Concert Documentation

Release 0.32.0dev

Matthias Vogelgesang, Tomas Farago, Marcus Zuber

Mar 28, 2023

CONTENTS

Welcome to the Concert experiment control system documentation. This is the first place to answer all your questions related to using Concert for an experiment and developing more modules.

You can take a *quick guided tutorial* to see how the system is effectively used or take a closer in-depth look for special topics in our *user manual*.

CHAPTER

ONE

USER DOCUMENTATION

1.1 Installation

1.1.1 openSUSE packages

We use the openSUSE Build Service to provide packages for openSUSE 12.2 until openSUSE 13.2. Add the repository first, e.g.:

```
$ sudo zypper ar http://download.opensuse.org/repositories/home:/ufo-kit/openSUSE_12.2/␣
↪concert-repo
```

and update and install the packages. Note, that you have to install IPython on your own, if you intend to use the `concert` binary for execution:

```
$ sudo zypper update
$ sudo zypper in python-concert
```

1.1.2 Install from PyPI

It is recommended to use pip for installing Concert. The fastest way to install it is from PyPI:

```
$ sudo pip install concert
```

This will install the latest stable version. If you prefer an earlier stable version, you can fetch a tarball and install with:

```
$ sudo pip install concert-x.y.z.tar.gz
```

If you haven't have pip available, you can extract the tarball and install using the supplied `setup.py` script:

```
$ tar xfz concert-x.y.z.tar.gz
$ cd concert-x.y.z
$ sudo python setup.py install
```

More information on installing Concert using the `setup.py` script, can be found in the official Python documentation.

To install the Concert from the current source, follow the instructions given in the *developer documentation*.

Installing into a virtualenv

It is sometimes a good idea to install third-party Python modules independent of the system installation. This can be achieved easily using pip and virtualenv. When virtualenv is installed, create a new empty environment and activate that with

```
$ virtualenv my_new_environment
$ . my_new_environment/bin/activate
```

Now, you can install Concert's requirements and Concert itself

```
$ pip install -e path_to_concert/
```

As long as `my_new_environment` is active, you can use Concert.

1.2 Tutorial

Concert is primarily a user interface to control devices commonly found at a Synchrotron beamline. This guide will briefly show you how to use and extend it.

1.2.1 Running a session

In case you don't have a beamline at hand, you can import our sample sessions with the *import* command:

```
$ concert import --repo https://github.com/ufo-kit/concert-examples
```

Now *start* the tutorial session:

```
$ concert start tutorial
```

You will be greeted by an IPython shell loaded with pre-defined devices, processes and utilities like the pint package for unit calculation. Although, this package is primarily used for talking to devices, you can also use it to do simple calculations:

```
tutorial > a = 9.81 * q.m / q.s**2
tutorial > "Velocity after 5 seconds: {0}".format(5 * q.s * a)

'Velocity after 5 seconds: 49.05 meter / second'
```

You can get an overview of all defined devices by calling the *ddoc()* function:

```
tutorial > ddoc()

--------------------------------------------------------------------------------
  Name         Description   Parameters
--------------------------------------------------------------------------------
  motor        None           Name      Access  Unit  Description
                              position  rw      m     Position of the motor
--------------------------------------------------------------------------------
  ...
```

Now, by typing just the name of a device, you can see it's currently set parameter values:

```
tutorial > motor

<concert.devices.motors.dummy.LinearMotor object at 0x9419f0c>
 Parameter  Value
 position   12.729455653 millimeter
```

To get an overview of all devices' parameter values, use the *dstate()* function:

```
tutorial > dstate()

------------------------------------------------
  Name         Parameters
------------------------------------------------
  motor        position  99.382 millimeter
------------------------------------------------
  ...
```

To change the value of a parameter, you simply assign a new value to it:

```
tutorial > motor.position = 2 * q.mm
```

Now, check the position to verify that the motor reached the target position:

```
tutorial > motor.position
<Quantity(2.0, 'millimeter')>
```

Depending on the device, changing a parameter will block as long as the device has not yet reached the final target state. You can read more about asynchronous execution in the *Device control* chapter.

Note: A parameter value is always checked for the correct unit and soft limit condition. If you get an error, check twice that you are using a compatible unit (setting two seconds on a motor position is obviously not) and are within the allowed parameter range.

---

pdoc() displays information about currently defined functions and processes and may look like this:

```
tutorial > pdoc()
------------------------------------------------------------------------------
Name                    Description
------------------------------------------------------------------------------
save_exposure_scan      Run an exposure scan and save the result as a NeXus
                        compliant file. This requires that libnexus and NexPy
                        are installed.
------------------------------------------------------------------------------
```

In case you are interested in the implementation of a function, you can use *code_of()*. For example:

```
tutorial > code_of(code_of)
def code_of(func):
    """Show implementation of *func*."""
    source = inspect.getsource(func)

    try:
        ...
```

Note: Because we are actually running an IPython shell, you can _always_ tab-complete objects and attributes. For example, to change the motor position to 1 millimeter, you could simply type `mot<Tab>.p<Tab> = 1 * q.mm`.

How to execute more things concurrently and how to stop execution can be found in *Concurrent execution*.

1.2.2 Creating a session

First of all, *initialize* a new session:

```
$ concert init new_session
```

and *start* the default editor with

```
$ concert edit new_session
```

At the top of the file, you can see a string enclosed in three ". This should be changed to something descriptive as it will be shown each time you start the session.

Sessions are just normal Python modules with one additional feature that you may use top-level `await` in them, i.e. outside of an `async def` function, for more info see *Importing*.

Adding devices

To create a device suited for your experiment you have to import it first. Concert uses the following packaging scheme to separate device classes and device implementations: `concert.devices.[class].[implementation]`. Thus if you want to create a dummy ring from the storage ring class, you would add this line to your session:

```
from concert.devices.storagerings.dummy import StorageRing
```

Once imported, you can create the device and give it a name that will be accessible from the command line shell:

```
from concert.devices.motors.dummy import LinearMotor

ring = await StorageRing()
motor = await LinearMotor()
```

Importing other sessions

To specify experiments that share a common set of devices, you can define a base session and import it from each sub-session:

```
from base import *
```

Now everything that was defined will be present when you start up the new session.

1.2.3 Hello World

Let's create a session:

```
concert edit scan
```

And then add some code inside so that we can discuss some of the core Concert features. You can download the scan example or just copy this:

```
"""# *scan* shows scanning of camera's exposure time.

## Usage
    await run(producer, line, acc)

## Notes
"""

import asyncio
import logging
from inspect import iscoroutinefunction
import concert
concert.require("0.30.0")

from concert.coroutines.base import broadcast
from concert.coroutines.sinks import Accumulate
from concert.quantities import q
from concert.session.utils import cdoc, ddoc, dstate, pdoc, code_of
from concert.devices.cameras.dummy import Camera
from concert.ext.viewers import PyplotViewer, PyQtGraphViewer
from concert.processes.common import ascan

LOG = logging.getLogger(__name__)
# Disable progress bar in order not to interfere with printing
concert.config.PROGRESS_BAR = False


async def feedback():
    """Our feedback just returns image mean."""
    # Let's pretend this is a serious operation which takes a while
    await asyncio.sleep(1)
    image = await camera.grab()
    # Also show the current image
    await viewer.show(image)

    return image.mean()


async def run(producer, line, accumulator):
    coros = broadcast(producer, line, accumulator)
    await asyncio.gather(*coros)

    return accumulator.items
```

(continues on next page)

(continued from previous page)

```
viewer = await PyQtGraphViewer()
# The last image will be quite bright
viewer.limits = 0, 10000
# Plot image mean
line = await PyplotViewer(style='-o')
# Dummy camera
camera = await Camera()
# For scan results collection
acc = Accumulate()
# Let's create a scan so that it can be directly plugged into *run*
producer = ascan(camera['exposure_time'], 1 * q.ms, 100 * q.ms, 10 * q.ms,␣
↪feedback=feedback)
```

With this code you can execute the scan showing both the image and the mean and storing the result in `acc` by:

```
items = await run(producer, line, acc)
print(items) # or print(acc.items)
# Gives
[(1 <Unit('millisecond')>, 101.01860026041666),
 (11 <Unit('millisecond')>, 1101.0648697916668),
 (21 <Unit('millisecond')>, 2101.0111751302084),
 (31 <Unit('millisecond')>, 3100.9252408854168),
 (41 <Unit('millisecond')>, 4101.011533203125),
 (51 <Unit('millisecond')>, 5101.0090625),
 (61 <Unit('millisecond')>, 6100.966005859375),
 (71 <Unit('millisecond')>, 7101.112858072916),
 (81 <Unit('millisecond')>, 8100.928743489583),
 (91 <Unit('millisecond')>, 9101.179690755209)]
```

or you can simply run the scan showing both the image and the mean to see the mean:

```
await line(producer)
```

or you can iterate through the values and decide what to do with them yourself:

```
async for x, y in producer:
    print(f'x={x}, y={y}')
# Gives
x=1 millisecond, y=101.00574544270833
x=11 millisecond, y=1100.9828515625
x=21 millisecond, y=2100.9941015625
x=31 millisecond, y=3100.982431640625
x=41 millisecond, y=4100.772060546875
x=51 millisecond, y=5100.855152994792
x=61 millisecond, y=6100.988649088542
x=71 millisecond, y=7101.148798828125
x=81 millisecond, y=8101.085227864583
x=91 millisecond, y=9100.949088541667
```

1.3 User manual

1.3.1 Command line shell

Concert comes with a command line interface that is launched by typing `concert` into a shell. Several subcommands define the action of the tool.

