Pysweep 2.0

A quick tour

Table of contents

Introduction; why pysweep?

- While QCoDeS has a loop mechanism build-in, scientists and PhD students often find this too limiting
- We can only sweep QCoDeS parameters and while we have nested loops, we cannot nest arbitrary measurement functions in the nest. For example, how do we do

```
for sweep1:
     some_measurement
     for sweep2:
           another_measurement
```

  in QCoDeS loops?

- There is no support for complex loops with feed backs

Guiding philosophy behind pysweep

- Intended to be flexible enough for most things
- It is intended to work well in an environment of rapidly changing code where the code itself is not the primary concern; an experiment is
  - Get rid of as much “boiler plate” code as possible to keep things succinct, even with complex measurements with hardware sweeps and feedbacks
  - We should be able to quickly see what an experiment does
  - We need to be able to quickly re-use bits of code from elsewhere
  - Sometimes we need different ways to accomplish the same thing as user preferences differ.

Part 1: the sweep object (1/2)

```
so = sweep(power_source1.voltage, [0, 1, 2])

for data_entry in so:
    print(data_entry)
```

```
OrderedDict([('power_source1_voltage', {'unit': 'V', 'value': 0, 'independent_parameter': True})])
OrderedDict([('power_source1_voltage', {'unit': 'V', 'value': 1, 'independent_parameter': True})])
OrderedDict([('power_source1_voltage', {'unit': 'V', 'value': 2, 'independent_parameter': True})])
```

The signature of a sweep object:

Sweep_object = sweep(consumer, producer)

At every iteration we take the next value from the producer and pass it on to the consumer. In the Examples of this presentation the producers will often be simple lists of values, e.g. [0, 1, 2]. The basic structure of a loop is very similar to what QCoDeS does already. However, some key innovations of pysweep are:

- Producers can also be other objects (e.g. A Python generator) which produce values which can depend on measurements. This enables us to for example create feedback loops and make adaptive steps.
- A consumer does not have to be a QCoDeS parameter.

Part 1: the sweep object (2/2)

```
so = sweep(power_source1.voltage, [0, 1, 2])

for data_entry in so:
    print(data_entry)
```

```
OrderedDict([('power_source1_voltage', {'unit': 'V', 'value': 0, 'independent_parameter': True})])
OrderedDict([('power_source1_voltage', {'unit': 'V', 'value': 1, 'independent_parameter': True})])
OrderedDict([('power_source1_voltage', {'unit': 'V', 'value': 2, 'independent_parameter': True})])
```

Notes:

1) Looping over a sweep object will give us a new data entry to put in our data set (and eventually database)
2) When a QCoDeS parameter is the consumer, the data key for every entry will be <instrument>_<parameter> and the parameters will be registered as “independent”

On the next slide we will see an example where the consumer is not an QCoDeS parameter

We can sweep functions as well as parameters

```python
def some_set_function(station, namespace, value):
    station.power_source1.voltage(value)
    fridge_temperature = get_fridge_temperature()

    return {"gate1": {"unit": "V", "value": value, "independent_parameter": True},
            "fridge_temperature": {"unit": "K", "value": fridge_temperature}}
```

```python
so = sweep(some_set_function, [0, 1, 2])

for data_entry in so:
    pprint.pprint(data_entry)
```

```
{'fridge_temperature': {'unit': 'K', 'value': 0.20245099089024307},
 'gate1': {'independent_parameter': True, 'unit': 'V', 'value': 0}}
{'fridge_temperature': {'unit': 'K', 'value': 0.1960185934886391},
 'gate1': {'independent_parameter': True, 'unit': 'V', 'value': 1}}
{'fridge_temperature': {'unit': 'K', 'value': 0.2080837345017436},
 'gate1': {'independent_parameter': True, 'unit': 'V', 'value': 2}}
```

Notes:

1) If the consumer is a set function, this needs to be a function of (station, namespace, value)
2) When the user specifies a set function then (s)he is responsible for correct data formatting and identifying independent parameters
3) The meaning of “namespace” in the set function will be clarified in slides to come

Nesting sweeps

```
so1 = sweep(power_source1.voltage, [0, 1, 2])
so2 = sweep(power_source2.voltage, [2, 4, 6])

nested_so = so1(so2)

for data_entry in nested_so:
    pprint.pprint(data_entry)
```

```
OrderedDict([('power_source2_voltage',
               {'independent_parameter': True, 'unit': 'V', 'value': 2}),
              ('power_source1_voltage',
               {'independent_parameter': True, 'unit': 'V', 'value': 0})])
OrderedDict([('power_source2_voltage',
               {'independent_parameter': True, 'unit': 'V', 'value': 4}),
              ('power_source1_voltage',
               {'independent_parameter': True, 'unit': 'V', 'value': 0})])
OrderedDict([('power_source2_voltage',
               {'independent_parameter': True, 'unit': 'V', 'value': 6}),
              ('power_source1_voltage',
               {'independent_parameter': True, 'unit': 'V', 'value': 0})])
OrderedDict([('power_source2_voltage',
               {'independent_parameter': True, 'unit': 'V', 'value': 2}),
              ('power_source1_voltage',
               {'independent_parameter': True, 'unit': 'V', 'value': 1})])
```

