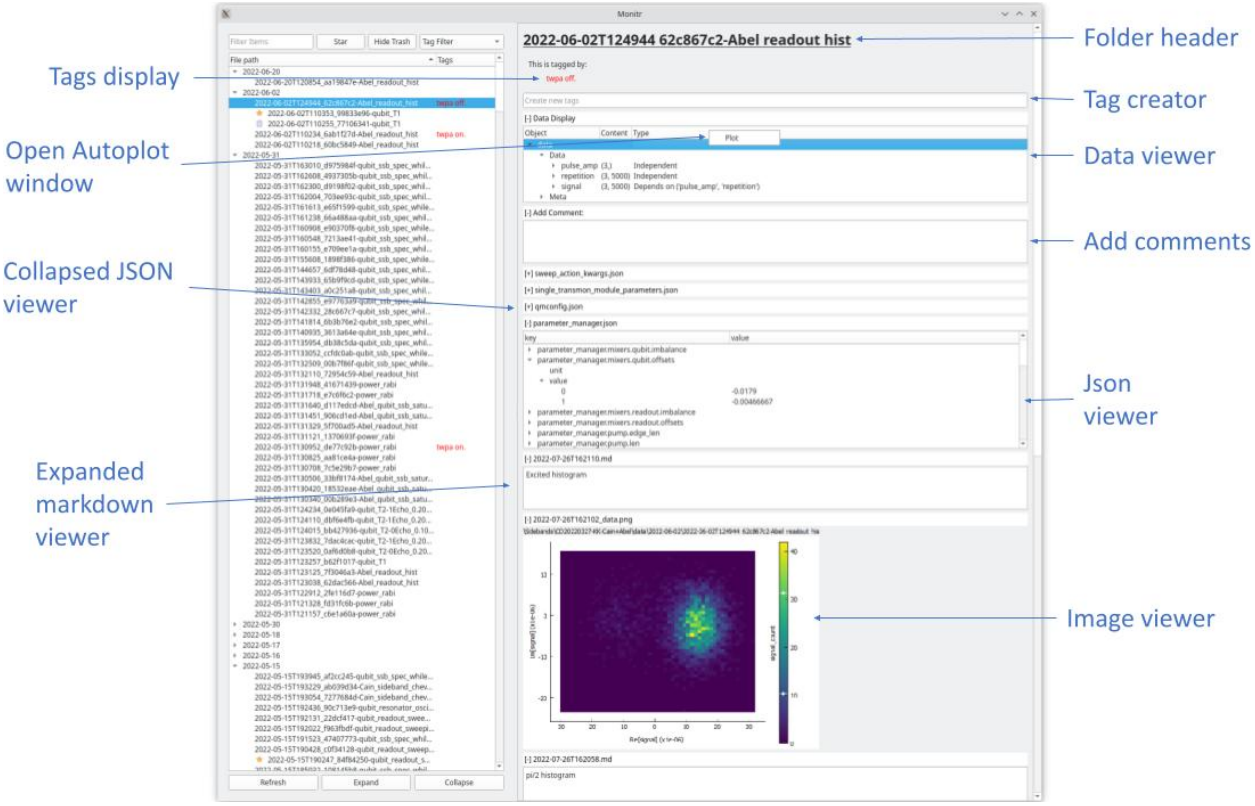
- Hide trash: Hides all trash items.

Note: The star and trash toggle buttons prioritize the parents status before their children. This means that if a parent folder (a folder containing datasets) is starred, all of its children will also be starred. In the same way, if a parent folder

is trash and the hide trash is activated, all of its children will also be hidden.

Right Side Window

When clicking on a dataset, the right side window will get populated with the files that are inside of it.



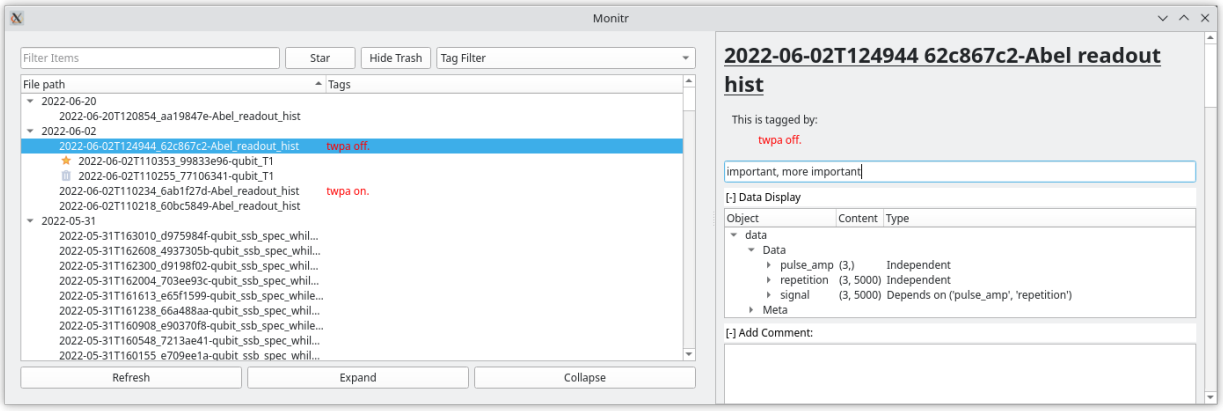
From the right side window you can open an autoplot window to plot the data of any data file that lives inside of that dataset.

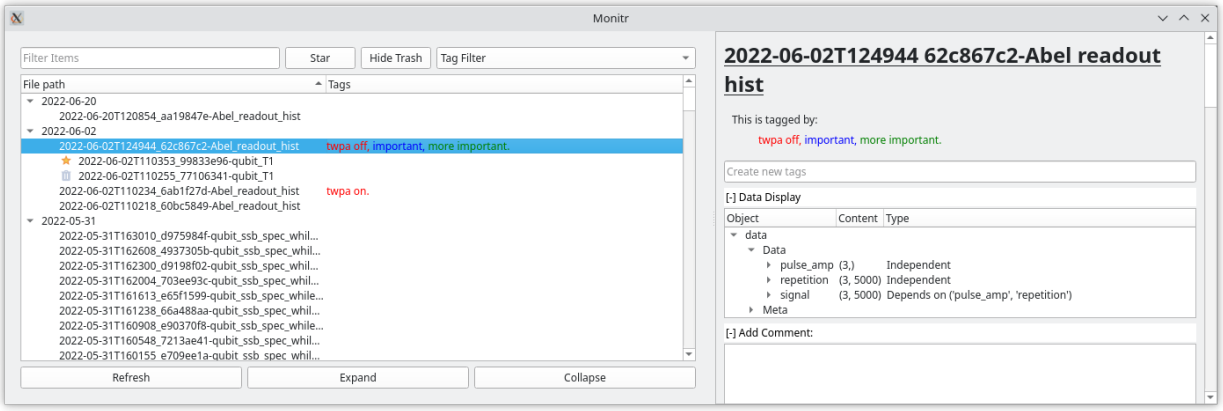
All viewer widgets (expect the dataset header, the tags display, and the tag creator) live inside collapsible windows. These windows can be collapsed to hide them.

Tags Display and Tag Creator

Under the file header you will see the tags display. In it, all tags that are tagging that dataset will appear with different colors.

Under the tag display, the tag creator is located. To create new tags simply type the tag in the line tag with the text: "Create new tags" and press enter. You can add multiple tags at a time by separating them with a coma:

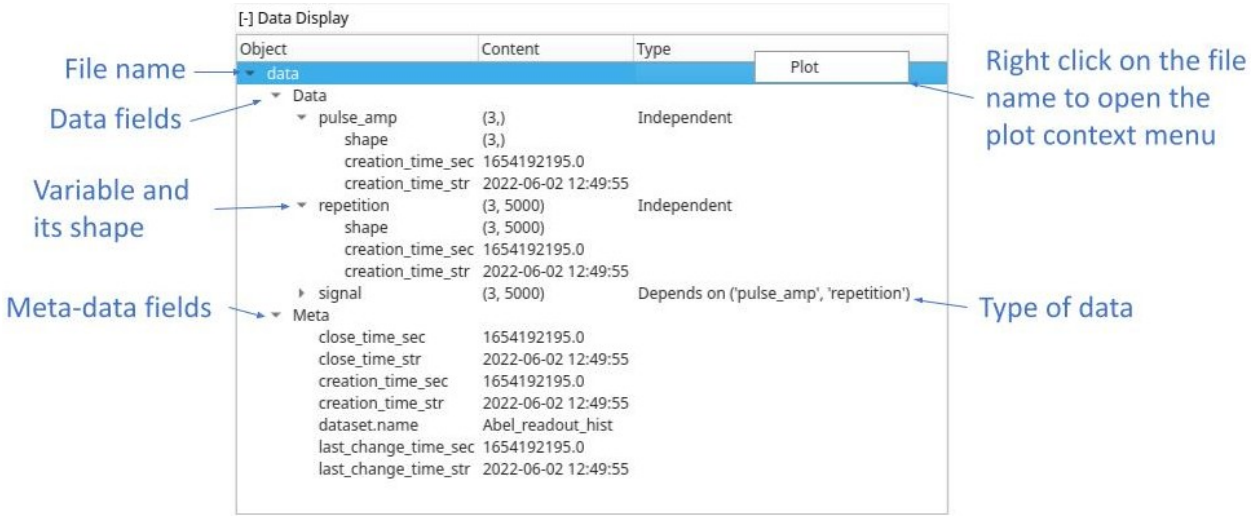




Data Viewer

If there is any valid data file in the selected dataset, the data viewer window will appear under the tag creator. All valid data files will appear as individual entries in that tree, including data files that are in folders inside the selected item (these will appear with the name of the folder before the file name).

To open an Autoplot window you have to right click in the file name line to open the plotting context menu.



Extra Files Viewers

All extra files are displayed under the comment creator (more on it later) and are ordered in alphabetical ordered. This is so that if the files have the correct time stamp at the beginning of their name they will appear from newest to oldest.