Session commands

The `concert` tool is run from the command line. Without any arguments, its help is shown:

```
$ concert
usage: concert [-h] [--version]  ...

optional arguments:
  -h, --help  show this help message and exit
  --version   show program's version number and exit

Concert commands:

    init      Create a new session
    edit      Edit a session
    log       Show session logs
    show      Show available sessions or details of a given *session*
    mv        Move session *source* to *target*
    cp        Copy session *source* to *target*
    rm        Remove one or more sessions
    import    Import an existing *session*
    export    Export all sessions as a Zip archive
    start     Start a session
    docs      Create documentation of *session* docstring
    spyder    Start session using Spyder
```

The tool is command-driven, that means you call it with a command as its first argument. To read command-specific help, use:

```
$ concert [command] -h
```

Note: When Concert is installed system-wide, a bash completion for the `concert` tool is installed too. This means, that commands and options will be completed when pressing the `Tab` key.

init

Create a new session with the given name:

```
concert init experiment
```

If such a session already exists, Concert will warn you.

--force

Create the session even if one already exists with this name.

--imports

List of module names that are added to the import list.

Note: The location of the session files depends on the chosen installation method. If you installed into a virtual environment `venv`, the files will be stored in `/path/to/venv/share/concert`. If you have installed Concert system-wide our without using a virtual environment, it is installed into `$XDG_DATA_HOME/concert` or `$HOME/.local/share/concert` if the former is not set. See the XDG Base Directory Specification for further information. It is probably a *very* good idea to put the session directory under version control.

---

edit

Edit the session file by launching `$EDITOR` with the associated Python module file:

```
concert edit session-name
```

This file can contain any kind of Python code, but you will most likely just add device definitions and import processes that you want to use in a session. If the `session-name` doesn't exist it is created.

log

Show log of session:

```
concert log session-name
```

If a session is not given, the log command shows entries from all sessions.

--follow

Instead of showing the past log, update as changes come in. This is the same operation as if the log file was viewed with `tail -f`.

By default, logs are gathered in `$XDG_DATA_HOME/concert/concert.log`. To change this, you can pass the `--logto` and `--logfile` options to the `start` command. For example, if you want to output log to `stderr` use

```
concert start experiment --logto=stderr
```

or if you want to get rid of any log data use

```
concert start experiment --logto=file --logfile=/dev/null
```

show

Show all available sessions or details of a given session:

```
concert show [session-name]
```

mv

Rename a session:

```
concert mv old-session new-session
```

cp

Copy a session:

```
concert cp session session-copy
```

rm

Remove one or more sessions:

```
concert rm session-1 session-2
```

> **Warning:** Be careful. The session file is unlinked from the file system and no backup is made.

import

Import an existing session from a Python file:

```
concert import some-session.py
```

Concert will warn you if you try to import a session with a name that already exists.

`--force`
: Overwrite session if it already exists.

`--repo`
: The URL denotes a Git repository from which the sessions are imported.

> **Warning:** The server certificates are *not* verified when specifying an HTTPS connection!

export

Export all sessions as a Zip archive:

```
concert export foobar
```

Creates a Zip archive named *foobar.zip* containing all sessions.

start

Load the session file and launch an IPython shell:

```
concert start session-name
```

The quantities package is already loaded and named `q`.

`--logto``={stderr, file}`

Specify a method for logging events. If this flag is not specified, `file` is used and assumed to be `$XDG_DATA_HOME/concert/concert.log`.

`--logfile``=<filename>`

Specify a log file if `--logto` is set to `file`.

`--loglevel``={debug, info, warning, error, critical}`

Specify lowest log level that is logged.

`--non-interactive`

Run the session as a script and do not launch a shell.

`--filename``=<filename>`

Start a session from a file without initializing.

Note: You may use the `await` keyword in session files and the sesion will be loaded correctly, for details see *Importing*.

docs

Create a PDF documentation for a session:

```
concert docs session-name
```

Creates a PDF manual named *session-name.zip* with the contents taken from the session's docstring. The docstring should be formatted in Markdown markup.

Note: This requires an installation of Pandoc and PDFLaTeX.

Importing

When you import a module or a session, before anything else, concert first looks into the sessions directory. If the module is not found, it looks into the current working directory and if it is not found even there it searches in `sys.path`, where all the standard paths are stored.

Concert can run sessions with top-level `await` (outside `async def` functions). Sessions can also import such modules into them and even nest such imports. There are two limitations to this:

- if you have a top-level `await` in a module, you cannot use the asyncio's loop, e.g. by concert's `run_in_loop` function
- you cannot import modules with top-level `await` inside functions, you need to put the imports to the top level

For example, this is possible (session `motors`):

```
from concert.devices.motors.dummy import LinearMotor

motor = await LinearMotor()
```

and this is possible:

```
from motors import motor
from concert.quantities import q

await motor.set_position(1 * q.mm)
```

On the other hand, this is *not* possible:

```
async def foo():
    import motors  # The example session above

await foo()
```

and this is *not* possible:

```
from concert.coroutines.base import run_in_loop
await asyncio.sleep(1)
run_in_loop(asyncio.sleep(1))
```

Remote access

Concert comes with two shell scripts that leverage the terminal multiplexer tmux and the secure shell protocol. Thus you *must* have installed and started an OpenSSH server as well as the relevant ports opened.

To start a Concert session server run:

```
concert-server <session-name>
```

This starts a new tmux session which you can *detach* from by typing Ctrl-B. On a client machine you can connect to the server and tmux session by running:

```
concert-connect <host address>
```

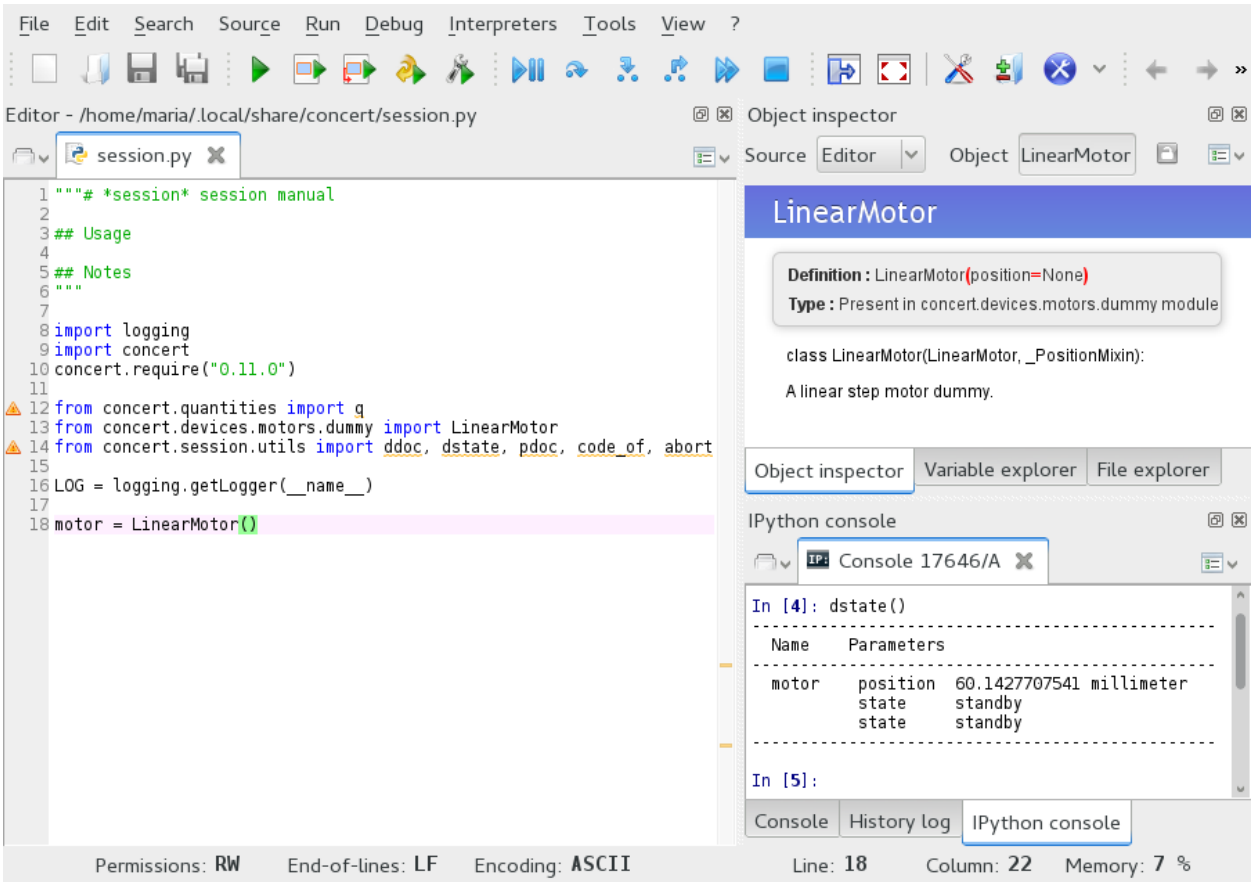
Extensions

Spyder

If Spyder is installed, start the session within the Spyder GUI:

```
concert spyder <session-name>
```