Nesting two sweeps: `nested_so = so1(so2)`

Nesting measurements

Of course in a useful experiment we want to vary the independent parameters and perform a measurement at each set point. That is, we want to nest some measurement in a sweep object

```
so = sweep(power_source1.voltage, [0, 1, 2])

nested_so = so1(station.multi_meter.current)

for data_entry in nested_so:
    pprint.pprint(data_entry)
```

```
OrderedDict([('multi_meter_current', {'unit': 'A', 'value': 0}),
             ('power_source1_voltage',
              {'independent_parameter': True, 'unit': 'V', 'value': 0})])
OrderedDict([('multi_meter_current', {'unit': 'A', 'value': 1}),
             ('power_source1_voltage',
              {'independent_parameter': True, 'unit': 'V', 'value': 1})])
OrderedDict([('multi_meter_current', {'unit': 'A', 'value': 4}),
             ('power_source1_voltage',
              {'independent_parameter': True, 'unit': 'V', 'value': 2})])
```

Nesting a sweep and measuremnt:

nested_so = so1(measurement)

Note that in general we will write in this presentation: Nested_so = so1(so2) where so2 can be either a sweep object or a measurement

We can have as many nests as we want

```
so1 = sweep(power_source1.voltage, [0, 1, 2])
so2 = sweep(power_source2.voltage, [2, 4, 6])

nested_so = so1(so2(station.multi_meter.current))

for data_entry in nested_so:
    pprint.pprint(data_entry)
```

```
OrderedDict([('multi_meter_current', {'unit': 'A', 'value': 4}),
             ('power_source2_voltage',
              {'independent_parameter': True, 'unit': 'V', 'value': 2}),
             ('power_source1_voltage',
              {'independent_parameter': True, 'unit': 'V', 'value': 0})])
OrderedDict([('multi_meter_current', {'unit': 'A', 'value': 16}),
             ('power_source2_voltage',
              {'independent_parameter': True, 'unit': 'V', 'value': 4}),
             ('power_source1_voltage',
              {'independent_parameter': True, 'unit': 'V', 'value': 0})])
OrderedDict([('multi_meter_current', {'unit': 'A', 'value': 36}),
             ('power_source2_voltage',
```

Here we have a tripple nest where the inner most nest is a measurement. We will thus produce a 3D data set of two independent parameters and one dependent, producing for example a 2D gray value picture.

Instead of a QCoDeS parameter we can also nest a measurement function

```python
def some_measurement_function(station, namespace):
    value = station.power_source1.voltage() + station.power_source2.voltage()
    return {"my_measurement": {"unit": "V", "value": value}}
```

```python
so1 = sweep(power_source1.voltage, [0, 1, 2])
so2 = sweep(power_source2.voltage, [2, 4, 6])

nested_so = so1(so2(some_measurement_function))

for data_entry in nested_so:
    pprint.pprint(data_entry)
```

```
{'my_measurement': {'unit': 'V', 'value': 2},
 'power_source1_voltage': {'independent_parameter': True,
                           'unit': 'V',
                           'value': 0},
 'power_source2_voltage': {'independent_parameter': True,
                           'unit': 'V',
                           'value': 2}}
{'my_measurement': {'unit': 'V', 'value': 4},
 'power_source1_voltage': {'independent_parameter': True,
                           'unit': 'V',
```

A “measurement” can either be a QCoDeS parameter or a measurement function.
If the latter, the function has to be a function of (station, namespace)

NB: As promissed, the meaning of namespace will become clear in the sections to come

More complicated nesting (1/5)

To write a measurment which in pseudo-code does this:

```
for sweep1:
      other_measurement()
      for sweep2:
            some_measurement()
```

We introduce the rule: SO1([SO2, SO3]) = SO1(SO2) + SO1(SO3)

An example is shown on the next slide

More complicated nesting (2/5)

Let's look at this code.

```
def other_measurement_function(station, namespace):
    current = station.multi_meter.current()
    power = power_source1.voltage() * current

    return {
        "other_measurement": {
            "unit": "W",
            "value": power
        }
    }
```

```
so1 = sweep(power_source1.voltage, [0, 1])
so2 = sweep(power_source2.voltage, [2, 6])

nested_so = so1([other_measurement_function, so2(some_measurement_function)])

for data_entry in nested_so:
    pprint.pprint(data_entry)
    print("\n")
```

This roughly means

```
for sweep1:
        other_measurement()
        for sweep2:
                some_measurement()
```

The output is shown on the next two slides

More complicated nesting (3/5)

```
{'other_measurement': {'unit': 'W', 'value': 0},
 'power_source1_voltage': {'independent_parameter': True,
                           'unit': 'V',
                           'value': 0}}
```

Iteration 1 of nested sweep object
Iteration 1 of SO1(other_measurement)

```
{'power_source1_voltage': {'independent_parameter': True,
                           'unit': 'V',
                           'value': 0},
 'power_source2_voltage': {'independent_parameter': True,
                           'unit': 'V',
                           'value': 2},
 'some_measurement': {'unit': 'V', 'value': 2}}
```

Iteration 2 of nested sweep object
Iteration 1,1 of SO1(SO2(some_measurement))

```
{'power_source1_voltage': {'independent_parameter': True,
                           'unit': 'V',
                           'value': 0},
 'power_source2_voltage': {'independent_parameter': True,
                           'unit': 'V',
                           'value': 6},
 'some_measurement': {'unit': 'V', 'value': 6}}
```

Iteration 3 of nested sweep object
Iteration 1,2 of SO1(SO2(some_measurement))