Monitr will display three different kinds of files:

- Json files: Json files will be displayed in a tree structure. They start collapsed by default.
- Image files: Currently Monitr only works with JPG or PNG file types. Images start expanded by default.
- Markdown files: We use markdown files for our comments. You can edit comments from Monitr by hovering in a comment window and clicking the edit button that will appear when hovering over it. Once you are done with your edit, the button will transform into a save button.

You can create new comments from the comment creator, just write the comment and click on the save button. A small dialog will appear when this happen asking for a file name. All comments created with this method will have a time stamp in their name before their given name. If the dialog is left empty, the comment will only have a time stamp as its file name.

If utilizing the pyqtgraph backend for autoplot, new images with the correct time stamp can be created by clicking the "Save Figure" button at the bottom toolbar. These images will be saved in the same folder as the DDH5 file automatically.

3.2 Instrumentserver

This is the main documentation for the instrument management tool *instrumentserver*. *instrumentserver* is written in python and has been tested to work well on Windows, Linux, and MacOS. This documentation is still very much ongoing work in progress, but should (hopefully!) already give an overview of what it's about, and how to use it.

The aim of *instrumentserver* is to facilitate QCoDeS access across a variety of process and devices. We communicate with the server through a TCP/IP connection allowing us to talk to it from any independent process or separate device in the same network.

instrumentserver also includes a virtual instrument called *parameter manager* whose job is to be a centralized and single source of truth for various parameters values with a user friendly graphical interface to facilitate changing parameters. For more information, please see

3.2.1 Basic Usage

Installation

At the moment *instrumentserver* is not on pip or conda so the only way of installing it is to install it from github directly. To do that first clone the github repo, and install into the desired environment using the editable pip install.

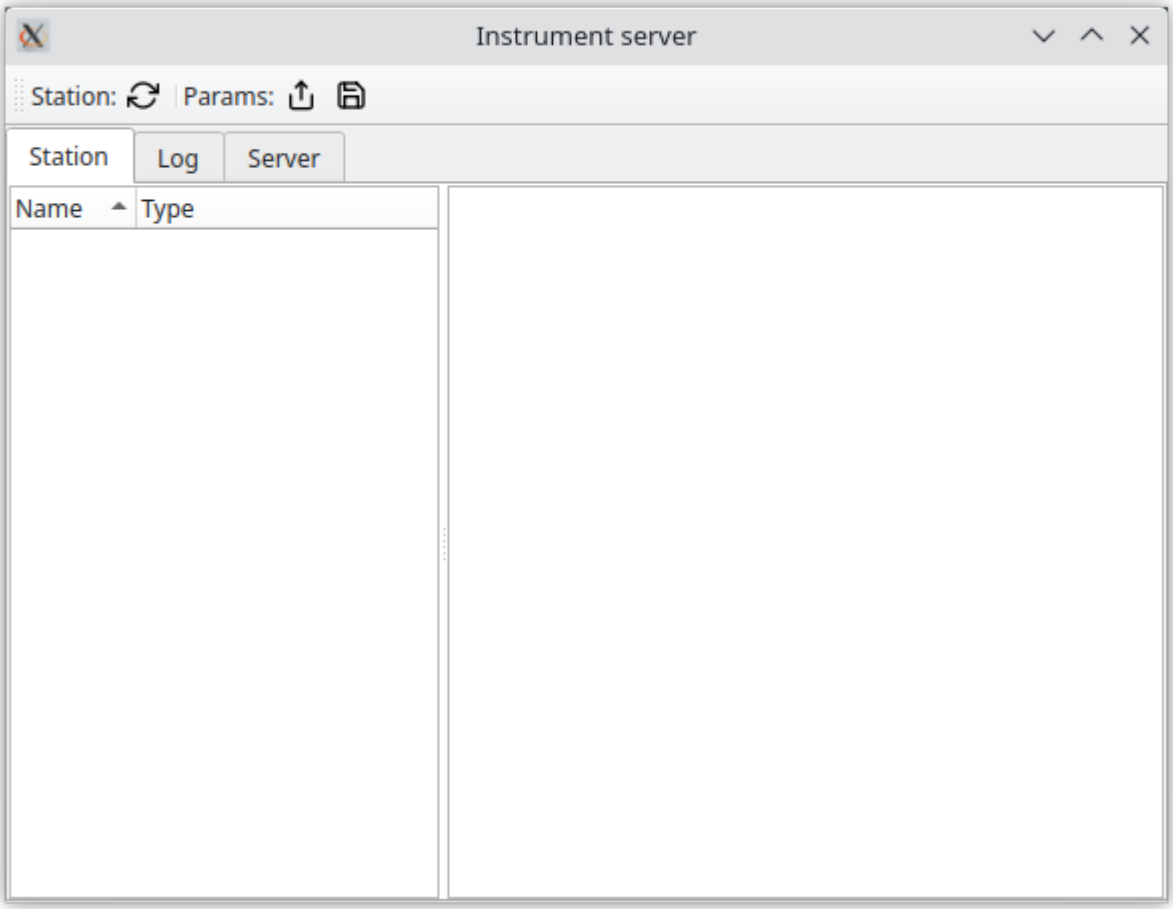
Quick Overview

Instrumentserver

To open the instrument server we simple run the command on a terminal:

```
$ instrumentserver
```

This will open the GUI of the server and start running it.



Note: The server can be run without a gui by passing the –gui False argument.

By default, instrumentserver listens to the local host IP address (127.0.0.1) and the port 5555. To be able to communicate with the server through other devices in the network we have to specify the IP address we want the server to listen to. For this we pass the argument -a <IP_address> and -p <port_number>:

```
$ instrumentserver -a 192.168.1.1 -p 12345
```

This will make the server listen to both the local host and the IP address 192.168.1.1 with port 12345.

We communicate with the server with Python code. This can be done anywhere that python can run, an IPython console, a Jupyter notebook, etc. The easiest way of creating *Client* and running the *find_or_create_instrument()*

method.

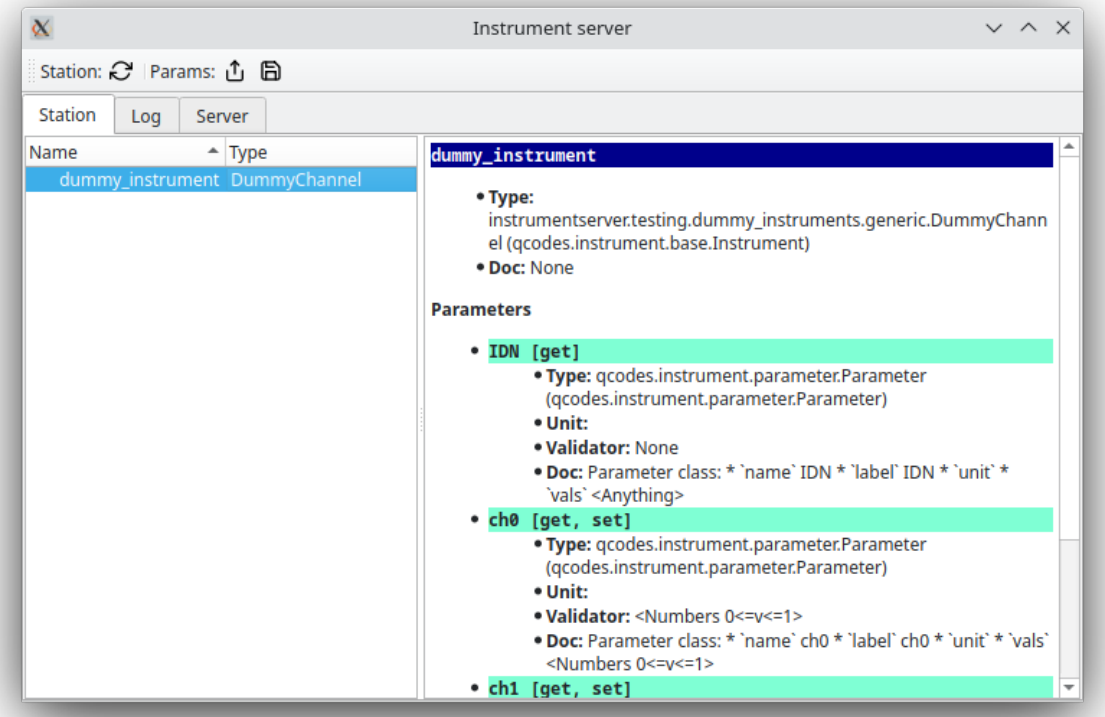
Note: Remember to pass the instrument specific arguments and keyword arguments necessary for the specific QCoDeS instrument you are trying to open.

This will look for the specified instrument with the given name in the server or create it if the instrument does not exist, and return it:

```
>>> cli = Client()
>>> dummy_instrument = cli.find_or_create_instrument(instrument_class='instrumentserver.
↪testing.dummy_instruments.generic.DummyChannel', name='dummy_instrument')
```

Note: If we are trying to talk to a server running in a different device in the network we need to specify the IP address and port with the arguments host and port when creating the *Client*.

After this we can see that the instrument has been created in the server.



After that we can use the instrument like a normal QCoDeS instrument. We can create a *Client* from any process and get the dummy_instrument by simply using the `get_instrument()` method:

```
>>> dummy_instrument = cli.get_instrument(name='dummy_instrument')
```

Parameter Manager

instrumentserver also comes with the virtual instrument Parameter Manager. The Parameter Manager allows us to store values in an instrument inside of the *instrumentserver*, allowing us to access them from any process or devices in the same network. The idea of it is to have a single source of truth for parameters whose values change frequently, and it provides a GUI from which you can change the values and easily see what they are.