In Spyder you can for example edit the session, check the documentation or run an IPython console or a Python interpreter:



1.3.2 Concurrent execution

Concert relies on concurrency instead of parallelism because what mostly happens is communication with devices, which is I/O bound. Concurrency is realized via *coroutines* and Python's asyncio module. A *coroutine* is a function defined as `async def` and inside it can yield execution for other coroutines via the `await` keyword. When you call a *coroutine function*, it returns a *coroutine* object, but the code of that function is not yet executed. One way of invoking execution in a *blocking* way is by the `await` keyword followed by a coroutine object. This will block the session until the coroutine is finished. Alternatively, you can start the execution in a *non-blocking* way by calling *`start()`* and get the control back immediately. *`start()`* returns a task object, which can also be `awaited`. Most of the `async def` functions in Concert are wrapped into tasks by the *`background()`* decorator, so you do not need to use the *`start()`* in order to start execution immediately. You should however keep this in mind when writing your own coroutines and decorate them (see below) if you want them to be automatically started upon invocation. Overall, in Concert there are two ways to execute coroutines:

1. as *non-blocking tasks*,
2. as *blocking tasks* in combination with the `await` syntax

An example:

```
import asyncio
from concert.coroutines.base import background, start

async def corofunc():
    await asyncio.sleep(0.1)
    return 1

@background
async def corofunc_run_immediately():
    await asyncio.sleep(0.1)
    return 1

coro = corofunc() # coro is a coroutine, not yet a task and has not started
task = start(coro) # wraps the coroutine into a task and starts it, does not block
result = await task # this blocks, result contains 1
await corofunc() # this blocks too

task = corofunc_run_immediately() # runs immediately, does not block
result = await task # this blocks, result contains 1
await corofunc_run_immediately() # this blocks too
```

A more reallistic example:

```
from concert.devices.motors.dummy import LinearMotor

motor = await LinearMotor()
task = motor.home() # this doesn't block
await task # this blocks
await motor.home() # this blocks too
```

You can cancel running tasks which are being `awaited` by pressing *ctrl-c*. This for instance stops a motor. By *ctlr-k*, you can also cancel *all* running background tasks which were started by the *start()* function or *background()* decorator. On the top of cancellation, *ctrl-k* will call *Device.emergency_stop()* on all devices in order to bring them to a standstill. Please note that if there is non-async function running, *ctrl-k* will only get triggered after it has finished. You can first cancel the running operation byt *ctrl-c* followed by *ctrl-k* to execute it as soon as possible.

Concurrency

Concurrent execution itself is realized via asyncio's tools, like gather, which executes given coroutines concurrently and returns their results:

```
async def corofunc():
    await asyncio.sleep(0.1)
    return 1

await asyncio.gather(corofunc(), corofunc())
```

Synchronization

When using the concurrent getters and setters of *Device* and *Parameter*, coroutines can not be sure if other coroutines manipulate the device. To lock devices or specific parameters, coroutines can use devices with context managers:

```
async with shutter, motor['position']:
    await motor.set_position(2 * q.mm)
    await shutter.open()
```

Inside the `async with` environment, a coroutine has exclusive access to the devices and parameters.

1.3.3 Device control

Parameters

In Concert, a *device* is a software abstraction for a piece of hardware that can be controlled. Each device consists of a set of named `Parameter` instances and device-specific methods. Devices rely heavily on concurrent execution, so that multiple devices can do multiple actions at the same time. In order for the devices to be able to use concurrency already in their constructors, they must be instantiated with the *await* keyword like this:

```
from concert.devices.motors.dummy import LinearMotor
motor = await LinearMotor()
```

The devices contain several parameters and if you know the parameter name, you can get a reference to the parameter object by using the index operator:

```
pos_parameter = motor['position']
```

To set and get parameters explicitly , you can use the `Parameter.get()` and `Parameter.set()` coroutine methods:

```
await pos_parameter.set(1 * q.mm)
print (await pos_parameter.get())
```

Both methods return a coroutine (see *Concurrent execution* for details) and give the control back to you, so that other things can (and should) happen concurrently. As you can see, to get the result of a coroutine you use the `await` keyword.

An easier way to set and get parameter values are properties via the dot-name-notation:

```
motor.position = 1 * q.mm
print (motor.position)
```

As you can see, accessing parameters this way will *always be synchronous* and *block* execution until the value is set or fetched. If you press *ctrl-c* while you are setting a parameter the function will stop and a cancelling action will be called, like stopping a motor, so that you don't accidentaly crush your devices. However, please be aware that this is up to device implementation, so you should check if the device you are using is safe in this manner.

Parameter objects are not only used to communicate with a device but also carry meta data information about the parameter. The most important ones are `Parameter.name`, `Quantity.unit` (in case of a parameter having a physical unit, like motor's position) and the doc string describing the parameter. Moreover, parameters can be queried for access rights using `Parameter.writable`.

To get all parameters of an object, you can iterate over the device itself

```
for param in motor:
    print("{0} => {1}".format(param.unit if hasattr(param,'unit') else None, param.name))
```

Saving state

In some scenarios you would like to come back to a certain state. Let's suppose, you have a motor that you want to check if it moves. If it does, you want it to go back to the same place it came from. For these cases you can use `Device.stash()` to store the current state of a device and `Device.restore()` to go back. Because this is done in a stacked fashion, you can, for example, model local coordinate pretty easily:

```
await motor.stash()

# Do movements aka modify the "local" coordinate system
await motor.move(1 * q.mm)

# Go back to the original state
await motor.restore()
```

Locking parameters

In case you want to prevent a parameter from being written you can use *ParameterValue.lock()*. If you specify a *permanent* parameter to be True the parameter cannot be unlocked anymore. In case you want to unlock a parameter you can use *ParameterValue.unlock()*, to get the state you can check the attribute *ParameterValue.locked*. All the parameters within a device can be locked and unlocked at once, for example one can do:

```
motor['position'].lock()
motor.position = 10 * q.mm
# Does not work, you will get a LockError
motor['position'].locked
True

motor['position'].unlock()

# Works as expected
motor.position = 10 * q.mm

# Lock the whole device (all parameters)
motor.lock(permanent=True)

# This will not work anymore
motor.unlock()
# You will get a LockError
```

Limits

Limits allow you to restrict setting *Quantity* to a certain range. You can specify the `Quantity.lower` and `Quantity.upper` limits. Usage:

```
# Blocking version
motor['position'].lower = -10 * q.mm
print(motor['position'].lower)
# Coroutine version
await motor['position'].set_lower(-15 * q.mm)
```

(continues on next page)

(continued from previous page)

```
print(await motor['position'].get_lower())

# Blocking version
motor['position'].upper = 10 * q.mm
print(motor['position'].upper)
# Coroutine version
await motor['position'].set_upper(15 * q.mm)
print(await motor['position'].get_upper())

# Locking
motor['position'].lock_limits()
# Will not work, you will get a LockError
motor['position'].lower = -10 * q.mm

motor['position'].unlock_limits()
# This will work again
motor['position'].lower = -10 * q.mm

motor['position'].lock_limits(permanent=True)
# This will not work anymore until you restart the session
motor['position'].unlock_limits()
```

Emergency stop

On *ctrl-k*, the background tasks are cancelled and on top of that on all devices *Device.emergency_stop()* will be called in order to bring them to a standstill.

1.3.4 Data processing

Coroutines

Coroutines provide a way to process data and yield execution until more data is produced. *Generators* represent the source of data and can be used as normal iterators, e.g. in a `for` loop. Coroutines can use the output of a generator to either process data and output a new result item in a *filter* fashion or process the data without further results in a *sink* fashion. For more on coroutines, see *Concurrent execution*.

Data processing with coroutines uses generators and iterates over their items like this:

```
async def producer(num):
    for i in range(num):
        yield i

async def printer(producer):
    async for item in producer:
        print(item)

# Usage:
await printer(producer(10))
```

printer coroutine fetches data items and prints them one by one. Because no data is produced, this coroutine falls into the sink category. Concert provides some common pre-defined sinks in the *sinks* module.

Filters hook into the data stream and process the input to produce some output. For example, to generate a stream of squared input, you would write:

```
def square(consumer):
    async for item in producer:
        yield item ** 2

# Usage:
await printer(square(producer(10)))
```

You can find a variety of pre-defined filters in the *filters* module.