More complicated nesting (4/5)

```
{'other_measurement': {'unit': 'W', 'value': 37},
 'power_source1_voltage': {'independent_parameter': True,
                           'unit': 'V',
                           'value': 1}}
```

Iteration 4 of nested sweep object
Iteration 2 of SO1(other_measurement)

```
{'power_source1_voltage': {'independent_parameter': True,
                           'unit': 'V',
                           'value': 1},
 'power_source2_voltage': {'independent_parameter': True,
                           'unit': 'V',
                           'value': 2},
 'some_measurement': {'unit': 'V', 'value': 3}}
```

Iteration 5 of nested sweep object
Iteration 2,1 of SO1(SO2(some_measurement))

```
{'power_source1_voltage': {'independent_parameter': True,
                           'unit': 'V',
                           'value': 1},
 'power_source2_voltage': {'independent_parameter': True,
                           'unit': 'V',
                           'value': 6},
 'some_measurement': {'unit': 'V', 'value': 7}}
```

Iteration 6 of nested sweep object
Iteration 2,2 of SO1(SO2(some_measurement))

More complicated nesting (5/5)

To summerize: We introduced the grammar rule:

```
Nested_so = so1([other_measurement, so2(some measurement)])
```

Is roughly equal to

```
Nested_so = so1(other_measurement) + so1(so2(some_measurement)
```

But the addition should not be taken too literally. It is NOT that we are performing two loops consecutively; the set points in so1 are only traverced once. Please study the sweep output on the previous to pages to understand what is going on.

Nesting a sweep in a measurement function (1/3)

On previous slides we have seen that

```
Nested_so = so1([other_measurement, so2(some measurement)])
```

will produce an output where the results of “other_measurement” and “some_measurement” are not present in each iteration of the nested object. This can be undesirable as we might want to perform a sweep of so2 for every result of “other_measurement”. In essence, we want to nest so2 in “other_measurement” as if the latter is a sweep object itself. For this we introduce the rule:

```
so1((so2, so3)) = so1(so2(so3))
```

Where so2 and so3 can either be a sweep object or a measurement function. Let’s inspect the result of this rule on the next slides

Nesting a sweep in a measurement function (2/3)

Let's look at this code. The output is shown on the next right (continued on next slide)

```
so1 = sweep(power_source1.voltage, [0, 1])
so2 = sweep(power_source2.voltage, [2, 6])

nested_so = so1((other_measurement_function, so2(some_measurement_function)))

for data_entry in nested_so:
    pprint.pprint(data_entry)
    print("\n")
```

Notice that we are using round brackets now.
For every output of other measurement function we are performing a sweep of so2, thus the output of this function is present in each iteration.

iterations

```
{'other_measurement': {'unit': 'W', 'value': 0},
 'power_source1_voltage': {'independent_parameter': True,
                           'unit': 'V',
                           'value': 0},
 'power_source2_voltage': {'independent_parameter': True,
                           'unit': 'V',
                           'value': 2},
 'some_measurement': {'unit': 'V', 'value': 2}}
```

1

```
{'other_measurement': {'unit': 'W', 'value': 0},
 'power_source1_voltage': {'independent_parameter': True,
                           'unit': 'V',
                           'value': 0},
 'power_source2_voltage': {'independent_parameter': True,
                           'unit': 'V',
                           'value': 6},
 'some_measurement': {'unit': 'V', 'value': 6}}
```

2

Output continued on next slide

Nesting a sweep in a measurement function (3/3)

Let's look at this code. The output is shown on the next right (continued from previous slide).

```
so1 = sweep(power_source1.voltage, [0, 1])
so2 = sweep(power_source2.voltage, [2, 6])

nested_so = so1((other_measurement_function, so2(some_measurement_function)))

for data_entry in nested_so:
    pprint.pprint(data_entry)
    print("\n")
```

Notice that we are using round brackets now.
For every output of other measurement function we are performing a sweep of so2, thus the output of this function is present in each iteration.

iterations

```
{'other_measurement': {'unit': 'W', 'value': 37},
 'power_source1_voltage': {'independent_parameter': True,
                           'unit': 'V',
                           'value': 1},
 'power_source2_voltage': {'independent_parameter': True,
                           'unit': 'V',
                           'value': 2},
 'some_measurement': {'unit': 'V', 'value': 3}}
```

3

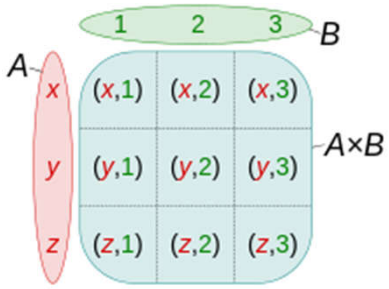
```
{'other_measurement': {'unit': 'W', 'value': 37},
 'power_source1_voltage': {'independent_parameter': True,
                           'unit': 'V',
                           'value': 1},
 'power_source2_voltage': {'independent_parameter': True,
                           'unit': 'V',
                           'value': 6},
 'some_measurement': {'unit': 'V', 'value': 7}}
```

4

Key pysweep 2.0 innovation (1/2)

To summerize the dicsussion until now; if one regards sweep objects and measurements as vectors, then creating a data set is akin to constructing a chain of sums and carthesian products of sweeps and measurements. **This is a key pysweep innovation.**

Catresian product:



(taken from wikipedia)

The next slides explains this idea in more detail.