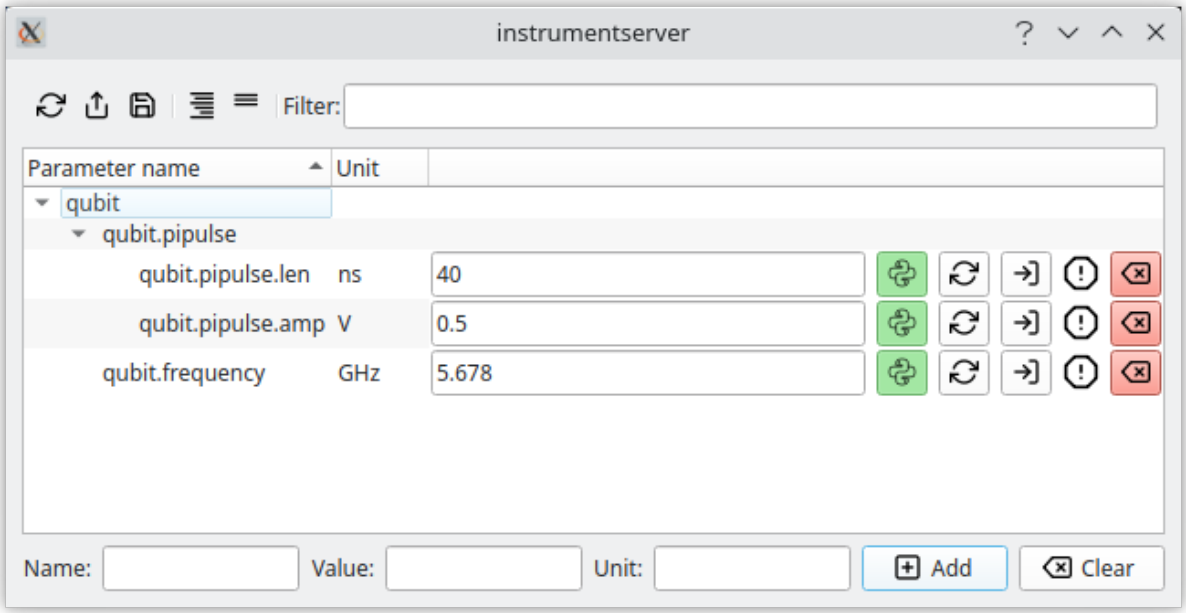
To open the Parameter Manager we first need to open the *instrumentserver*. Once we have the server open, we can run the command:

```
$ instrumentserver-param-manager
```

This will create an instance of the virtual instrument in the *instrumentserver* and will open the GUI for the Parameter Manager.

Note: At the moment the parameter manager can only be opened from the device that is currently hosting the server. If you are utilizing a different port, this can be specified by passing the terminal argument *–port* followed by the port.

We'll simply get an empty window now. The bottom of the window allows us to add arbitrary parameters and values, where dots serve as hierarchy separators (like objects and their children in python).



We can add some parameters and then retrieve them from anywhere that can run python code:

```
>>> cli = Client()
>>> params = cli.get_instrument('parameter_manager') # 'parameter_manager` is the name␣
↪the startup script gives the instrument by default
>>> params.qubit.pipulse.len()
40
```

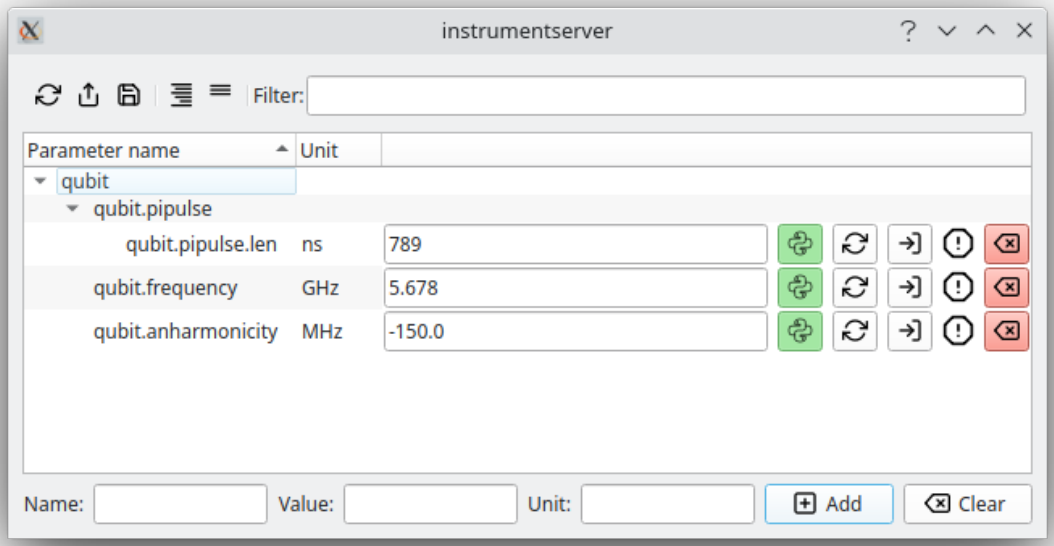
We can change parameters by calling the same function but passing as an argument the new value:

```
>>> params.qubit.pipulse.len(789)
```

We can add or remove parameters with code too:

```
>>> params.add_parameter('qubit.anharmonicity', initial_value=-150.0, unit='MHz')
>>> params.remove_parameter('qubit.pipulse.amp')
```

All of these changes get updated live in the GUI:



Changing things in the GUI will also be reflected in the code.

Warning: Changing something from the GUI only changes the code if we are calling the parameter manager directly. If we store a value in a separate variable and then change the GUI, the value in the variable might not get update. Because of this, we always recommend to call the Parameter Manager directly instead of saving the values in variables.

3.2.2 Code Documentation

Server

instrumentserver.server.core

Core functionality of the instrument server.

class instrumentserver.server.core.**Operation**(*value*)

Valid operations for the server.

get_existing_instruments = 'get_existing_instruments'

Get a list of instruments the server has instantiated.

create_instrument = 'create_instrument'

Create a new instrument.

get_blueprint = 'get_blueprint'

Get the blueprint of an object.

call = 'call'

Make a call to an object.

get_param_dict = 'get_param_dict'

Get the station contents as parameter dict.

set_params = 'set_params'

Set station parameters from a dictionary.

class instrumentserver.server.core.**InstrumentCreationSpec**(*instrument_class: str*, *name: str = ''*, *args: Optional[Tuple] = None*, *kwargs: Optional[Dict[str, Any]] = None*)

Spec for creating an instrument instance.

instrument_class: str

Driver class as string, in the format "global.path.to.module.DriverClass".

args: Optional[Tuple] = None

Arguments to pass to the constructor.

kwargs: Optional[Dict[str, Any]] = None

kw args to pass to the constructor.

class instrumentserver.server.core.**CallSpec**(*target: str*, *args: Optional[Any] = None*, *kwargs: Optional[Dict[str, Any]] = None*)

Spec for executing a call on an object in the station.

target: str

Full name of the callable object, as string, relative to the station object. E.g.: "instrument.my_callable" refers to `station.instrument.my_callable`.

args: Optional[Any] = None

Positional arguments to pass.

kwargs: Optional[Dict[str, Any]] = None

kw args to pass.

class instrumentserver.server.core.**ParameterBluePrint**(*name: str*, *path: str*, *base_class: str*, *parameter_class: str*, *gettable: bool = True*, *settable: bool = True*, *unit: str = ''*, *vals: Optional[Validator] = None*, *docstring: str = ''*, *setpoints: Optional[List[str]] = None*)

Spec necessary for creating parameter proxies.

class instrumentserver.server.core.**MethodBluePrint**(*name: str*, *path: str*, *call_signature: Signature*, *docstring: str = ''*)

Spec necessary for creating method proxies.

class instrumentserver.server.core.**InstrumentModuleBluePrint**(*name: str*, *path: str*, *base_class: str*, *instrument_module_class: str*, *docstring: str = ''*, *parameters: ~typing.Optional[~typing.Dict[str, ~instrumentserver.server.core.ParameterBluePrint]] = <factory>*, *methods: ~typing.Optional[~typing.Dict[str, ~instrumentserver.server.core.MethodBluePrint]] = <factory>*, *submodules: ~typing.Optional[~typing.Dict[str, ~instrumentserver.server.core.InstrumentModuleBluePrint]] = <factory>*)

Spec necessary for creating instrument proxies.

class instrumentserver.server.core.**ParameterBroadcastBluePrint**(*name: str*, *action: str*, *value: Optional[int] = None*, *unit: Optional[str] = None*)

Blueprint to broadcast parameter changes.

toDictFormat()

Formats the blueprint for easy conversion to dictionary later.

class instrumentserver.server.core.**ParameterSerializeSpec**(*path: Optional[str] = None*, *attrs: List[str] = <factory>*, *args: Optional[Any] = <factory>*, *kwargs: Optional[Dict[str, Any]] = <factory>*)

path: Optional[str] = None

Path of the object to serialize. None refers to the station as a whole.

attrs: List[str]

Which attributes to include for each parameter. Default is ['values'].

args: Optional[Any]

Additional arguments to pass to the serialization function serialize.toParamDict().

kwargs: Optional[Dict[str, Any]]

Additional kw arguments to pass to the serialization function serialize.toParamDict().

class instrumentserver.server.core.**ServerInstruction**(*operation: ~instrumentserver.server.core.Operation, create_instrument_spec: ~typing.Optional[~instrumentserver.server.core.InstrumentCreationSp = None, call_spec: ~typing.Optional[~instrumentserver.server.core.CallSpec] = None, requested_path: ~typing.Optional[str] = None, serialization_opts: ~typing.Optional[~instrumentserver.server.core.ParameterSerializeSp = None, set_parameters: ~typing.Optional[~typing.Dict[str, ~typing.Any]] = <factory>, args: ~typing.Optional[~typing.List[~typing.Any]] = <factory>, kwargs: ~typing.Optional[~typing.Dict[str, ~typing.Any]] = <factory>*)

Instruction spec for the server.