Broadcasting

To fan out a single input stream to multiple consumers, you can use the *broadcast()*. Its first argument is the producer and the rest are consumers. *broadcast()* creates the connections from producer to consumers and returns a list of coroutines, which can be used by `asyncio.gather()` function, like this:

```
from concert.coroutines.base import broadcast

coros = broadcast(producer(10), printer, printer)
await asyncio.gather(*coros)
```

High-performance processing

The generators and coroutines yield execution, but if the data production should not be stalled by data consumption the coroutine should only provide data buffering and delegate the real consumption to a separate thread or process. The same can be achieved by first buffering the data and then yielding them by a generator. It comes from the fact that a generator will not produce a new value until the old one has been consumed.

High-performance computing

The *ufo* module provides classes to process data from an experiment with the UFO data processing framework. The simplest example could look like this:

```
from concert.ext.ufo import InjectProcess
from gi.repository import Ufo
import numpy as np
import scipy.misc

pm = Ufo.PluginManager()
writer = pm.get_task('write')
writer.props.filename = 'foo-%05i.tif'

proc = InjectProcess(writer)

proc.start()
await proc.insert(scipy.misc.ascent())
proc.wait()
```

To save yourself some time, the *ufo* module provides a wrapper around the raw `UfoPluginManager`:

```
from concert.ext.ufo import PluginManager

pm = PluginManager()
writer = pm.get_task('write', filename='foo-%05i.tif')
```

Viewing processed data

Concert has a Matplotlib integration to simplify viewing 1D time series with the *PyplotViewer*. For 2D, there are multiple implementations, for details see *Viewers* and Concert examples.

Writing image data

Concert provides *DirectoryWalker* for traversing the filesystem and writing image sequences. You can use its *descend()* method to descend into a sub-directory and the *ascend()* method to return one level back.

If you just want to write images in the current directory use the *write()* method. To create an image writer in either the current directory or one level below, you can use the *create_writer()* method. This method creates the writer and if you specify a sub-directory also ascends back. You should use this in a *with* statement to make sure that while you are creating the image writer, some other coroutine does not change walker's path. The writing itself can then happen after the *with* statement:

```
async with walker:
    writer = walker.create_writer(producer, name='subdirectory')

# create_writer ascends back so the writing itself can happen outside of the
# with statement
await writer
```

1.3.5 Experiments

Experiments connect data acquisition and processing. They can be run multiple times by the `base.Experiment.run()`, they take care of proper file structure and logging output.

Acquisition

Experiments consist of *Acquisition* objects which encapsulate data generator and consumers for a particular experiment part (dark fields, radiographs, ...). This way the experiments can be broken up into smaller logical pieces. A single acquisition object needs to be reproducible in order to repeat an experiment more times, thus we specify its generator and consumers as callables which return the actual generator or consumer. We need to do this because generators cannot be "restarted".

It is very important that you enclose the executive part of the production and consumption code in *try-finally* to ensure proper clean up. E.g. if a producer starts rotating a motor, then in the *finally* clause there should be the call *await motor.stop()*.

An example of an acquisition could look like this:

```
from concert.experiments.base import Acquisition

# This is a real generator, num_items is provided somewhere in our session
```

(continues on next page)

(continued from previous page)

```
async def produce():
    try:
        for i in range(num_items):
            yield i
    finally:
        # Clean up here
        pass

# A simple coroutine sink which prints items
async def consume(producer):
    try:
        async for item in producer:
            print(item)
    finally:
        # Clean up here
        pass

acquisition = await Acquisition('foo', produce, consumers=[consume])
# Now we can run the acquisition
await acquisition()
```

class concert.experiments.base.**Acquisition**(*self*, *name*, *producer*, *consumers=None*, *acquire=None*)

An acquisition acquires data, gets it and sends it to consumers.

producer

a callable with no arguments which returns a generator yielding data items once called.

consumers

a list of callables with no arguments which return a coroutine consuming the data once started, can be empty.

acquire

a coroutine function which acquires the data, takes no arguments, can be None.

Base

Base *base.Experiment* makes sure all acquisitions are executed. It also holds *addons.Addon* instances which provide some extra functionality, e.g. live preview, online reconstruction, etc. To make a simple experiment for running the acquisition above and storing log with *concert.storage.Walker*:

```
import logging
from concert.experiments.base import Acquisition, Experiment
from concert.storage import DirectoryWalker

LOG = logging.getLogger(__name__)

walker = DirectoryWalker(log=LOG)
acquisitions = [await Acquisition('foo', produce)]
experiment = await Experiment(acquisitions, walker)

await experiment.run()
```

class concert.experiments.base.**Experiment**(*self*, *acquisitions*, *walker=None*, *separate_scans=True*, *name_fmt='scan_{:>04}'*)

Experiment base class. An experiment can be run multiple times with the output data and log stored on disk. You can prepare every run by *prepare()* and finish the run by *finish()*. These methods do nothing by default. They can be useful e.g. if you need to reinitialize some experiment parts or want to attach some logging output.

acquisitions

A list of acquisitions this experiment is composed of

walker

A *concert.storage.Walker* descends to a data set specific for every run if given

separate_scans

If True, *walker* does not descend to data sets based on specific runs

name_fmt

Since experiment can be run multiple times each iteration will have a separate entry on the disk. The entry consists of a name and a number of the current iteration, so the parameter is a formattable string.

ready_to_prepare_next_sample

asyncio.Event that can be used to tell a processes.experiment.Director that the next iteration can be prepared. Can be set() to allow the preparation while the experiment is still running.

await acquire()

Acquire data by running the acquisitions. This is the method which implements the data acquisition and should be overwritten if more functionality is required, unlike `run()`.

property acquisitions

Acquisitions is a read-only attribute which has to be manipulated by explicit methods provided by this class.

add(*acquisition*)

Add *acquisition* to the acquisition list and make it accessible as an attribute:

```
frames = Acquisition(...)
experiment.add(frames)
# This is possible
experiment.frames
```

await finish()

Gets executed after every experiment run.

get_acquisition(*name*)

Get acquisition by its *name*. In case there are more like it, the first one is returned.

await get_running_acquisition()

Get the currently running acquisition.

await prepare()

Gets executed before every experiment run.

remove(*acquisition*)

Remove *acquisition* from experiment.

swap(*first*, *second*)

Swap acquisition *first* with *second*. If there are more occurrences of either of them then the ones which are found first in the acquisitions list are swapped.

Experiments also have a `base.Experiment.log` attribute, which gets a new handler on every experiment run and this handler stores the output in the current experiment working directory defined by it's *concert.storage.Walker*.

Advanced

Sometimes we need finer control over when exactly is the data acquired and worry about the download later. We can use the *acquire* argument to *Acquisition*. This means that the data acquisition can be invoked before data download. *Acquisition* calls its *acquire* first and only when it is finished connects producer with consumers.

The Experiment class has the attribute *base.Experiment.ready_to_prepare_next_sample* which is an instance of an `asyncio.Event`. This can be used to tell that most of the experiment is finished and a new iteration of this experiment can be prepared (e.g. by the *concert.directors.base.Director*. In the `base.Experiment.run()` the *base.Experiment.ready_to_prepare_next_sample* will be set that at the end of an experiment is is always set. In the beginning of the `base.Experiment.run()` it will be cleared. This is an example implementation making use of this:

```
from concert.experiments.base import Experiment, Acquisition
class MyExperiment(Experiment):
        async def __ainit__(self, walker, camera):
                acq = Acquisition("acquisition", self._produce_frames)
                self._camera = camera
                await super().__ainit__([acq], walker)

        async def _produce_frame(self):
                num_frames = 100
                async with self._camera.recording():
                        # Do the acquisition of the frames in camera memory

                # Only the readout and nothing else will happen after this point.
                self.ready_to_prepare_next_sample.set()

                async with self._camera.readout():
                        for i in range(num_frames):
                                yield await self._camera.grab()
```

Imaging

A basic frame acquisition generator which triggers the camera itself is provided by *frames()*

async for ... in concert.experiments.imaging.**frames**(*num_frames*, *camera*, *callback=None*)

A generator which takes *num_frames* using *camera*. *callback* is called after every taken frame.

There are tomography helper functions which make it easier to define the proper settings for conducting a tomographic experiment.

concert.experiments.imaging.**tomo_angular_step**(*frame_width*)

Get the angular step required for tomography so that every pixel of the frame rotates no more than one pixel per rotation step. *frame_width* is frame size in the direction perpendicular to the axis of rotation.

concert.experiments.imaging.**tomo_projections_number**(*frame_width*)

Get the minimum number of projections required by a tomographic scan in order to provide enough data points for every distance from the axis of rotation. The minimum angular step is considered to be needed smaller than one pixel in the direction perpendicular to the axis of rotation. The number of pixels in this direction is given by *frame_width*.

concert.experiments.imaging.**tomo_max_speed**(*frame_width*, *frame_rate*)

Get the maximum rotation speed which introduces motion blur less than one pixel. *frame_width* is the width of the frame in the direction perpendicular to the rotation and *frame_rate* defines the time required for recording one frame.