Key pysweep 2.0 innovation (2/2)

$$\text{Sweep}\left(p, \begin{bmatrix}0\\1\\2\end{bmatrix}\right) \times [m()] = \begin{bmatrix} p=0, m() \\ p=1, m() \\ p=2, m() \end{bmatrix}$$

m() is a measurement. Colors indicate unique measurements. Pysweep is attempting to invented a grammer to encode the chains and carthesian producs in a convinient way.

$$\text{Sweep}\left(p, \begin{bmatrix}0\\1\\2\end{bmatrix}\right) \times [m()] \times \text{sweep}\left(q, \begin{bmatrix}1\\2\\4\end{bmatrix}\right) = \begin{bmatrix} p=0, m(), q=1 & p=0, m(), q=2 & p=0, m(), q=4 \\ p=1, m(), q=1 & p=1, m(), q=2 & p=1, m(), q=4 \\ p=2, m(), q=1 & p=2, m(), q=2 & p=2, m(), q=4 \end{bmatrix}$$

$$\text{Sweep}\left(p, \begin{bmatrix}0\\1\\2\end{bmatrix}\right) \times \left[[m()] + \text{sweep}\left(q, \begin{bmatrix}1\\2\\4\end{bmatrix}\right) \right] = \begin{bmatrix} p=0, m() \\ p=1, m() \\ p=2, m() \end{bmatrix} + \begin{bmatrix} p=0, q=1 & p=0, q=2 & p=0, q=4 \\ p=1, q=1 & p=1, q=2 & p=1, q=4 \\ p=2, q=1 & p=2, q=2 & p=2, q=4 \end{bmatrix}$$

Part 2: measurements and examples

We have introduced the notion of a sweep object and how mathematical operations of these and measurements can create other sweep objects.

Looping over these sweep objects will give us a data entry at each iteration. Part 2 will dicsuss how these data entries can be grouped togehter to create data tables and data sets.

The Measurement class

The signature of the measurement class is as follows:

```
measurement = Measurement(
            [setup_functions],
            [cleanup_functions],
            sweep_args
)
```

Where “sweep_args” are arguments which are used to construct a top level sweep object. For instance:

- Sweep_args=(so1, so2) will create a sweep object which is constructed from a nest of so1 and so2.
- Sweep_args=so1(so2) is equivalent to sweep_args=(so1, so2)
- Sweep_args=[so1, (so2, so3)] will create a sweep object which is a chain of so1 with the nest of so2 and so3.

Setup functions are functions which prepare instruments for data aqcuisition. Clean up functions are functions which make sure that the instruments are left in a well defined state after the measurement. Both these functions are functions of (station, namespace).

We will look at examples om the comming slides.

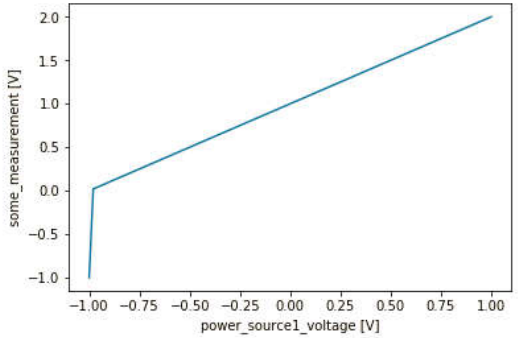
Performing a measurement I

```
measurement = Measurement(
    setup,
    cleanup,
    sweep(power_source1.voltage, np.linspace(-1, 1, 100))(
        some_measurement_function,
        sweep(power_source2.voltage, np.linspace(-1, 1, 100))(
            station.multi_meter.current,
        )
    )
)

data = measurement.run()
```

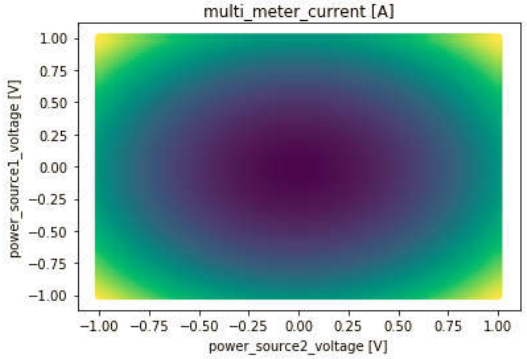
```
setting up
cleaning up
```

```
DataPlot(data.output("some_measurement"))
```




```
<__main__.DataPlot at 0x1bb26a873c8>
```

```
DataPlot(data.output("multi_meter_current"))
```



Notice the strait forward, “python-like” syntax of our measurement class

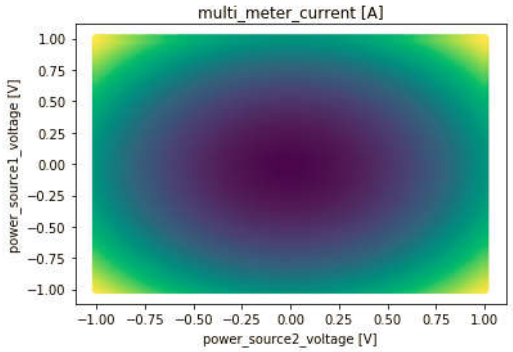
These plots may not look to special at first glance, but notice how we have not told our plotting function what to put along the axes of the plot, or even if we have a 1D or 2D data set. The next slide contains an even more poignant example

```
measurement = Measurement(
    setup,
    cleanup,
    sweep(power_source1.voltage, np.linspace(-1, 1, 100))(
        sweep(power_source2.voltage, np.linspace(-1, 1, 100))(
            station.multi_meter.current,
        ),
        sweep(power_source3.voltage, np.linspace(-2, 1, 150))(
            some_measurement_function,
        )
    )
)

data = measurement.run()
```