Valid operations:

- *Operation.get_existing_instruments* – get the instruments currently instantiated in the station.
  - **Required options:** -
  - **Return message:** dictionary with instrument name and class (as string).
- *Operation.create_instrument* – create a new instrument in the station.
  - **Required options:** *create_instrument_spec*
  - **Return message:** None
- *Operation.call* – make a call to an object in the station.
  - **Required options:** *call_spec*
  - **Return message:** The return value of the call.
- *Operation.get_blueprint* – request the blueprint of an object
  - **Required options:** *requested_path*
  - **Return message:** The blueprint of the object.
- *Operation.get_param_dict* – request parameters as dictionary Get the parameters of either the full station or a single object.
  - **Options:** *serialization_opts*
  - **Return message:** param dict.

operation: *Operation*

This is the only mandatory item. Which other fields are required depends on the operation.

create_instrument_spec: Optional[*InstrumentCreationSpec*] = None

Specification for creating an instrument.

call_spec: Optional[*CallSpec*] = None

Specification for executing a call.

requested_path: Optional[str] = None

Name of the instrument for which we want the blueprint.

serialization_opts: Optional[*ParameterSerializeSpec*] = None

Options for serialization.

set_parameters: Optional[Dict[str, Any]]

Setting parameters in bulk with a paramDict.

args: Optional[List[Any]]

Generic arguments.

kwargs: Optional[Dict[str, Any]]

Generic keyword arguments.

class instrumentserver.server.core.**ServerResponse**(*message: Optional[Any] = None*, *error: Union[None, str, Warning, Exception] = None*)

Spec for what the server can return.

If the requested operation succeeds, *message* will the return of that operation, and *error* is None. See *ServerInstruction* for a documentation of the expected returns. If an error occurs, *message* is typically None, and *error* contains an error message or object describing the error.

message: Optional[Any] = None

The return message.

error: Union[None, str, Warning, Exception] = None

Any error message occured during execution of the instruction.

class instrumentserver.server.core.**StationServer**(*parent: Optional[QObject] = None*, *port: int = 5555*, *allowUserShutdown: bool = False*, *addresses: List[str] = []*, *initScript: Optional[str] = None*)

The main server object.

Encapsulated in a separate object so we can run it in a separate thread.

Port should always be an odd number to allow the next even number to be its corresponding publishing port.

messageReceived

Signal(str, str) – emit messages for display in the gui (or other stuff the gui wants to do with it. Arguments: the message received, and the reply sent.

serverStarted

Signal(int) – emitted when the server is started. Arguments: the port.

finished

Signal() – emitted when we shut down.

instrumentCreated

Signal(Dict) – emitted when a new instrument was created. Argument is the blueprint of the instrument.

parameterSet

Signal(str, Any) – emitted when a parameter was set Arguments: full parameter location as string, value.

parameterGet

Signal(str, Any) – emitted when a parameter was retrieved Arguments: full parameter location as string, value.

funcCalled

Signal(str, List[Any], Dict[str, Any], Any) – emitted when a function was called Arguments: full function location as string, arguments, kw arguments, return value.

startServer() → None

Start the server. This function does not return until the ZMQ server has been shut down.

executeServerInstruction(*instruction:* ServerInstruction) → Tuple[*ServerResponse*, str]

This is the interpreter function that the server will call to translate the dictionary received from the proxy to instrument calls.

Parameters
instruction – The instruction object.

Returns
The results returned from performing the operation.

instrumentserver.server.core.**startServer**(*port: int = 5555*, *allowUserShutdown: bool = False*, *addresses: List[str] = []*, *initScript: Optional[str] = None*) → Tuple[*StationServer*, QThread]

Create a server and run in a separate thread.

Returns
The server object and the thread it's running in.

Client

Base client

class instrumentserver.client.core.**BaseClient**(*host='localhost'*, *port=5555*, *connect=True*, *timeout=5000*)

Simple client for the StationServer. When a timeout happens, a RunTimeError is being raised. This error is there just to warn the user that a timeout has occurred. After that the client will restart the socket to continue the normal work.

Parameters

- **host** – The host address of the server, defaults to localhost.
- **port** – The port of the server, defaults to the value of DEFAULT_PORT.
- **connect** – If true, the server connects as it is being constructed, defaults to True.
- **timeout** – Amount of time that the client waits for an answer before declaring timeout in ms. Defaults to 5000.

recv_timeout

Timeout for server replies.

Proxy module

Client:

class instrumentserver.client.proxy.**Client**(*host='localhost'*, *port=5555*, *connect=True*, *timeout=5000*)

Client with common server requests as convenience functions.

list_instruments() → Dict[str, str]

Get the existing instruments on the server.

find_or_create_instrument(*instrument_class: str*, *name: str*, **args: Any*, ***kwargs: Any*) → ProxyInstrumentModule

Create a new instrument on the server and return a proxy for the new instrument.

Parameters

- **instrument_class** – Class of the instrument to create or a string of of the class.
- **name** – Name of the new instrument.
- **args** – Positional arguments for new instrument instantiation.
- **kwargs** – Keyword arguments for new instrument instantiation.

Returns
A new virtual instrument.

SubClient:

class instrumentserver.client.proxy.**SubClient**(*instruments: Optional[List[str]] = None*, *sub_host: str = 'localhost'*, *sub_port: int = 5556*)

Specific subscription client used for real-time parameter updates.

update
Signal(str) – emitted when the server broadcast either a new parameter or an update to an existing one.

connect()
Connects the subscription client with the broadcast and runs an infinite loop to check for updates.

It should always be run on a separate thread or the program will get stuck in the loop.

3.3 Labcore

This is the main documentation for the experiment control tools *labcore*. *labcore* is written in python and has been tested to work well on Windows, Linux, and MacOS. This documentation is still very much ongoing work in progress, but should (hopefully!) already give an overview of what it's about, and how to use it.

The aim of *labcore* is to provide tools to facilitate the modularization of repetition of experiment control software. *Labcore* allows you to separate different actions of an experiment and modularizing it, allowing you to mix and match different parts or entire parts of experiments. Having this structure around experiments allow you to have access to main characteristics (such as the type of data acquired during the run) in advance, before any measurement code is executed.

3.3.1 Basic Usage

Installation

At the moment *instrumentserver* is not on pip or conda so the only way of installing it is to install it from github directly. To do that first clone the github repo, and install into the desired environment using the editable pip install.

Quick Overview

Labcore is a set of tools that help you run experiments in your lab. Currently it only includes the functionalities for creating `Sweeps`. A Sweep object can be iterated over and is our primary way of executing measurements. Each iteration performs some actions and return some data in the form of a dictionary, we call this dictionary generated after each step a **record**.

A sweep is composed of 2 parts, a **pointer**, and **actions**:

- A **pointer** is again an iterable, and represents what our "independent" variable, this represents the variable we control and change to the values specified in the iterator.
- An **action** is a callable that may take the values the pointer is returning at each iteration as arguments. Each action is executed once for each pointer, and you can have as many actions for a single pointer as needed.

Defining and executing a basic sweep would look something like this:

```
>>> sweep_object = Sweep(range(5), func_1, func_2)
>>> for data in sweep_object:
>>>     print(data)
>>> {variable_1: some_data, variable_2: more_data}
>>> {variable_1: different_data, variable_2: more_different_data}
```

Where the `range` function is your **pointer** and the 2 functions are your **actions**.

Once the sweep is created but before execution we can easily infer what records the sweep produces. Sweeps can be combined in ways that result in nesting, zipping, or appending. Combining sweeps again results in a Sweep. This will allow us to construct modular measurements from pre-defined blocks. For more information please see *Sweeping*.

3.3.2 Sweeping

Basic Example

A Sweep is created out of two main components, an iterable **pointer** and a variable number of **actions**. Both **pointer** and **actions** may generate **records**.

The most bare example would look like this:

```
>>> for data in Sweep(range(3)):
>>>     print(data)
{}
{}
{}
```

In this example, the `range(3)` iterable object is our **pointer**. This Sweep does not contain any **actions** or generate any **records**, but instead simply loops over the iterable **pointer**.

Recording Data

Concepts

Even though the **pointer** in the previous example does generate data, we cannot see it when we iterate through the sweep. To have **pointers** and **actions** generate visible data, we need to annotate them so that they generate **records**.

For a Sweep to know that either a **pointer** (an iterable object), or an **action** (a callable object) they need to be wrapped by an instance of *DataSpec*. *DataSpec* is a data class that holds information about the variable itself. Understanding the inner workings are not necessary to fully utilize Sweeps, however, it is good to know they exists and what information they hold.

Annotating a function or generator does not change what they do, it gets the values generated by them and places them inside a dictionary with the key for those values being the name of the variable we assign.

The important fields of a *DataSpec* are its *name* and *depends_on* fields. *name*, simply indicates the name of the variable, e.i. the key that the sweep will have for the value of this variable. *depends_on* indicates whether the variable is an independent variable (we control) or a dependent variable (the things we are trying to measure). If `depends_on=None` it means this variable is an independent variable. If `depends_on=['x']`, this variable is dependent on a separate variable with name `x`. If `depends_on=[]`, the variable will be automatically assigned as a dependent of all other independents in the same Sweep.

Note: *DataSpec*, also contains two more fields: *unit* and *type*, these however, have no impact in the way the code behaves and are for adding extra metadata for the user.

While this might seem like a lot of information, its use is very intuitive and easy to use once you get used to it.

Implementation

To wrap functions we use the *recording* decorator on the function we want to annotate:

```
>>> @recording(DataSpec('x'), DataSpec('y', depends_on=['x'], type='array'))
>>> def measure_stuff(n, *args, **kwargs):
>>>     return n, np.random.normal(size=n)
>>>
>>> measure_stuff(1)
{'x': 1, 'y': array([0.70663348])}
```

In the example above we annotate the function `measure_stuff()` indicating that the first item it returns is `x`, an independent variable since it does not have a `depends_on` field, and the second item is `y`, a variable that depends on `x`.