Note: frame rate is required instead of exposure time because the exposure time is usually shorter due to the camera chip readout time. We need to make sure that by the next exposure the sample hasn't moved more than one pixel from the previous frame, thus we need to take into account the whole frame taking procedure (exposure + readout).

Synchrotron and X-Ray tube experiments

In concert.experiments.synchrotron and concert.experiments.xraytube implementations of Radiography, SteppedTomography, ContinuousTomography and SteppedSpiralTomography, ContinuousSpiralTomography and GratingInterferometryStepping are implemented for the two different source types.

For detailed information how they are implemented, you can have a look at the base classes *concert.experiments.imaging.Radiography*, *concert.experiments.imaging.Tomography*, *concert.experiments.imaging.SteppedTomography*, *concert.experiments.imaging.ContinuousTomography*, *concert.experiments.imaging.SteppedSpiralTomography*, *concert.experiments.imaging.ContinuousSpiralTomography* and concert.experiments.imaging.GratingInterferometryStepping.

In the standard configuration, all tomography and radiography experiments first acquire the dark images, then the flat images and the projection images of the sample at the end. This order can be adjusted by the *swap()* command.

Radiography

class concert.experiments.synchrotron.**Radiography**(*self*, *walker*, *flat_motor*, *radio_position*, *flat_position*, *camera*, *shutter*, *num_flats=200*, *num_darks=200*, *num_projections=3000*, *separate_scans=True*)

Radiography experiment

await __ainit__(*walker*, *flat_motor*, *radio_position*, *flat_position*, *camera*, *shutter*, *num_flats=200*, *num_darks=200*, *num_projections=3000*, *separate_scans=True*)

Parameters

- **walker** (concert.storage.Walker) – Walker for storing experiment data.
- **flat_motor** – Motor for moving sample in and out of the beam. Must feature a 'position' property.
- **radio_position** – Position of *flat_motor* that the sample is positioned in the beam. Unit must be the same as flat_motor['position'].
- **flat_position** – Position of *flat_motor* that the sample is positioned out of the beam. Unit must be the same as flat_motor['position'].
- **camera** (concert.devices.cameras.base.Camera) – Camera to acquire the images.
- **shutter** (concert.devices.shutters.base.Shutter) – Stutter
- **num_flats** (*int*) – Number of images for flatfield correction.
- **num_darks** (*int*) – Number of images for dark correction.
- **num_projections** (*int*) – Number of projections.

class concert.experiments.xraytube.**Radiography**(*self*, *walker*, *flat_motor*, *radio_position*, *flat_position*, *camera*, *xray_tube*, *num_flats=200*, *num_darks=200*, *num_projections=3000*, *separate_scans=True*)

Radiography experiment

Parameters

xray_tube (concert.devices.xraytubes.base.XRayTube) – X-ray tube

await **__ainit__**(*walker*, *flat_motor*, *radio_position*, *flat_position*, *camera*, *xray_tube*, *num_flats=200*, *num_darks=200*, *num_projections=3000*, *separate_scans=True*)

Parameters

- **walker** (concert.storage.Walker) – Walker for storing experiment data.
- **flat_motor** – Motor for moving sample in and out of the beam. Must feature a 'position' property.
- **radio_position** – Position of *flat_motor* that the sample is positioned in the beam. Unit must be the same as flat_motor['position'].
- **flat_position** – Position of *flat_motor* that the sample is positioned out of the beam. Unit must be the same as flat_motor['position'].
- **camera** (concert.devices.cameras.base.Camera) – Camera to acquire the images.
- **xray_tube** (concert.devices.xraytubes.base.XRayTube) – X-ray tube
- **num_flats** (*int*) – Number of images for flatfield correction.
- **num_darks** (*int*) – Number of images for dark correction.
- **num_projections** (*int*) – Number of projections.

SteppedTomography

class concert.experiments.synchrotron.**SteppedTomography**(*self*, *walker*, *flat_motor*, *tomography_motor*, *radio_position*, *flat_position*, *camera*, *shutter*, *num_flats=200*, *num_darks=200*, *num_projections=3000*, *angular_range=<Quantity(180, 'degree')>*, *start_angle=<Quantity(0, 'degree')>*, *separate_scans=True*)

Stepped tomography

await **__ainit__**(*walker*, *flat_motor*, *tomography_motor*, *radio_position*, *flat_position*, *camera*, *shutter*, *num_flats=200*, *num_darks=200*, *num_projections=3000*, *angular_range=<Quantity(180, 'degree')>*, *start_angle=<Quantity(0, 'degree')>*, *separate_scans=True*)

Parameters

- **walker** (concert.storage.Walker) – Walker for storing experiment data.
- **flat_motor** – Motor for moving sample in and out of the beam. Must feature a 'position' property.
- **tomography_motor** (concert.devices.motors.base.RotationMotor) – Rotation-Motor for tomography scan.

- **radio_position** – Position of *flat_motor* that the sample is positioned in the beam. Unit must be the same as flat_motor['position'].
- **flat_position** – Position of *flat_motor* that the sample is positioned out of the beam. Unit must be the same as flat_motor['position'].
- **camera** (*concert.devices.camera.base.Camera*) – Camera to acquire the images.
- **shutter** (concert.devices.shutters.base.Shutter) – Stutter
- **num_flats** (*int*) – Number of images for flatfield correction.
- **num_darks** (*int*) – Number of images for dark correction.
- **num_projections** (*int*) – Number of projections.
- **angular_range** (*q.deg*) – Range for the scan of the *tomography_motor*.
- **start_angle** (*q.deg*) – Start position of *tomography_motor* for the first projection.

class concert.experiments.xraytube.**SteppedTomography**(*self*, *walker*, *flat_motor*, *tomography_motor*, *radio_position*, *flat_position*, *camera*, *xray_tube*, *num_flats=200*, *num_darks=200*, *num_projections=3000*, *angular_range=<Quantity(360, 'degree')>*, *start_angle=<Quantity(0, 'degree')>*, *separate_scans=True*)

Stepped tomography experiment.

Parameters

xray_tube (concert.devices.xraytubes.base.XRayTube) – X-ray tube

await **__ainit__**(*walker*, *flat_motor*, *tomography_motor*, *radio_position*, *flat_position*, *camera*, *xray_tube*, *num_flats=200*, *num_darks=200*, *num_projections=3000*, *angular_range=<Quantity(360, 'degree')>*, *start_angle=<Quantity(0, 'degree')>*, *separate_scans=True*)

Parameters

- **walker** (concert.storage.Walker) – Walker for storing experiment data.
- **flat_motor** – Motor for moving sample in and out of the beam. Must feature a 'position' property.
- **tomography_motor** (concert.devices.motors.base.RotationMotor) – Rotation-Motor for tomography scan.
- **radio_position** – Position of *flat_motor* that the sample is positioned in the beam. Unit must be the same as flat_motor['position'].
- **flat_position** – Position of *flat_motor* that the sample is positioned out of the beam. Unit must be the same as flat_motor['position'].
- **camera** (*concert.devices.camera.base.Camera*) – Camera to acquire the images.
- **xray_tube** (concert.devices.xraytubes.base.XRayTube) – X-ray tube
- **num_flats** (*int*) – Number of images for flatfield correction.
- **num_darks** (*int*) – Number of images for dark correction.
- **num_projections** (*int*) – Number of projections.
- **angular_range** (*q.deg*) – Range for the scan of the *tomography_motor*.

- **start_angle** (*q.deg*) – Start position of *tomography_motor* for the first projection.

ContinuousTomography

class concert.experiments.synchrotron.**ContinuousTomography**(*self*, *walker*, *flat_motor*, *tomography_motor*, *radio_position*, *flat_position*, *camera*, *shutter*, *num_flats=200*, *num_darks=200*, *num_projections=3000*, *angular_range=<Quantity(180, 'degree')>*, *start_angle=<Quantity(0, 'degree')>*, *separate_scans=True*)

Continuous tomography

await **__ainit__**(*walker*, *flat_motor*, *tomography_motor*, *radio_position*, *flat_position*, *camera*, *shutter*, *num_flats=200*, *num_darks=200*, *num_projections=3000*, *angular_range=<Quantity(180, 'degree')>*, *start_angle=<Quantity(0, 'degree')>*, *separate_scans=True*)

Parameters

- **walker** (concert.storage.Walker) – Walker for storing experiment data.
- **flat_motor** – Motor for moving sample in and out of the beam. Must feature a 'position' property.
- **tomography_motor** (concert.devices.motors.base.ContinuousRotationMotor) – ContinuousRotationMotor for tomography scan.
- **radio_position** – Position of *flat_motor* that the sample is positioned in the beam. Unit must be the same as flat_motor['position'].
- **flat_position** – Position of *flat_motor* that the sample is positioned out of the beam. Unit must be the same as flat_motor['position'].
- **camera** (*concert.devices.camera.base.Camera*) – Camera to acquire the images.
- **shutter** (concert.devices.shutters.base.Shutter) – Stutter
- **num_flats** (*int*) – Number of images for flatfield correction.
- **num_darks** (*int*) – Number of images for dark correction.
- **num_projections** (*int*) – Number of projections.
- **angular_range** (*q.deg*) – Range for the scan of the *tomography_motor*.
- **start_angle** (*q.deg*) – Start position of *tomography_motor* for the first projection.