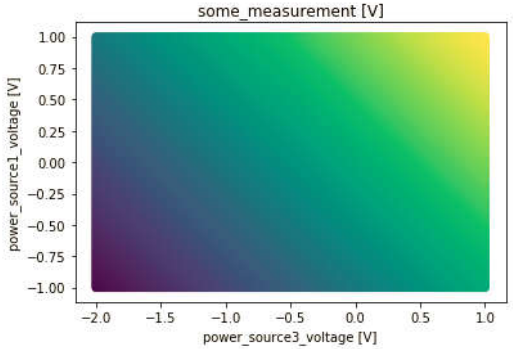
```
setting up
cleaning up
```

```
DataPlot(data.output("multi_meter_current"))
```




```
<__main__.DataPlot at 0x21914eb42b0>
```

```
DataPlot(data.output("some_measurement"))
```



Performing a measurement 2

Here is what we see:

- Here we see two nested loops within a single outer loop.
- Both the measurement of the multimeter current and “some_measurement_function” are sweeped over power source 1.
- The multimeter current is additionally sweeped over power source 2 on the second axis.
- “some_measurement_function” is sweeped over power source 3 on the second axis.
- In the plotting commands we do not tell what the independent parameters are nor the dimensionality of the data. It just “knows”.

The context of the measurement is preserved
in the data set. On the next slide we explain how this happens.

Preservation of measurement context in the dataset

Let m = “some_measurement” and n=“multi_meter_current” from the last slide. Additionally, p = “power_source1.voltage” and q = “power_source2.voltage”. We can write down the measurement from the previous slide as:

$$\text{Sweep}\left(p, \begin{bmatrix}0\\1\\2\end{bmatrix}\right) \times \left[[m()] + \left[\text{sweep}\left(q, \begin{bmatrix}1\\2\\4\end{bmatrix}\right) \times [n()]\right]\right] = \begin{bmatrix} p=0,\ m() \\ p=1,\ m() \\ p=2,\ m() \end{bmatrix} + \begin{bmatrix} p=0,\ q=1,\ n() & p=0,\ q=2,\ n() & p=0,\ q=4,\ n() \\ p=1,\ q=1,\ n() & p=1,\ q=2,\ n() & p=1,\ q=4,\ n() \\ p=2,\ q=1,\ n() & p=2,\ q=2,\ n() & p=2,\ q=4,\ n() \end{bmatrix}$$

Table 1 Table 2

When we give the command to plot “m” (or “some_measurement”) we lookup in which table this can be found and plot “m” with respect to the independent parameters found in that table. The same thing happens for “n”, or “multi_meter_current”. This results in the plots on the previous slide. For a more detailed explaination, see the appendix.

Some practical example I

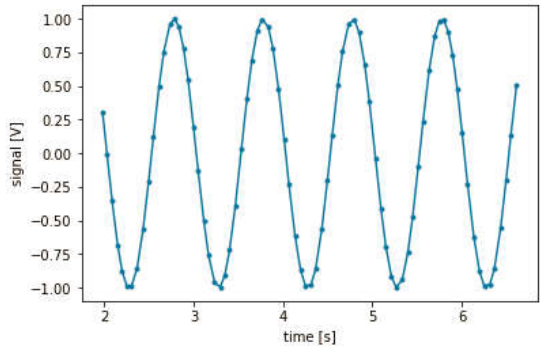
```
dt = 0.05
total_t = 4
triggers = range(int(total_t/dt))

measurement = Measurement(
    setup,
    cleanup,
    [
        sweep(lockin_amplifier.trigger, triggers)(
            log_time(),
            sleep(dt)
        ),
        get_lockin_amplifier_buffer
    ]
)
```

```
output = measurement.run()
```

```
setting up
cleaning up
```

```
plot(output, "time", "signal")
```




```
def get_lockin_amplifier_buffer(station, namespace):
    return {"signal": {"unit": "V", "value": station.lockin_amplifier.play_buffer()}}

def log_time():
    tb = time.time()

    def inner(station, namespace):
        return {"time": {"unit": "s", "value":time.time() - tb}}
    return inner
```

Let us consider a practical example. Suppose a lockin-amplifier is measuring a sinusoidal signal at 1 Hz. Rather then directly sampling the signal, we send triggers to the lockin amplifier to aqcuire the signal for us and write it to an internal instrument buffer. At the end of the loop, we want to read the buffer and plot the result.