We can annotate generators in the same way:

```
>>> @recording(DataSpec('a'))
>>> def make_sequence(n):
>>>     for i in range(n):
>>>         yield i
>>>
>>> for data in make_sequence(3):
>>>     print(data)
{'a': 0}
{'a': 1}
{'a': 2}
```

A nicer way of creating *DataSpec* instances is to use the functions *independent* and *dependent*. This function just makes the recording of data easier to read. *independent* does not let you indicate the depends_on field while *dependent*, has an empty list (indicating that it depends an all other independents) as a default.

```
>>> @recording(independent('x'), dependent('y', type='array'))
>>> def measure_stuff(n, *args, **kwargs):
>>>     return n, np.random.normal(size=n)
>>>
>>> measure_stuff(1)
{'x': 1, 'y': array([1.60113794])}
```

Note: You can also use the abbreviations:

- *ds* for shorter *DataSpec*
- *indep()* for shorter *independent*
- *dep()* for shorter *dependent*

Sometimes we don't want to annotate a function or generator itself, but instead we want to annotate at the moment of execution. For this we can use the function *record_as()* to annotate any function or generator on the fly:

```
>>> def get_some_data(n):
>>>     return np.random.normal(size=n)
>>>
>>> record_as(get_some_data, independent('random_var'))(3)
{'random_var': array([0.16099358, 0.74873271, 0.01160423])}
```

You can add multiple *DataSpecs* with in a single *record_as()*:

```
>>> for data in record_as(zip(np.linspace(0,1,3), np.arange(3)), indep('x'), dep('y')):
>>>     print(data)
{'x': 0.0, 'y': 0}
{'x': 0.2, 'y': 1}
{'x': 0.4, 'y': 2}
```

It will also make sure to add items for annotated **records** (by adding None items to any empty **record**) that do not have any values assigned to them:

```
>>> for data in record_as(np.linspace(0,1,3), indep('x'), dep('y')):
>>>     print(data)
{'x': 0.0, 'y': None}
{'x': 0.5, 'y': None}
{'x': 1.0, 'y': None}
```

And it will ignore any extra values that are not annotated:

```
>>> for data in record_as(zip(np.linspace(0,1,3), np.arange(3)), indep('x')):
>>>     print(data)
{'x': 0.0}
{'x': 0.5}
{'x': 1.0}
```

Construction of Sweeps

Now that we know how to annotate data so that it generates records, we can finally start creating Sweeps that creates some data. A Sweep is composed of two main parts: **pointers** and **actions**:

- **Pointers** are iterables that the Sweep iterates through, these usually represent the independent variables of our experiments.
- **Actions** are callables that get called after each iteration of our **pointer** and usually are in charge of performing anything that needs to happen at every iteration of the experiment. This can be either set up a instruments and usually includes measuring a dependent variable too.

Both **pointers** and **actions** can generate **records** if annotated correctly, but it is not a requirement.

Basic Sweeps

A basic annotated Sweep looks something like this:

```
>>> def my_func():
>>>     return 0
>>>
>>> sweep = Sweep(
>>>     record_as(range(3), independent('x')), # This is the pointer. We specify 'x' as␣
↪an independent (we control it).
>>>     record_as(my_func, dependent('y'))) # my_func is an action. We specify 'y' as a␣
↪dependent.
```

Once the Sweep is created we can see the **records** it will produce by utilising the function method *get_data_specs()*:

```
>>> sweep.get_data_specs()
(x, y(x))
```

Printing a Sweep will also display more information about, specifying the pointers, the actions taken afterwards and the **records** it will produce:

```
>>> print(sweep)
range(0, 3) as {x} >> my_func() as {y}
==> {x, y(x)}
```

Now to run the Sweep we just have to iterate through it:

```
>>> for data in sweep:
>>>     print(data)
{'x': 0, 'y': 0}
{'x': 1, 'y': 0}
{'x': 2, 'y': 0}
```

If you are trying to sweep over a single parameter, a more convenient syntax for creating Sweep is to utilize the *sweep_parameter()* function:

```
>>> sweep = sweep_parameter('x', range(3), record_as(my_func, 'y'))
>>> for data in sweep:
>>>     print(data)
{'x': 0, 'y': 0}
```

(continues on next page)

(continued from previous page)

```
{'x': 1, 'y': 0}
{'x': 2, 'y': 0}
```

There is no restriction on how many parameters a **pointer** or an **action** can generate as long as each parameter is properly annotated.

```
>>> def my_func():
>>>     return 1, 2
>>>
>>> sweep = Sweep(
>>>     record_as(zip(range(3), ['a', 'b', 'c']), independent('number'), independent(
↪'string')), # a pointer with two parameters
>>>     record_as(my_func, 'one', 'two')) # an action with two parameters
>>>
>>> print(sweep.get_data_specs())
>>>
>>> for data in sweep:
>>>     print(data)
(number, string, one(number, string), two(number, string))
{'number': 0, 'string': 'a', 'one': 1, 'two': 2}
{'number': 1, 'string': 'b', 'one': 1, 'two': 2}
{'number': 2, 'string': 'c', 'one': 1, 'two': 2}
```

Specifying Options Before Executing a Sweep

Many **actions** we are using take optional parameters that we only want to specify just before executing the Sweep (but are constant throughout the Sweep).

If we don't want to resort to global variables we can do so by using the method *set_options()*. It accepts the names of any **action** function in that Sweep as keywords, and dictionaries containing keyword arguments to pass to those functions as values. Keywords specified in this way always override keywords that are passed around internally in the sweep (for more information see the *Passing Parameters in a Sweep* section):

```
>>> def test_fun(a_property=False, *args, **kwargs):
>>>     print('inside test_fun:')
>>>     print(f"a_property: {a_property}")
>>>     print(f"other stuff:", args, kwargs)
>>>     print('----')
>>>     return 0
>>>
>>> sweep = sweep_parameter('value', range(3), record_as(test_fun, dependent('data')))
>>> sweep.set_options(test_fun=dict(a_property=True, another_property='Hello'))
>>>
>>> for data in sweep:
>>>     print("Data:", data)
>>>     print('----')
inside test_fun:
property: True
other stuff: () {'value': 0, 'another_property': 'Hello'}
----
Data: {'value': 0, 'data': 0}
```

(continues on next page)

(continued from previous page)

```
----
inside test_fun:
property: True
other stuff: () {'value': 1, 'data': 0, 'another_property': 'Hello'}
----
Data: {'value': 1, 'data': 0}
----
inside test_fun:
property: True
other stuff: () {'value': 2, 'data': 0, 'another_property': 'Hello'}
----
Data: {'value': 2, 'data': 0}
```

A QCoDeS Parameter Sweep

If you are using QCoDeS to interact with hardware, it is very common to want to do a sweep over a QCoDeS parameter. In this minimal example we set a parameter (`x`) to a range of values, and get data from another parameter for each set value.

```
>>> def measure_stuff():
>>>     return np.random.normal()
>>>
>>> x = Parameter('x', set_cmd=lambda x: print(f'setting x to {x}'), initial_value=0) #␣
↪QCoDeS Parameter
>>> data = Parameter('data', get_cmd=lambda: np.random.normal()) # QCoDeS Parameter
>>>
>>> for record in sweep_parameter(x, range(3), get_parameter(data)):
>>>     print(record)
setting x to 0
setting x to 0
{'x': 0, 'data': -0.4990053668503893}
setting x to 1
{'x': 1, 'data': -0.5132204673887943}
setting x to 2
{'x': 2, 'data': 1.8634243556469932}
```

Sweep Combinations

One of the most valuable features of Sweeps is their ability to be combined through the use of operators. This allows us to mix and match different aspects of an experiment without having to rewrite code. We can combine different Sweeps with each other or different annotated **actions**

Appending

The most basic combination of Sweeps is appending them. When appending two Sweeps, the resulting sweep will execute the first Sweep to completion followed by the second Sweep to completion. To append two Sweeps or actions we use the + symbol:

```
>>> def get_random_number():
>>>     return np.random.rand()
>>>
>>> Sweep.record_none = False # See note on what this does.
>>>
>>> sweep_1 = sweep_parameter('x', range(3), record_as(get_random_number, dependent('y
↪')))
>>> sweep_2 = sweep_parameter('a', range(4), record_as(get_random_number, dependent('b
↪')))
>>> my_sweep = sweep_1 + sweep_2
>>>
>>> for data in my_sweep:
>>>     print(data)
{'x': 0, 'y': 0.34404570192577155}
{'x': 1, 'y': 0.02104831292457654}
{'x': 2, 'y': 0.9006367857458307}
{'a': 0, 'b': 0.10539935409724577}
{'a': 1, 'b': 0.9368463758729733}
{'a': 2, 'b': 0.9550070757291859}
{'a': 3, 'b': 0.9812445448108895}
```

Note: `Sweep.return_none` controls whether we include data fields that have returned nothing during setting a pointer or executing an action. Setting it to true (the default) guarantees that each data spec of the sweep has an entry per sweep point, even if it is `None`. For more information see: *Passing Parameters in a Sweep* section.

Multiplying

By multiplying we refer to an inner product, i.e. the result is what you'd expect from zip-ing two iterables. To multiply two Sweeps or actions we use the * symbol. A basic example is if we have a sweep and want to attach another action to each sweep point:

```
>>> my_sweep = (
>>>     sweep_parameter('x', range(3), record_as(get_random_number, dependent('data_1')))
>>>     * record_as(get_random_number, dependent('data_2'))
>>> )
>>>
>>> print(sweep.get_data_specs())
>>> print('----')
```

(continues on next page)

(continued from previous page)

```
>>>
>>> for data in my_sweep:
>>>     print(data)
(x, data_1(x), data_2(x))
----
{'x': 0, 'data_1': 0.12599818360565485, 'data_2': 0.09261266841087679}
{'x': 1, 'data_1': 0.5665798938860637, 'data_2': 0.7493750740615404}
{'x': 2, 'data_1': 0.9035085438172156, 'data_2': 0.5419023528195611}
```

If you are combining two different Sweeps, then we get zip-like behavior while maintain the dependency structure separate:

```
>>> my_sweep = (
>>>     sweep_parameter('x', range(3), record_as(get_random_number, dependent('data_1')))
>>>     * sweep_parameter('y', range(5), record_as(get_random_number, dependent('data_2
↪')))
>>> )
>>>
>>> print(sweep.get_data_specs())
>>> print('----')
>>>
>>> for data in my_sweep:
>>>     print(data)
(x, data_1(x), y, data_2(y))
----
{'x': 0, 'data_1': 0.3808452915069015, 'y': 0, 'data_2': 0.14309246334791337}
{'x': 1, 'data_1': 0.6094608905204076, 'y': 1, 'data_2': 0.3560530722571186}
{'x': 2, 'data_1': 0.15950240245080072, 'y': 2, 'data_2': 0.2477391943438858}
```