class concert.experiments.xraytube.**ContinuousTomography**(*self*, *walker*, *flat_motor*, *tomography_motor*, *radio_position*, *flat_position*, *camera*, *xray_tube*, *num_flats=200*, *num_darks=200*, *num_projections=3000*, *angular_range=<Quantity(180, 'degree')>*, *start_angle=<Quantity(0, 'degree')>*, *separate_scans=True*)

Continuous tomography experiment

Parameters

xray_tube (concert.devices.xraytubes.base.XRayTube) – X-ray tube

await __ainit__(*walker*, *flat_motor*, *tomography_motor*, *radio_position*, *flat_position*, *camera*, *xray_tube*, *num_flats=200*, *num_darks=200*, *num_projections=3000*, *angular_range=<Quantity(180, 'degree')>*, *start_angle=<Quantity(0, 'degree')>*, *separate_scans=True*)

Parameters

- **walker** (concert.storage.Walker) – Walker for storing experiment data.
- **flat_motor** – Motor for moving sample in and out of the beam. Must feature a 'position' property.
- **tomography_motor** (concert.devices.motors.base.ContinuousRotationMotor) – ContinuousRotationMotor for tomography scan.
- **radio_position** – Position of *flat_motor* that the sample is positioned in the beam. Unit must be the same as flat_motor['position'].
- **flat_position** – Position of *flat_motor* that the sample is positioned out of the beam. Unit must be the same as flat_motor['position'].
- **camera** (*concert.devices.camera.base.Camera*) – Camera to acquire the images.
- **xray_tube** (concert.devices.xraytubes.base.XRayTube) – X-ray tube
- **num_flats** (*int*) – Number of images for flatfield correction.
- **num_darks** (*int*) – Number of images for dark correction.
- **num_projections** (*int*) – Number of projections.
- **angular_range** (*q.deg*) – Range for the scan of the *tomography_motor*.
- **start_angle** (*q.deg*) – Start position of *tomography_motor* for the first projection.

SteppedSpiralTomography

class concert.experiments.synchrotron.**SteppedSpiralTomography**(*self*, *walker*, *flat_motor*, *tomography_motor*, *vertical_motor*, *radio_position*, *flat_position*, *camera*, *shutter*, *start_position_vertical*, *sample_height*, *vertical_shift_per_tomogram*, *num_flats=200*, *num_darks=200*, *num_projections=3000*, *angular_range=<Quantity(180, 'degree')>*, *start_angle=<Quantity(0, 'degree')>*, *separate_scans=True*)

Stepped spiral tomography

await __ainit__(*walker*, *flat_motor*, *tomography_motor*, *vertical_motor*, *radio_position*, *flat_position*, *camera*, *shutter*, *start_position_vertical*, *sample_height*, *vertical_shift_per_tomogram*, *num_flats=200*, *num_darks=200*, *num_projections=3000*, *angular_range=<Quantity(180, 'degree')>*, *start_angle=<Quantity(0, 'degree')>*, *separate_scans=True*)

Parameters

- **walker** (concert.storage.Walker) – Walker for storing experiment data.
- **flat_motor** – Motor for moving sample in and out of the beam. Must feature a 'position' property.
- **tomography_motor** (concert.devices.motors.base.RotationMotor) – Rotation-Motor for tomography scan.
- **vertical_motor** (concert.devices.motors.base.LinearMotor) – LinearMotor to translate the sample along the tomographic axis.
- **radio_position** – Position of *flat_motor* that the sample is positioned in the beam. Unit must be the same as flat_motor['position'].
- **flat_position** – Position of *flat_motor* that the sample is positioned out of the beam. Unit must be the same as flat_motor['position'].
- **camera** (concert.devices.cameras.base.Camera) – Camera to acquire the images.
- **shutter** (concert.devices.shutters.base.Shutter) – Stutter
- **start_position_vertical** (*q.mm*) – Start position of *vertical_motor*.
- **sample_height** (*q.mm*) – Height of the sample.
- **vertical_shift_per_tomogram** (*q.mm*) – Distance *vertical_motor* is translated during one *angular_range*.
- **num_flats** (*int*) – Number of images for flatfield correction.
- **num_darks** (*int*) – Number of images for dark correction.
- **num_projections** (*int*) – Number of projections.
- **angular_range** (*q.deg*) – Range for the scan of the *tomography_motor*.
- **start_angle** (*q.deg*) – Start position of *tomography_motor* for the first projection.

class concert.experiments.xraytube.**SteppedSpiralTomography**(*self*, *walker*, *flat_motor*, *tomography_motor*, *vertical_motor*, *radio_position*, *flat_position*, *camera*, *xray_tube*, *start_position_vertical*, *sample_height*, *vertical_shift_per_tomogram*, *num_flats=200*, *num_darks=200*, *num_projections=3000*, *angular_range=<Quantity(180, 'degree')>*, *start_angle=<Quantity(0, 'degree')>*, *separate_scans=True*)

Stepped spiral tomography experiment

Parameters

xray_tube (concert.devices.xraytubes.base.XRayTube) – X-ray tube

await **__ainit__**(*walker*, *flat_motor*, *tomography_motor*, *vertical_motor*, *radio_position*, *flat_position*, *camera*, *xray_tube*, *start_position_vertical*, *sample_height*, *vertical_shift_per_tomogram*, *num_flats=200*, *num_darks=200*, *num_projections=3000*, *angular_range=<Quantity(180, 'degree')>*, *start_angle=<Quantity(0, 'degree')>*, *separate_scans=True*)

Parameters

- **walker** (concert.storage.Walker) – Walker for storing experiment data.

- **flat_motor** – Motor for moving sample in and out of the beam. Must feature a 'position' property.
- **tomography_motor** (concert.devices.motors.base.RotationMotor) – RotationMotor for tomography scan.
- **vertical_motor** (concert.devices.motors.base.LinearMotor) – LinearMotor to translate the sample along the tomographic axis.
- **radio_position** – Position of *flat_motor* that the sample is positioned in the beam. Unit must be the same as flat_motor['position'].
- **flat_position** – Position of *flat_motor* that the sample is positioned out of the beam. Unit must be the same as flat_motor['position'].
- **camera** (concert.devices.cameras.base.Camera) – Camera to acquire the images.
- **xray_tube** (concert.devices.xraytubes.base.XRayTube) – X-ray tube
- **start_position_vertical** (*q.mm*) – Start position of *vertical_motor*.
- **sample_height** (*q.mm*) – Height of the sample.
- **vertical_shift_per_tomogram** (*q.mm*) – Distance *vertical_motor* is translated during one *angular_range*.
- **num_flats** (*int*) – Number of images for flatfield correction.
- **num_darks** (*int*) – Number of images for dark correction.
- **num_projections** (*int*) – Number of projections.
- **angular_range** (*q.deg*) – Range for the scan of the *tomography_motor*.
- **start_angle** (*q.deg*) – Start position of *tomography_motor* for the first projection.

ContinuousSpiralTomography

class concert.experiments.synchrotron.**ContinuousSpiralTomography**(*self*, *walker*, *flat_motor*, *tomography_motor*, *vertical_motor*, *radio_position*, *flat_position*, *camera*, *shutter*, *start_position_vertical*, *sample_height*, *vertical_shift_per_tomogram*, *num_flats=200*, *num_darks=200*, *num_projections=3000*, *angular_range=<Quantity(180, 'degree')>*, *start_angle=<Quantity(0, 'degree')>*, *separate_scans=True*)

Continuous spiral tomography

await __ainit__(*walker*, *flat_motor*, *tomography_motor*, *vertical_motor*, *radio_position*, *flat_position*, *camera*, *shutter*, *start_position_vertical*, *sample_height*, *vertical_shift_per_tomogram*, *num_flats=200*, *num_darks=200*, *num_projections=3000*, *angular_range=<Quantity(180, 'degree')>*, *start_angle=<Quantity(0, 'degree')>*, *separate_scans=True*)