Some practical example II

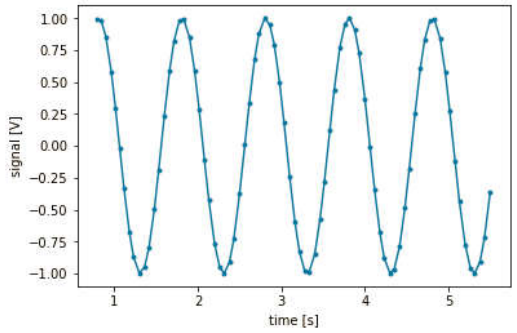
```
dt = 0.05
total_t = 4
triggers = range(int(total_t/dt))

measurement = Measurement(
    setup,
    cleanup,
    [
        sweep(lockin_amplifier.trigger, triggers)(
            log_time(),
            sleep(dt),
            get_lockin_amplifier_buffer()
        ),
        get_lockin_amplifier_buffer(force=True)
    ]
)
```

```
output = measurement.run()
```

```
setting up
cleaning up
```

```
plot(output, "time", "signal")
```




```
def setup(station, namespace):
    print("setting up")
    namespace.trigger_count = 0
    station.lockin_amplifier.play_buffer()  # empty the buffer
    return {}

def cleanup(station, namespace):
    print("cleaning up")
    return {}

def get_lockin_amplifier_buffer(force=False):

    def inner (station, namespace):

        perform_read = False
        namespace.trigger_count += 1
        if namespace.trigger_count == LockinAmplifier.max_buffer_size:
            perform_read = True
            namespace.trigger_count = 0

        if force or perform_read:
            buffer = station.lockin_amplifier.play_buffer()
            return {"signal": {"unit": "V", "value": buffer}}
        else:
            return {}

    return inner

def log_time():
    tb = time.time()

    def inner(station, namespace):
        return {"time": {"unit": "s", "value":time.time() - tb}}
    return inner
```

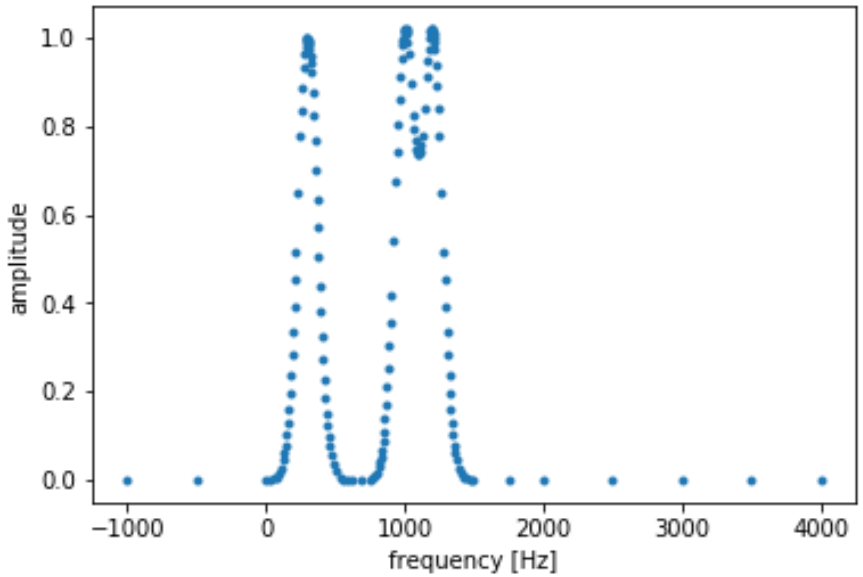
On the previous slide we have not taken into account that instrument buffers are finite. We cannot simply run a loop for an arbitrary length of time and aqcuire the buffer, or we may experience a buffer overrun. We need to periodically read and empty the buffer.

In this usecase, we finally see what “namespace” is used for. The “get_lockin_amplifier_buffer” keeps a track of the number of triggers which have been send through a variable in the namespace: “namespace.trigger_count”, which was initialized in the setup function.

The “get_lockin_amplifier_buffer” only reads the buffer when the trigger count reaches the “max_buffer_size”, or when the user specifies that we are at the end of a loop and a read needs to be forced.

Some practical example III: Adaptive stepping

```
measurement = Measurement(
    setup,
    cleanup,
    sweep(r.frequency, adaptive_sampler("amplitude", -1000, 4000, 6))(
        measure
    )
)
```

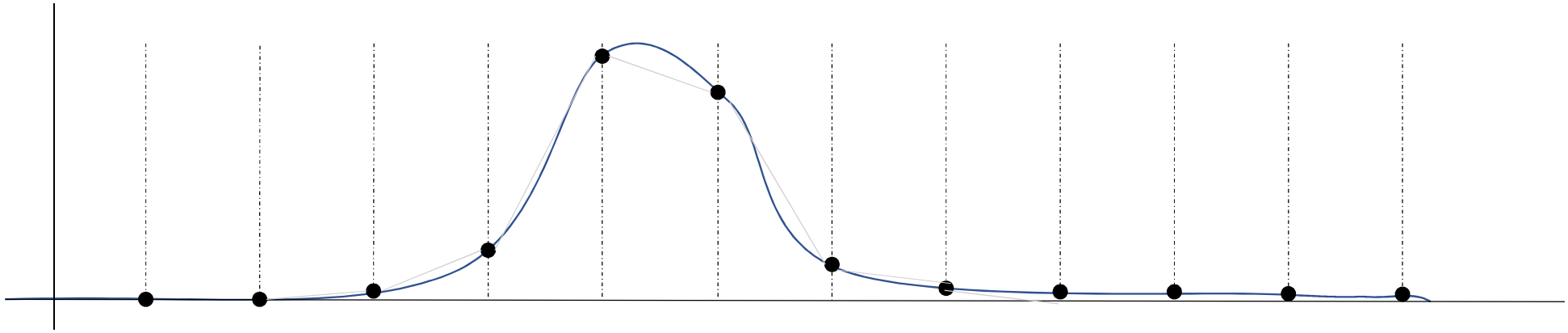


Let us suppose we have a resonator we would like to acquire a spectrum of. We do not know where the resonace frequencies are so that is we take a broad spectrum.