Nesting

Nesting two Sweeps runs the entire second Sweep for each point of the first Sweep. A basic example is if we have multiple Sweep parameters against each other and we want to perform a measurement at each point. To nest two Sweeps we use the `@` symbol:

```
>>> def measure_something():
>>>     return np.random.rand()
>>>
>>> my_sweep = (
>>>     sweep_parameter('x', range(3))
>>>     @ sweep_parameter('y', np.linspace(0,1,3))
>>>     @ record_as(measure_something, 'my_data')
>>> )
>>>
>>> for data in my_sweep:
>>>     print(data)
{'x': 0, 'y': 0.0, 'my_data': 0.727404046865409}
{'x': 0, 'y': 0.5, 'my_data': 0.11112429412122715}
{'x': 0, 'y': 1.0, 'my_data': 0.09081900115421426}
{'x': 1, 'y': 0.0, 'my_data': 0.8160224024098803}
{'x': 1, 'y': 0.5, 'my_data': 0.1517092154216605}
```

(continues on next page)

(continued from previous page)

```
{'x': 1, 'y': 1.0, 'my_data': 0.9253018251769569}
{'x': 2, 'y': 0.0, 'my_data': 0.881089486629102}
{'x': 2, 'y': 0.5, 'my_data': 0.3897577898200387}
{'x': 2, 'y': 1.0, 'my_data': 0.6895312744116066}
```

Nested sweeps can be as complex as needed, with as many actions as they need. An example of this can be executing measurements on each nested level:

```
>>> def measure_something():
>>>     return np.random.rand()
>>>
>>> sweep_1 = sweep_parameter('x', range(3), record_as(measure_something, 'a'))
>>> sweep_2 = sweep_parameter('y', range(2), record_as(measure_something, 'b'))
>>> my_sweep = sweep_1 @ sweep_2 @ record_as(get_random_number, 'more_data')
>>>
>>> for data in my_sweep:
>>>     print(data)
{'x': 0, 'a': 0.09522178419462424, 'y': 0, 'b': 0.1821505218348034, 'more_data': 0.
↪13257002268089835}
{'x': 0, 'a': 0.09522178419462424, 'y': 1, 'b': 0.014940266372080457, 'more_data': 0.
↪9460879863404558}
{'x': 1, 'a': 0.13994892182170526, 'y': 0, 'b': 0.4708657480125388, 'more_data': 0.
↪12792337523097086}
{'x': 1, 'a': 0.13994892182170526, 'y': 1, 'b': 0.8209492135277935, 'more_data': 0.
↪23270477191895111}
{'x': 2, 'a': 0.06159208933324678, 'y': 0, 'b': 0.651545802505077, 'more_data': 0.
↪8944257582518365}
{'x': 2, 'a': 0.06159208933324678, 'y': 1, 'b': 0.9064557565446919, 'more_data': 0.
↪8258102740474211}
```

Note: All operators symbols are just there for syntactic brevity. All three of them have corresponding functions attached to them:

- Appending: *append_sweeps()*
- Multiplying: *zip_sweeps()*
- Nesting: *nest_sweeps()*

Passing Parameters in a Sweep

Often times our measurement actions depend on the states of previous steps. Because of that, everything that is generated by **pointers**, **actions** or other Sweeps can be passed on subsequently executed elements.

Note: There are two different Sweep configuration related to passing arguments in Sweeps. For more information on them see the *Configuring Sweeps*.

Positional Arguments

When there are no record annotations, the values generated *only* by **pointers** are passed as positional arguments to all actions, but values generated by **actions** are not passed to other **actions**:

```
>>> def test(*args, **kwargs):
>>>     print('test:', args, kwargs)
>>>     return 101
>>>
>>> def test_2(*args, **kwargs):
>>>     print('test_2:', args, kwargs)
>>>     return 102
>>>
>>> for data in Sweep(range(3), test, test_2):
>>>     print(data)
test: (0,) {}
test_2: (0,) {}
{}
test: (1,) {}
test_2: (1,) {}
{}
test: (2,) {}
test_2: (2,) {}
{}
```

Because it would get too confusing otherwise, positional arguments only get passed when originating from a **pointer** to all **actions** in a single sweep. Meaning that if we combine two or more sweeps, positional arguments would only get to the **actions** of their respective Sweeps:

```
>>> for data in Sweep(range(3), test) * Sweep(zip(['x', 'y'], [True, False]), test):
>>>    print(data)
(0,) {}
('x', True) {}
{}
(1,) {}
('y', False) {}
{}
(2,) {}
```

As we can see the `test` function in the second sweep is only getting (`x`, `True`) or (`y`, `False`) but not any arguments from the first Sweep. It is also important to note that hte values generated by either `test` function are not being passed to any other object.

In previous examples, the functions we used were accepting the arguments because their signature included variation positional arguments (`*args`). The situation changes when this is not the case. **Actions** only receive arguments that they can accept:

```
>>> def test_3(x=10):
>>>     print(x)
>>>     return True
>>>
>>> for data in Sweep(zip([1,2], [3,4]), test_3):
>>>     pass
1
```

(continues on next page)

(continued from previous page)

```
2
```

As we can see, `test_3` only accepted the first argument.

Keyword Arguments

Passing keyword arguments is more flexible. Any **record** that gets produced is passed to all subsequent **pointers** or **actions** in the sweep that accept that keyword. This is true even across different sub-sweeps. If a **pointer** yields non-annotated values, these are still used as positional arguments, but only when accepted, and with higher priority given to keywords.

In the following example we can see this in action:

```
>>> def test(x, y, z=5):
>>>     print(f'my three arguments, x: {x}, y: {y}, z: {z}')
>>>     return x, y, z
>>>
>>> def print_all_args(*args, **kwargs):
>>>     print(f'arguments at the end of the line, args: {args}, kwargs: {kwargs}')
>>>
>>> sweep = sweep_parameter('x', range(3), record_as(test, dep('xx'), dep('yy'), dep('zz
↪'))) * \
>>>         Sweep(range(3), print_all_args)
>>> for data in sweep:
>>>     pass
my three arguments, x: 0, y: None, z: 5
arguments at the end of the line, args:(0,), kwargs:{'x': 0, 'xx': 0, 'zz': 5}
my three arguments, x: 1, y: None, z: 5
arguments at the end of the line, args:(1,), kwargs:{'x': 1, 'xx': 1, 'zz': 5}
my three arguments, x: 2, y: None, z: 5
arguments at the end of the line, args:(2,), kwargs:{'x': 2, 'xx': 2, 'zz': 5}
```

In the example above we have two different sweeps. The **pointer** of the first one is producing **records** which is why we see its value in the test function for `x`. Since the first sweep is being multiplied to the second sweep we can see how all the **records** (both produced by the **pointer** and **action**) of the first sweep reach the second sweep as keyword arguments, and the non-annotated value of its own **pointer** reaches the action of the second sweep as a positional argument.

> **Warning:** When creating **records**, it is very important that each **record** has a *unique* name. Having multiple variables create **records** with the same names, will make the passing of arguments behave in unpredictable ways.

A simple way of renaming conflicting arguments and **records** is to use the combination of `lambda` and *`record_as()`*:

```
>>> sweep = (
>>>     Sweep(record_as(zip(range(3), range(10,13)), independent('x'), independent('y')),
↪ record_as(test, dependent('xx'), dependent('yy'), dependent('zz')))
>>>     @ record_as(lambda xx, yy, zz: test(xx, yy, zz), dependent('some'), dependent(
↪'different'), dependent('names'))
>>>     @ print_all_args
>>>     + print_all_args)
>>>
```

(continues on next page)

(continued from previous page)

```
>>> print(sweep.get_data_specs())
>>>
>>> for data in sweep:
>>>     print("data:", data)
(x, y, xx(x, y), yy(x, y), zz(x, y), some(x, y), different(x, y), names(x, y))
my three arguments: 0 10 5
my three arguments: 0 10 5
arguments at the end of the line: () {'x': 0, 'y': 10, 'xx': 0, 'yy': 10, 'zz': 5, 'some
↪': 0, 'different': 10, 'names': 5}
data: {'x': 0, 'y': 10, 'xx': 0, 'yy': 10, 'zz': 5, 'some': 0, 'different': 10, 'names':␣
↪5}
my three arguments: 1 11 5
my three arguments: 1 11 5
arguments at the end of the line: () {'x': 1, 'y': 11, 'xx': 1, 'yy': 11, 'zz': 5, 'some
↪': 1, 'different': 11, 'names': 5}
data: {'x': 1, 'y': 11, 'xx': 1, 'yy': 11, 'zz': 5, 'some': 1, 'different': 11, 'names':␣
↪5}
my three arguments: 2 12 5
my three arguments: 2 12 5
arguments at the end of the line: () {'x': 2, 'y': 12, 'xx': 2, 'yy': 12, 'zz': 5, 'some
↪': 2, 'different': 12, 'names': 5}
data: {'x': 2, 'y': 12, 'xx': 2, 'yy': 12, 'zz': 5, 'some': 2, 'different': 12, 'names':␣
↪5}
arguments at the end of the line: () {'x': 2, 'y': 12, 'xx': 2, 'yy': 12, 'zz': 5, 'some
↪': 2, 'different': 12, 'names': 5}
data: {}
```

Configuring Sweeps

The class *Sweep* has three global parameters that are used to configure the behaviour of it.

record_none

`Sweep.record_none` , `True` by default, adds `None` to any **action** or **pointer** that didn't generate any **record** that iteration. This is useful if we want every variable we are storing to be composed of arrays of the same number of items:

```
>>> def get_random_number():
>>>     return np.random.rand()
>>>
>>> sweep_1 = sweep_parameter('x', range(3), record_as(get_random_number, dependent('y
↪')))
>>> sweep_2 = sweep_parameter('a', range(4), record_as(get_random_number, dependent('b
↪')))
>>> my_sweep = sweep_1 + sweep_2
>>>
>>> Sweep.record_none = False
>>> print(f'----record_none=False----')
>>> for data in my_sweep:
>>>     print(data)
>>>
```

(continues on next page)