Parameters

- **walker** (concert.storage.Walker) – Walker for storing experiment data.
- **flat_motor** – Motor for moving sample in and out of the beam. Must feature a 'position' property.
- **tomography_motor** (concert.devices.motors.base.ContinuousRotationMotor) – ContinuousRotationMotor for tomography scan.
- **vertical_motor** (concert.devices.motors.base.ContinuousLinearMotor) – ContinuousLinearMotor to translate the sample along the tomographic axis.
- **radio_position** – Position of *flat_motor* that the sample is positioned in the beam. Unit must be the same as flat_motor['position'].
- **flat_position** – Position of *flat_motor* that the sample is positioned out of the beam. Unit must be the same as flat_motor['position'].
- **camera** (concert.devices.cameras.base.Camera) – Camera to acquire the images.
- **shutter** (concert.devices.shutters.base.Shutter) – Stutter
- **start_position_vertical** (*q.mm*) – Start position of *vertical_motor*.
- **sample_height** (*q.mm*) – Height of the sample.
- **vertical_shift_per_tomogram** (*q.mm*) – Distance *vertical_motor* is translated during one *angular_range*.
- **num_flats** (*int*) – Number of images for flatfield correction.
- **num_darks** (*int*) – Number of images for dark correction.
- **num_projections** (*int*) – Number of projections.
- **angular_range** (*q.deg*) – Range for the scan of the *tomography_motor*.
- **start_angle** (*q.deg*) – Start position of *tomography_motor* for the first projection.

class concert.experiments.xraytube.**ContinuousSpiralTomography**(*self*, *walker*, *flat_motor*, *tomography_motor*, *vertical_motor*, *radio_position*, *flat_position*, *camera*, *xray_tube*, *start_position_vertical*, *sample_height*, *vertical_shift_per_tomogram*, *num_flats=200*, *num_darks=200*, *num_projections=3000*, *angular_range=<Quantity(180, 'degree')>*, *start_angle=<Quantity(0, 'degree')>*, *separate_scans=True*)

Continuous spiral tomography experiment

Parameters

xray_tube (concert.devices.xraytubes.base.XRayTube) – X-ray tube

await **__ainit__**(*walker*, *flat_motor*, *tomography_motor*, *vertical_motor*, *radio_position*, *flat_position*, *camera*, *xray_tube*, *start_position_vertical*, *sample_height*, *vertical_shift_per_tomogram*, *num_flats=200*, *num_darks=200*, *num_projections=3000*, *angular_range=<Quantity(180, 'degree')>*, *start_angle=<Quantity(0, 'degree')>*, *separate_scans=True*)

Parameters

- **walker** (concert.storage.Walker) – Walker for storing experiment data.
- **flat_motor** – Motor for moving sample in and out of the beam. Must feature a 'position' property.
- **tomography_motor** (concert.devices.motors.base.ContinuousRotationMotor) – ContinuousRotationMotor for tomography scan.
- **vertical_motor** (concert.devices.motors.base.ContinuousLinearMotor) – ContinuousLinearMotor to translate the sample along the tomographic axis.
- **radio_position** – Position of *flat_motor* that the sample is positioned in the beam. Unit must be the same as flat_motor['position'].
- **flat_position** – Position of *flat_motor* that the sample is positioned out of the beam. Unit must be the same as flat_motor['position'].
- **camera** (concert.devices.cameras.base.Camera) – Camera to acquire the images.
- **xray_tube** (concert.devices.xraytubes.base.XRayTube) – X-ray tube
- **start_position_vertical** (*q.mm*) – Start position of *vertical_motor*.
- **sample_height** (*q.mm*) – Height of the sample.
- **vertical_shift_per_tomogram** (*q.mm*) – Distance *vertical_motor* is translated during one *angular_range*.
- **num_flats** (*int*) – Number of images for flatfield correction.
- **num_darks** (*int*) – Number of images for dark correction.
- **num_projections** (*int*) – Number of projections.
- **angular_range** (*q.deg*) – Range for the scan of the *tomography_motor*.
- **start_angle** (*q.deg*) – Start position of *tomography_motor* for the first projection.

GratingInterferometryStepping

In this grating based phase contrast imaging implementation a single projection is generated. The grating is stepped with and without the sample while images are recorded. Dark images are also recorded. If the *concert.experiments.addons.PhaseGratingSteppingFourierProcessing* addon is attached, directly the intensity, visibility and differential phase are reconstructed.

class concert.experiments.synchrotron.**GratingInterferometryStepping**(*self*, *walker*, *camera*, *shutter*, *flat_motor*, *stepping_motor*, *flat_position*, *radio_position*, *grating_period*, *num_darks*, *stepping_start_position*, *num_periods*, *num_steps_per_period*, *propagation_distance*, *separate_scans*)

await **__ainit__**(*walker*, *camera*, *shutter*, *flat_motor*, *stepping_motor*, *flat_position*, *radio_position*, *grating_period*, *num_darks*, *stepping_start_position*, *num_periods*, *num_steps_per_period*, *propagation_distance*, *separate_scans*)

Parameters

- **walker** (concert.storage.DirectoryWalker) – Walker for the experiment
- **camera** (concert.devices.cameras.base.Camera) – Camera to acquire the images
- **shutter** (concert.devices.shutters.base.Shutter) – Shutter
- **flat_motor** – Motor for moving sample in and out of the beam. Must feature a 'position' property.
- **stepping_motor** (concert.devices.motors.base.LinearMotor) –
- **radio_position** – Position of *flat_motor* that the sample is positioned in the beam. Unit must be the same as flat_motor['position'].
- **flat_position** – Position of *flat_motor* that the sample is positioned out of the beam. Unit must be the same as flat_motor['position'].
- **grating_period** (*q.um*) – Periodicity of the stepped grating.
- **num_darks** (*int*) – Number of dark images that are acquired.
- **stepping_start_position** (*q.um*) – First stepping position.
- **num_periods** (*int*) – Number of grating periods that are sampled by the stepping.
- **num_steps_per_period** (*int*) – Number stepping positions per grating period.
- **propagation_distance** (*q.mm*) – Distance between the sample and the analyzer grating. Only used by the processing addon to determine the phase shift in angles.

class concert.experiments.xraytube.**GratingInterferometryStepping**(*self*, *walker*, *camera*, *xray_tube*, *flat_motor*, *stepping_motor*, *flat_position*, *radio_position*, *grating_period*, *num_darks*, *stepping_start_position*, *num_periods*, *num_steps_per_period*, *propagation_distance*, *separate_scans*)

Parameters

xray_tube (concert.devices.xraytubes.base.XRayTube) – X-ray tube

await **__ainit__**(*walker*, *camera*, *xray_tube*, *flat_motor*, *stepping_motor*, *flat_position*, *radio_position*, *grating_period*, *num_darks*, *stepping_start_position*, *num_periods*, *num_steps_per_period*, *propagation_distance*, *separate_scans*)

Parameters

- **walker** (concert.storage.DirectoryWalker) – Walker for the experiment
- **camera** (concert.devices.cameras.base.Camera) – Camera to acquire the images
- **xray_tube** (concert.devices.xraytubes.base.XRayTube) – Xray tube
- **flat_motor** – Motor for moving sample in and out of the beam. Must feature a 'position' property.

- **stepping_motor** (concert.devices.motors.base.LinearMotor) –
- **radio_position** – Position of *flat_motor* that the sample is positioned in the beam. Unit must be the same as flat_motor['position'].
- **flat_position** – Position of *flat_motor* that the sample is positioned out of the beam. Unit must be the same as flat_motor['position'].
- **grating_period** (*q.um*) – Periodicity of the stepped grating.
- **num_darks** (*int*) – Number of dark images that are acquired.
- **stepping_start_position** (*q.um*) – First stepping position.
- **num_periods** (*int*) – Number of grating periods that are sampled by the stepping.
- **num_steps_per_period** (*int*) – Number stepping positions per grating period.
- **propagation_distance** (*q.mm*) – Distance between the sample and the analyzer grating. Only used by the processing addon to determine the phase shift in angles.

Control

Experiment automation based on on-line data analysis.

class concert.experiments.control.**ClosedLoop**

An abstract feedback loop which acquires data, analyzes it on-line and provides feedback to the experiment. The data acquisition procedure is done iteratively until the result of some metric converges to a satisfactory value. Schematically, the class is doing the following in an iterative way:

```
initialize -> measure -> compare -> OK -> success
                ^            |
                |           NOK
                |            |
                -- control <--
```

await compare()

Return True if the metric is satisfied, False otherwise. This is the decision making process.

await control()

React on the result of a measurement.

await initialize()

Bring the experimental setup to some defined initial (reference) state.

await measure()

Conduct a measurement from data acquisition to analysis.

await run(*self*, *max_iterations=10*)

Run the loop until the metric is satisfied, if we don't converge in *max_iterations* then the run is considered unsuccessful and False is returned, otherwise True.

class concert.experiments.control.**DummyLoop**

A dummy optimization loop.

await compare()

Return True if the metric is satisfied, False otherwise. This is the decision making process.