If we would apply a homogeneous sampling then we would have to make a trade off between the total measurement time (and the resulting data set size) and resolution, with higher resolution requiring more time and space.

However, we could also try to have a higher sampling rate only where required. This is shown in the left plot. But how do we achieve This? The next slides explains the algorithm.

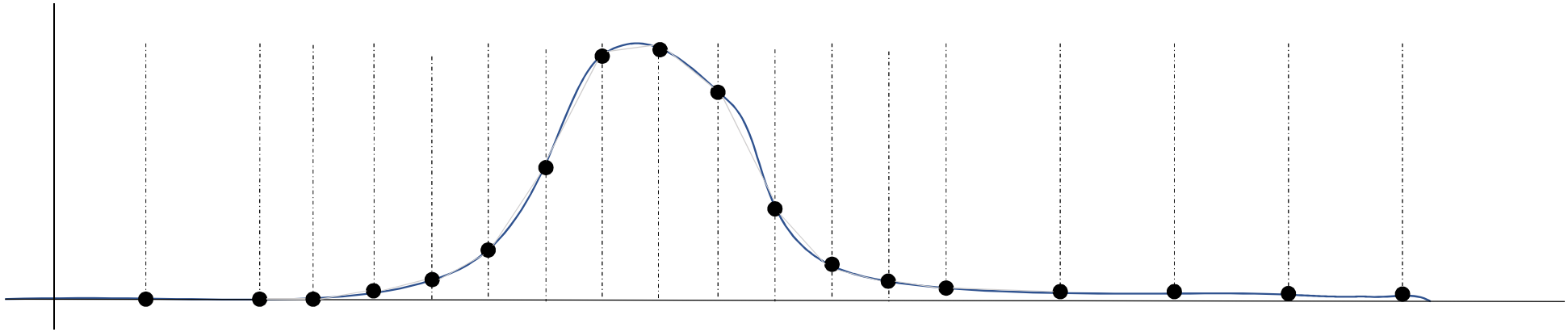
Adaptive sampling algorithm



Initial segmenting of the domain.

For each segment, test if the function is approximated sufficiently. If not break the segment in two and add the new segment to the list.

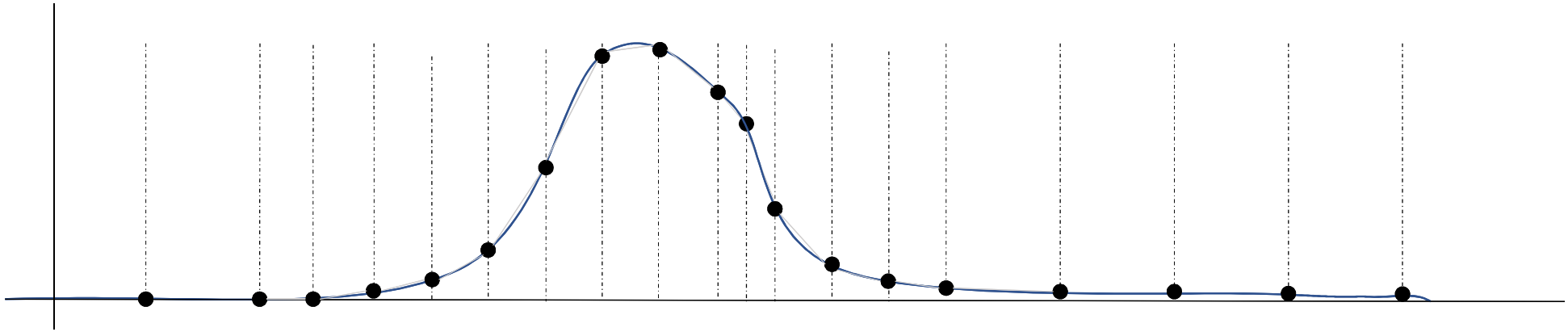
Adaptive sampling algorithm



Segments after the first iteration.

For each segment, test if the function is approximated sufficiently. If not break the segment in two and add the new segment to the list.

Adaptive sampling algorithm



Segments after the last iteration.

For each segment, test if the function is approximated sufficiently. If not break the segment in two and add the new segment to the list.

Appendix: Pysweep and databases

For i in $\text{Sweep}\left(p, \begin{bmatrix}0\\1\\2\end{bmatrix}\right) \times \left[[m()] + \left[\text{sweep}\left(q, \begin{bmatrix}1\\2\\4\end{bmatrix}\right) \times [n()]\right]\right]$:

Add_to_db(i)

Iteration 1: **i =** {“p”: {“value”: 0, “unit”: “V”, “independent”: True}, “other_measurement”: {“value”: 0.23, “unit”: “V”}}
Iteration 2: **i =** {“p”: {“value”: 0, “unit”: “V”, “independent”: True}, {“q”: {“value”: 1, “unit”: “V”, “independent”: True}, , “some_measurement”: {“value”: 4.5, “unit”: “V”}}
Iteration 3: **i =** {“p”: {“value”: 0, “unit”: “V”, “independent”: True}, {“q”: {“value”: 2, “unit”: “V”, “independent”: True}, , “some_measurement”: {“value”: 4.3, “unit”: “V”}}
Iteration 4: **i =** {“p”: {“value”: 0, “unit”: “V”, “independent”: True}, {“q”: {“value”: 4, “unit”: “V”, “independent”: True}, , “some_measurement”: {“value”: 5.1, “unit”: “V”}}
Iteration 5: **i =** {“p”: {“value”: 1, “unit”: “V”, “independent”: True}, “other_measurement”: {“value”: m2, “unit”: “V”}}
Iteration 6: .....