(continued from previous page)

```
>>> Sweep.record_none = True
>>> print(f'----record_none=True----')
>>> for data in my_sweep:
>>>     print(data)
----record_none=False----
{'x': 0, 'y': 0.804635124804199}
{'x': 1, 'y': 0.24410055642545125}
{'x': 2, 'y': 0.10828652013926787}
{'a': 0, 'b': 0.4303128288315823}
{'a': 1, 'b': 0.9498154942316515}
{'a': 2, 'b': 0.7150406031589893}
{'a': 3, 'b': 0.2012281139956017}
----record_none=True----
{'x': 0, 'y': 0.22753548379033073, 'a': None, 'b': None}
{'x': 1, 'y': 0.9024597689210428, 'a': None, 'b': None}
{'x': 2, 'y': 0.11393941613249503, 'a': None, 'b': None}
{'x': None, 'y': None, 'a': 0, 'b': 0.8678669225696442}
{'x': None, 'y': None, 'a': 1, 'b': 0.3537275760737344}
{'x': None, 'y': None, 'a': 2, 'b': 0.23555393946522196}
{'x': None, 'y': None, 'a': 3, 'b': 0.19388827122308672}
```

pass_on_returns

`Sweep.pass_on_returns` , `True` by default, specifies if we want arguments to be passed between sweeps. When it is set to `False` no **record** will be passed either as positional arguments or as keyword arguments:

```
>>> sweep = (
>>>     sweep_parameter('y', range(3), record_as(test, dependent('xx'), dependent('yy'),␣
↪dependent('zz')))
>>>     @ print_all_args)
>>>
>>> Sweep.pass_on_returns = False
>>> print(f'----pass_on_returns=False----')
>>> for data in sweep:
>>>     print("data:", data)
>>>
>>> Sweep.pass_on_returns = True
>>> print(f'----pass_on_returns=True----')
>>> for data in sweep:
>>>     print("data:", data)
----pass_on_returns=False----
my three arguments: None None 5
arguments at the end of the line: () {}
data: {'y': 0, 'xx': None, 'yy': None, 'zz': 5}
my three arguments: None None 5
arguments at the end of the line: () {}
data: {'y': 1, 'xx': None, 'yy': None, 'zz': 5}
my three arguments: None None 5
arguments at the end of the line: () {}
data: {'y': 2, 'xx': None, 'yy': None, 'zz': 5}
----pass_on_returns=True----
```

(continues on next page)

(continued from previous page)

```
my three arguments: None 0 5
arguments at the end of the line: () {'zz': 5, 'y': 0, 'yy': 0}
data: {'y': 0, 'xx': None, 'yy': 0, 'zz': 5}
my three arguments: None 1 5
arguments at the end of the line: () {'zz': 5, 'y': 1, 'yy': 1}
data: {'y': 1, 'xx': None, 'yy': 1, 'zz': 5}
my three arguments: None 2 5
arguments at the end of the line: () {'zz': 5, 'y': 2, 'yy': 2}
data: {'y': 2, 'xx': None, 'yy': 2, 'zz': 5}
```

pass_on_none

`Sweep.pass_on_none`, `False` by default, specifies if variables that return `None` should be passed as arguments to other **actions** or Sweeps (Because `None` is typically indicating that function did not return anything as data even though a record was declared using *recording* or *record_as()*):

```
>>> sweep = (
>>>     sweep_parameter('y', range(3), record_as(test, dependent('xx'), dependent('yy'),␣
↪dependent('zz')))
>>>     @ print_all_args)
>>>
>>> Sweep.pass_on_none = False
>>> print(f'----pass_on_none=False----')
>>> for data in sweep:
>>>     print("data:", data)
>>>
>>> Sweep.pass_on_none = True
>>> print(f'----pass_on_returns=True----')
>>> for data in sweep:
>>>     print("data:", data)
----pass_on_none=False----
my three arguments: None 0 5
arguments at the end of the line: () {'y': 0, 'yy': 0, 'zz': 5}
data: {'y': 0, 'xx': None, 'yy': 0, 'zz': 5}
my three arguments: None 1 5
arguments at the end of the line: () {'y': 1, 'yy': 1, 'zz': 5}
data: {'y': 1, 'xx': None, 'yy': 1, 'zz': 5}
my three arguments: None 2 5
arguments at the end of the line: () {'y': 2, 'yy': 2, 'zz': 5}
data: {'y': 2, 'xx': None, 'yy': 2, 'zz': 5}
----pass_on_returns=True----
my three arguments: None 0 5
arguments at the end of the line: (None,) {'y': 0, 'yy': 0, 'zz': 5, 'xx': None}
data: {'y': 0, 'xx': None, 'yy': 0, 'zz': 5}
my three arguments: None 1 5
arguments at the end of the line: (None,) {'y': 1, 'yy': 1, 'zz': 5, 'xx': None}
data: {'y': 1, 'xx': None, 'yy': 1, 'zz': 5}
my three arguments: None 2 5
arguments at the end of the line: (None,) {'y': 2, 'yy': 2, 'zz': 5, 'xx': None}
data: {'y': 2, 'xx': None, 'yy': 2, 'zz': 5}
```

Running Sweeps

As seen in previous examples, the most basic way of running a Sweep is to just iterate through it. This is simple but does not do much else. If we only want to store the data generated by a Sweep in disk for later analysis we can use the function *run_and_save_sweep()*:

```
>>> sweep = sweep_parameter('x', range(3), record_as(my_func, 'y'))
>>> run_and_save_sweep(sweep, './data', 'my_data')
Data location:  data/2022-12-05/2022-12-05T142539_fbfce3e4-my_data/data.ddh5
The measurement has finished successfully and all of the data has been saved.
```

run_and_save_sweep() automatically runs the Sweep indicated, stores the **records** generated by it in a *DataDict* in a ddh5 file with time tag followed by a random sequence followed by the third argument, in the directory passed by the second argument. Internally the function utilizes the *DDH5Writer* from *plottr*. For more information on how *plottr* handles data please see: *Data formats*.

Note: *run_and_save_sweep()* can save multiple objects to disk by accepting them as extra arguments. It is a good idea to read over its documentation if you want to be able to save things with it.

Sometimes we have an **action** that we want to run a single time, some kind of setup function or maybe a closing function (or any single **action** in between sweeps). If we also need this **action** to be a Sweep, the function *once()* will create a Sweep with no **pointer** that runs an **action** a single time:

```
>>> def startup_function():
>>>     print(f'starting an instrument')
>>>
>>> def closing_function():
>>>     print(f'closing an instrument')
>>>
>>> sweep = sweep_parameter('x', range(3), record_as(my_func, 'y'))
>>> starting_sweep = once(startup_function)
>>> closing_sweep = once(closing_function)
>>>
>>> for data in starting_sweep + sweep + closing_sweep:
>>>     print(data)
starting an instrument
{}
{'x': 0, 'y': 1}
{'x': 1, 'y': 1}
{'x': 2, 'y': 1}
closing an instrument
{}
```

Reference

Sweep

class labcore.measurement.sweep.**PointerFunction**(*func*, **data_specs*)

A class that allows using a generator function as a pointer.

using(**args*, ***kwargs*) → *PointerFunction*

Set the default positional and keyword arguments that will be used when the function is called.

Returns
A copy of the object. This is to allow setting different defaults to multiple uses of the function.

labcore.measurement.sweep.**pointer**(**data_specs: Union[str, Tuple[str, Union[None, List[str], Tuple[str]], str, str], Dict[str, Union[str, None, List[str], Tuple[str]]], DataSpec]*) → Callable

Create a decorator for functions that return pointer generators.

labcore.measurement.sweep.**as_pointer**(*fun: Callable*, **data_specs: Union[str, Tuple[str, Union[None, List[str], Tuple[str]], str, str], Dict[str, Union[str, None, List[str], Tuple[str]]], DataSpec]*) → *PointerFunction*

Convenient in-line creation of a pointer function.

labcore.measurement.sweep.**once**(*action: Callable*) → *Sweep*

Return a sweep that executes the action once.

labcore.measurement.sweep.**sweep_parameter**(*param: Union[str, Parameter, Tuple[str, Union[None, List[str], Tuple[str]], str, str], DataSpec]*, *sweep_iterable: Iterable*, **actions: Callable*) → *Sweep*

Create a sweep over a parameter.

Parameters

- **param** – One of:
  - A string: Generates an independent, scalar data parameter.
  - A tuple or list: will be passed to the constructor of *DataSpec*; see *make_data_spec()*.
  - A *DataSpec* instance.
  - A qcodes parameter. In this case the parameter's `set` method is called for each value during the iteration.
- **sweep_iterable** – An iterable that generates the values the parameter will be set to.
- **actions** – An arbitrary number of action functions.

class labcore.measurement.sweep.**Sweep**(*pointer: Optional[Iterable]*, **actions: Callable*)

Base class for sweeps.

Can be iterated over; for each pointer value the associated actions are executed. Each iteration step produces a record, containing all values produced by pointer and actions that have been annotated as such. (see: *record_as()*)

Parameters

- **pointer** – An iterable that defines the steps over which we iterate
- **actions** – A variable number of functions. Each will be called for each iteration step, with the pointer value(s) as arguments, provided the function can accept it.

static update_option_dict(*src: Dict[str, Any]*, *target: Dict[str, Any]*, *level: int*) → None

Rules: work in progress :).

static link_sweep_properties(*src:* Sweep, *target:* Sweep) → None

Share state properties between sweeps.

append_action(*action: Callable*)

Add an action to the sweep.

run() → *SweepIterator*

Create the iterator for the sweep.

set_options(***action_kwargs: Dict[str, Any]*)

Configure the sweep actions.

Parameters

action_kwargs – Keyword arguments to pass to action functions format: {'<action_name>': {'key': 'value'} <action_name> is what action_function.__name__ returns.

get_data_specs() → Tuple[*DataSpec*, ...]

Return the data specs of the sweep.

class labcore.measurement.sweep.**SweepIterator**(*sweep:* Sweep, *state: Dict[str, Any]*, *pass_kwargs=typing.Dict[str, typing.Any]*, *action_kwargs=typing.Dict[str, typing.Dict[str, typing.Any]]*)

Iterator for the *Sweep* class.