Appendix: Pysweep and databases

Iteration 1:
{“p”: {“value”: 0, “unit”: “V”, “independent”: True}, “other_measurement”: {“value”: 0.23, “unit”: “V”}}

Parameters: {
“p”: {“hash”: 1, “unit”: “V”, “independent”: True},
“other_measurement”: {“hash”: 2, “unit”: “V”}
}

Values: [
{1: 0, 2: 0.23},
]

Appendix: Pysweep and databases

Iteration 2:
{“p”: {“value”: 0, “unit”: “V”, “independent”: True}, {“q”: {“value”: 1, “unit”: “V”, “independent”: True}, , “some_measurement”: {“value”: 4.5, “unit”: “V”}}

Parameters: {
“p”: {“hash”: 1, “unit”: “V”, “independent”: True},
“other_measurement”: {“hash”: 2, “unit”: “V”},
“q” ”: {“hash”: 3, “unit”: “V”, “independent”: True},
“some_measurement”: {“hash”: 4, “unit”: “V”}
}

Values: [
{1: 0, 2: 0.23},
{1: 0, 3: 1, 4: 4.5}
]

Appendix: Pysweep and databases

Iteration 3:
{“p”: {“value”: 0, “unit”: “V”, “independent”: True}, {“q”: {“value”: 2, “unit”: “V”, “independent”: True}, , “some_measurement”: {“value”: 4.3, “unit”: “V”}}

Parameters: {
“p”: {“hash”: 1, “unit”: “V”, “independent”: True},
“other_measurement”: {“hash”: 2, “unit”: “V”},
“q” ”: {“hash”: 3, “unit”: “V”, “independent”: True},
“some_measurement”: {“hash”: 4, “unit”: “V”}
}

Values: [
{1: 0, 2: 0.23},
{1: 0, 3: 1, 4: 4.5},
{1: 0, 3: 2, 4: 4.3}
]

Appendix: Pysweep and databases

Iteration 4:
{“p”: {“value”: 0, “unit”: “V”, “independent”: True}, {“q”: {“value”: 4, “unit”: “V”, “independent”: True}, , “some_measurement”: {“value”: 5.1, “unit”: “V”}}

Parameters: {
“p”: {“hash”: 1, “unit”: “V”, “independent”: True},
“other_measurement”: {“hash”: 2, “unit”: “V”},
“q” ”: {“hash”: 3, “unit”: “V”, “independent”: True},
“some_measurement”: {“hash”: 4, “unit”: “V”}
}

Values: [
{1: 0, 2: 0.23},
{1: 0, 3: 1, 4: 4.5},
{1: 0, 3: 2, 4: 4.3},
{1: 0, 3: 4, 4: 5.1}
]

Appendix: Pysweep and databases

Iteration 5:
{“p”: {“value”: 1, “unit”: “V”, “independent”: True}, “other_measurement”: {“value”: 0.21, “unit”: “V”}}

Parameters: {
“p”: {“hash”: 1, “unit”: “V”, “independent”: True},
“other_measurement”: {“hash”: 2, “unit”: “V”},
“q” ”: {“hash”: 3, “unit”: “V”, “independent”: True},
“some_measurement”: {“hash”: 4, “unit”: “V”}
}

Values: [
{1: 0, 2: 0.23},
{1: 0, 3: 1, 4: 4.5},
{1: 0, 3: 2, 4: 4.3},
{1: 0, 3: 4, 4: 5.1},
{1: 1, 2: 0.21}
]

Appendix: Pysweep and databases

Iteration 5:
{“p”: {“value”: 1, “unit”: “V”, “independent”: True}, “other_measurement”: {“value”: 0.21, “unit”: “V”}}

Parameters: {
“p”: {“hash”: 1, “unit”: “V”, “independent”: True},
“other_measurement”: {“hash”: 2, “unit”: “V”},
“q” ”: {“hash”: 3, “unit”: “V”, “independent”: True},
“some_measurement”: {“hash”: 4, “unit”: “V”}
}

Save the parameters dictionary as a JSON file

Values: [
{1: 0, 2: 0.23},
{1: 0, 3: 1, 4: 4.5},
{1: 0, 3: 2, 4: 4.3},
{1: 0, 3: 4, 4: 5.1},
{1: 1, 2: 0.21}
]

Let each entry in the data set be a key value pairs where the keys are 8-bit integers

Appendix: Pysweep and databases

How do we retrive a measurement from the database?Lets suppose a command like:

dataset.get(“some_measurement”)

“some_measurement” has hash value 4, which means that we need to retrieve all dictionaries with this has value in the data set:

[{1: 0, 3: 1, 4: 4.5}, {1: 0, 3: 2, 4: 4.3}, {1: 0, 3: 4, 4: 5.1}]

Merge the values in these dictionaries:

{1: [0, 0, 0], 3: [1, 2, 4], 4: [4.5, 4.3, 5.1]}

From the parameters dictionary in the data set, we see that this translates to:

{“p”: [0, 0, 0], “q”: [1, 2, 4], “some_measurement”: [4.5, 4.3, 5.1]}