Manages the actual iteration of the pointer, and the execution of action functions. Manages and updates the state of the sweep.

labcore.measurement.sweep.**append_sweeps**(*first:* Sweep, *second:* Sweep) → *Sweep*

Append two sweeps.

Iteration over the combined sweep will first complete the first sweep, then the second sweep.

labcore.measurement.sweep.**zip_sweeps**(*first:* Sweep, *second:* Sweep) → *Sweep*

Zip two sweeps.

Iteration over the combined sweep will elementwise advance the two sweeps together.

labcore.measurement.sweep.**nest_sweeps**(*outer:* Sweep, *inner:* Sweep) → *Sweep*

Nest two sweeps.

Iteration over the combined sweep will execute the full inner sweep for each iteration step of the outer sweep.

class labcore.measurement.sweep.**AsyncRecord**(**specs*)

Base class decorator used to record asynchronous data from instrument. Use the decorator with create_background_sweep function to create Sweeps that collect asynchronous data from external devices running experiments independently of the measurement PC, e.i. the measuring happening is not being controlled by a Sweep but instead an external device (e.g. the OPX). Each instrument should have its own custom setup_wrapper (see setup_wrapper docstring for more info), and a custom collector. Auxiliary functions for the start_wrapper and collector should also be located in this class.

Parameters

specs – A list of the DataSpecs to record the data produced.

wrap_setup(*fun: Callable*, **args: Any*, ***kwargs: Any*) → Callable

Wraps the start function. setup_wrapper should consist of another function inside of it decorated with @wraps with fun as its argument. In this case the wrapped function is setup. Setup should accept the *args and **kwargs of fun. It should also place any returns from fun in the communicator. setup_wrapper needs to return the wrapped function (setup).

Parameters

fun – The measurement function. In the case of the OPX this would be the function that returns the QUA code with any arguments that it might use.

Record

class labcore.measurement.record.**DataType**(*value*)

Valid options for data types used in *DataSpec*

scalar = 'scalar'

scalar (single-valued) data. typically numeric, but also bool, etc.

array = 'array'

multi-valued data. typically numpy-arrays.

class labcore.measurement.record.**DataSpec**(*name: str*, *depends_on: Union[None, List[str], Tuple[str]] = None*, *type: Union[str,* DataType*] = 'scalar'*, *unit: str = ''*)

Specification for data parameters to be recorded.

name: str

name of the parameter

depends_on: Union[None, List[str], Tuple[str]] = None

dependencies. if None, it is independent.

type: Union[str, *DataType*] = 'scalar'

information about data format

unit: str = ''

physical unit of the data

copy() → *DataSpec*

return a deep copy of the DataSpec instance.

labcore.measurement.record.**ds**

shorter notation for constructing DataSpec objects

labcore.measurement.record.**DataSpecFromTupleType**

The type for creating a ds from a tuple (i.e., what can be passed to the constructor of *DataSpec*)

alias of Tuple[str, Union[None, List[str], Tuple[str]], str, str]

labcore.measurement.record.**DataSpecFromDictType**

The type for creating a ds from a dict (i.e., what can be passed to the constructor of *DataSpec* as keywords)

alias of Dict[str, Union[str, None, List[str], Tuple[str]]]

labcore.measurement.record.**DataSpecCreationType**

The type from which we can create a DataSpec.

alias of Union[str, Tuple[str, Union[None, List[str], Tuple[str]], str, str], Dict[str, Union[str, None, List[str], Tuple[str]]], *DataSpec*]

labcore.measurement.record.**data_specs_label**(**dspecs:* DataSpec) → str

Create a readable label for multiple data specs.

Format:
{data_name_1 (dep_1, dep_2), data_name_2 (dep_3), etc.}

Parameters
dspecs – data specs as positional arguments.

Returns
label as string.

labcore.measurement.record.**make_data_spec**(*value: Union[str, Tuple[str, Union[None, List[str], Tuple[str]], str, str], Dict[str, Union[str, None, List[str], Tuple[str]]], DataSpec]*) → *DataSpec*

Instantiate a DataSpec object.

Parameters
value – May be one of the following with the following behavior:

- A string create a dependent with name given by the string
- A tuple of values that can be used to pass to the constructor of *DataSpec*
- A dictionary entries of which will be passed as keyword arguments to the constructor of *DataSpec*
- A *DataSpec* instance

labcore.measurement.record.**make_data_specs**(**specs: Union[str, Tuple[str, Union[None, List[str], Tuple[str]], str, str], Dict[str, Union[str, None, List[str], Tuple[str]]], DataSpec]*) → Tuple[*DataSpec*, ...]

Create a tuple of DataSpec instances.

Parameters
specs – will be passed individually to *make_data_spec()*

labcore.measurement.record.**combine_data_specs**(**specs:* DataSpec) → Tuple[*DataSpec*, ...]

Create a tuple of DataSpecs from the inputs. Removes duplicates.

labcore.measurement.record.**independent**(*name: str*, *unit: str = ''*, *type: str = 'scalar'*) → *DataSpec*

Create a the spec for an independent parameter. All arguments are forwarded to the *DataSpec* constructor. depends_on is set to None.

labcore.measurement.record.**indep**(*name: str*, *unit: str = ''*, *type: str = 'scalar'*) → *DataSpec*

Create a the spec for an independent parameter. All arguments are forwarded to the *DataSpec* constructor. depends_on is set to None.

labcore.measurement.record.**dependent**(*name: str*, *depends_on: List[str] = []*, *unit: str = ''*, *type: str = 'scalar'*)

Create a the spec for a dependent parameter. All arguments are forwarded to the *DataSpec* constructor. depends_on may not be set to None.

labcore.measurement.record.**dep**(*name: str*, *depends_on: List[str] = []*, *unit: str = ''*, *type: str = 'scalar'*)

Create a the spec for a dependent parameter. All arguments are forwarded to the *DataSpec* constructor. depends_on may not be set to None.

labcore.measurement.record.**recording**(**data_specs: Union[str, Tuple[str, Union[None, List[str], Tuple[str]], str, str], Dict[str, Union[str, None, List[str], Tuple[str]]], DataSpec]*) → Callable

Returns a decorator that allows adding data parameter specs to a function.

labcore.measurement.record.**record_as**(*obj: Union[Callable, Iterable, Iterator]*, **specs: Union[str, Tuple[str, Union[None, List[str], Tuple[str]], str, str], Dict[str, Union[str, None, List[str], Tuple[str]]], DataSpec]*)

Annotate produced data as records.

Parameters

- **obj** – a function that returns data or an iterable/iterator that produces data at each iteration step
- **specs** – specs for the data produced (see *make_data_specs()*)

labcore.measurement.record.**produces_record**(*obj: Any*) → bool

Check if *obj* is annotated to generate records.

class labcore.measurement.record.**IteratorToRecords**(*iterable: Iterable*, **data_specs: Union[str, Tuple[str, Union[None, List[str], Tuple[str]], str, str], Dict[str, Union[str, None, List[str], Tuple[str]]], DataSpec]*)

A wrapper that converts the iteration values to records.

class labcore.measurement.record.**FunctionToRecords**(*func*, **data_specs*)

A wrapper that converts a function return to a record.

using(**args*, ***kwargs*) → *FunctionToRecords*

Set the default positional and keyword arguments that will be used when the function is called.

Returns
a copy of the object. This is to allow setting different defaults to multiple uses of the function.

ddh5

plottr.data.datadict_storage

Provides file-storage tools for the DataDict class.

Description of the HDF5 storage format

We use a simple mapping from DataDict to the HDF5 file. Within the file, a single DataDict is stored in a (top-level) group of the file. The data fields are datasets within that group.

Global meta data of the DataDict are attributes of the group; field meta data are attributes of the dataset (incl., the *unit* and *axes* values). The meta data keys are given exactly like in the DataDict, i.e., incl the double underscore pre- and suffix.

class labcore.ddh5.**NumpyEncoder**(**, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, sort_keys=False, indent=None, separators=None, default=None*)

default(*obj*)

Implement this method in a subclass such that it returns a serializable object for `o`, or calls the base implementation (to raise a `TypeError`).

For example, to support arbitrary iterators, you could implement default like this:

```
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
        return list(iterable)
    # Let the base class default method raise the TypeError
    return JSONEncoder.default(self, o)
```

labcore.ddh5.**run_and_save_sweep**(*sweep: Sweep*, *data_dir: str*, *name: str*, *ignore_all_None_results: bool = True*, *save_action_kwargs: bool = False*, *add_timestamps=False*, *archive_files: Optional[List[str]] = None*, ***extra_saving_items*) → None

Iterates through a sweep, saving the data coming through it into a file called <name> at <data_dir> directory.

Parameters

- **sweep** – Sweep object to iterate through.
- **data_dir** – Directory of file location.
- **name** – Name of the file.
- **ignore_all_None_results** – if `True`, don't save any records that contain a `None`. if `False`, only do not save records that are all-`None`.
- **save_action_kwargs** – If `True`, the action_kwargs of the sweep will be saved as a json file named after the first key of the kwargs dctionary followed by '_action_kwargs' in the same directory as the data.
- **archive_files** – List of files to copy into a folder called 'archived_files' in the same directory that the data is saved. It should be a list of paths (str), regular expressions are supported. If a folder is passed, it will copy the entire folder and all of its subdirectories and files into the archived_files folder. If one of the arguments could not be found, a message will be printed and the measurement will be performed without the file being archived. An exception is raised if the type is invalid.

  e.g. archive_files=['*.txt', 'calibration_files', '../test_file.py'].* '.txt' will copy every txt file located in the working directory. 'calibration_files' will copy the entire folder called calibration_files from the working directory into the archived_files folder. '../test_file.py' will copy the script test_file.py from one directory above the working directory.
- **extra_saving_items** – Kwargs for extra objects that should be saved. If the kwarg is a dictionary, the function will try and save it as a JSON file. If the dictionary contains objects that are not JSON serializable it will be pickled. Any other kind of object will be pickled too. The files will have their keys as names.

Raises

TypeError – A Typerror is raised if the object passed for archive_files is not correct

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

i

l

p

INDEX

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Z