Addons

Addons are special features which are attached to experiments and operate on their data acquisition. For example, to save images on disk:

```
from concert.experiments.addons import ImageWriter

# Let's assume an experiment is already defined
writer = ImageWriter(experiment.acquisitions, experiment.walker)
writer.attach()
# Now images are written on disk
await experiment.run()
# To remove the writing addon
writer.detach()
```

Add-ons for acquisitions are standalone extensions which can be applied to them. They operate on the acquired data, e.g. write images to disk, do tomographic reconstruction etc.

class concert.experiments.addons.**Accumulator**(*acquisitions*, *shapes=None*, *dtype=None*)

An addon which accumulates data.

acquisitions

a list of *Acquisition* objects

shapes

a list of shapes for different acquisitions

dtype

the numpy data type

class concert.experiments.addons.**Addon**(*acquisitions*)

A base addon class. An addon can be attached, i.e. its functionality is applied to the specified *acquisitions* and detached.

acquisitions

A list of *Acquisition* objects. The addon attaches itself on construction.

attach()

Attach the addon to all acquisitions.

detach()

Detach the addon from all acquisitions.

exception concert.experiments.addons.**AddonError**

Addon errors.

class concert.experiments.addons.**Consumer**(*acquisitions*, *consumer*)

An addon which applies a specific coroutine-based consumer to acquisitions.

acquisitions

a list of *Acquisition* objects

consumer

A callable which returns a coroutine which processes the incoming data from acquisitions

class concert.experiments.addons.**ImageWriter**(*acquisitions*, *walker*)

An addon which writes images to disk.

acquisitions

a list of *Acquisition* objects

walker

A *Walker* instance

class concert.experiments.addons.**OnlineReconstruction**(*self*, *experiment*, *reco_args*, *do_normalization=True*, *average_normalization=True*, *walker=None*, *slice_directory='online-slices'*)

exception concert.experiments.addons.**OnlineReconstructionError**

class concert.experiments.addons.**PCOTimestampCheck**(*experiment*)

exception concert.experiments.addons.**PCOTimestampCheckError**

class concert.experiments.addons.**PhaseGratingSteppingFourierProcessing**(*experiment*, *output_directory='contrasts'*)

Addon for concert.experiments.imaging.GratingInterferometryStepping to process the raw data. The order of the acquisitions can be changed.

await process_darks(*producer*)

Processes dark images. All dark images are averaged.

Parameters

producer – Dark image producer

Returns

Running an experiment

To demonstrate how a typical experiment can be run in an empty session with dummy devices:

```
from concert.storage import DirectoryWalker
from concert.ext.viewers import PyplotImageViewer
from concert.experiments.addons import Consumer, ImageWriter
from concert.devices.motors.dummy import LinearMotor, ContinuousRotationMotor
from concert.devices.camera.dummy import Camera
from concert.devices.shutters.dummy import Shutter

# Import experiment
from concert.experiments.synchrotron import ContinuousTomography

# Devices
camera = await Camera()
shutter = await Shutter()
flat_motor = await LinearMotor()
tomo_motor = await ContinuousRotationMotor()
```

(continues on next page)

(continued from previous page)

```
viewer = await PyplotImageViewer()
walker = DirectoryWalker(root="folder to write data")
exp = await ContinuousTomography(walker=walker,
                                 flat_motor=flat_motor,
                                 tomography_motor=tomo_motor,
                                 radio_position=0*q.mm,
                                 flat_position=10*q.mm,
                                 camera=camera,
                                 shutter=shutter)

# Attach live_view to the experiment
live_view = Consumer(exp.acquisitions, viewer)

# Attach image writer to experiment
writer = ImageWriter(exp.acquisitions, walker)

# check all parameters by typing 'exp'

# Run the experiment
f = exp.run()

# Wait until the experiment is done
await f
```

1.3.6 Directors

Directors can be employed to run *concert.experiments.base.Experiment* multiple times. The function `concert.directors.base._prepare_run()` is used to prepare an experiment run. E.g. this function can be used to exchange specimens or modify experiment properties.

The *base.Experiment.ready_to_prepare_next_sample* can be used to trigger the `concert.directors.base._prepare_run()` already while the experiment is still running.

class concert.directors.base.**Director**(*self*, *experiment*)

Class to handle multiple experiment executions.

await __ainit__(*experiment*)

Parameters

experiment (concert.experiments.base.Experiment) – Experiment that is run. If the experiment features a 'ready_to_prepare_next_sample' event (asyncio.Event) this will be waited within the experiment execution. When set() the next iteration will be prepared while the experiment is still running. This could be used to prepare a future iteration while still data is stored or processed. The separate_scans property of the experiment should be set to False, since the director handles the naming of the sub-folders.

await pause()

Waits (after the current iteration is done and the next is prepared) with the next iteration until resume() is called.

await resume()

Resumes a currently paused director run.

XY Scanning

class concert.directors.scanning.**XYScan**(*self*, *experiment*, *x_motor*, *y_motor*, *x_min*, *x_max*, *x_step*, *y_min*, *y_max*, *y_step*)

Director to scan a specimen within a plane.

await **__ainit__**(*experiment*, *x_motor*, *y_motor*, *x_min*, *x_max*, *x_step*, *y_min*, *y_max*, *y_step*)

Parameters

- **experiment** (concert.experiments.base.Experiment) – Experiment that is run. If the experiment features a 'ready_to_prepare_next_sample' event (asyncio.Event) this will be waited within the experiment execution. When set() the next iteration will be prepared while the experiment is still running. This could be used to prepare a future iteration while still data is stored or processed. The separate_scans property of the experiment should be set to False, since the director handles the naming of the sub-folders.
- **x_motor** (concert.devices.motors.base.LinearMotor) – Linear motor for scanning in x direction
- **y_motor** (concert.devices.motors.base.LinearMotor) – Linear motor for scanning in y direction
- **x_min** (*q.mm*) – Starting position of x
- **x_max** (*q.mm*) – Stop position of x
- **x_step** (*q.mm*) – Step width of x scanning
- **y_min** (*q.mm*) – Starting position of y
- **y_max** (*q.mm*) – Stop position of y
- **y_step** (*q.mm*) – Step width of y scanning

CHAPTER

TWO

DEVELOPER DOCUMENTATION

2.1 Development

2.1.1 Writing devices and experiments

Get the code

Concert is developed using Git on the popular GitHub platform. To clone the repository call:

```
$ git clone https://github.com/ufo-kit/concert
```

To get started you are encouraged to install the *development* dependencies via pip:

```
$ cd concert
$ sudo pip install -r requirements.txt
```

After that you can simply install the development source with

```
$ sudo make install
```

Run the tests

The core of Concert is tested using Python's standard library `unittest` module and pytest. To run all tests, you can call pytest directly in the root directory or run make with the `check` argument

```
$ make check
```

Some tests take a lot of time to complete and are marked with the `@slow` decorator. To skip them during regular development cycles, you can run

```
$ make check-fast
```

You are highly encouraged to add new tests when you are adding a new feature to the core or fixing a known bug.

Basic concepts

The core abstraction of Concert is a *Parameter*. A parameter has at least a name but most likely also associated setter and getter callables. Moreover, a parameter can have units and limiters associated with it.

The modules related to device creation are found here

```
concert/
|-- base.py
`-- devices
    |-- base.py
    |-- cameras
    |   |-- base.py
    |   `-- ...
    |-- __init__.py
    |-- motors
    |   |-- base.py
    |   `-- ...
    `-- storagerings
        |-- base.py
        `-- ...
```

Asynchronous constructors

Devices and many other classes in concert subclass `concert.base.AsyncObject` which does not use the classical `def __init__(...)` constructor but an `async def __ainit__(...)`. That is because parameter getters and setters are coroutine funcions (`async def`) and when a Parameterizable instance is created, there is a good chance that some parameters should be read or written and that must be done with the `await param.get()` syntax and that is only possible in coroutine functions, which a normal `__init__` constructor is not. Hence, we introduced a new kind of constructor `__ainit__` which allows such syntax. Inheritance works as usual but if your class inherits from another `AsyncObject` (the base of Parameterizable) *and* a normal class with just an `__init__` contructor, you need to call *both* in your constructor, like this:

```
class Foo(Parameterizable, StandardClass):
    async def __ainit__(self, async_param, sync_param):
        await super().__ainit__(async_param)
        super().__init__(sync_param)
```

Classes subclassing `AsyncObject` cannot define `__init__` constructors, which would lead to ambiguities.

Adding a new device

To add a new device to an existing device class (such as motor, pump, monochromator etc.), a new module has to be added to the corresponding device class package. Inside the new module, the concrete device class must then import the base class, inherit from it and implement all abstract method stubs.

Concert is based on asyncio, see also the *user documentation*. In order for the concurrent execution to work well, all concert code needs to adhere to the concepts of asyncio and the device implementations as well. That means that all methods which actually manipulate the device in any way need to be defined as *async def*. All parameter getters and setters already are defined in this way, and so have to be their underscored implementations (see below).

Let's assume we want to add a new motor called `FancyMotor`. We first create a new module called `fancy.py` in the `concert/devices/motors` directory package. In the `fancy.py` module, we first import the base class

```
from concert.devices.motors.base import LinearMotor
```

Our motor will be a linear one, let's sub-class *LinearMotor*:

```
class FancyMotor(LinearMotor):
    """This is a docstring that can be looked up at run-time by the `ddoc`
    tool."""
```

In order to install all required parameters, we have to call the base constructor. Now, all that's left to do, is implementing the abstract methods that would raise a *AccessorNotImplementedError*:

```
async def _get_position(self):
    # the returned value must have units compatible with units set in
    # the Quantity this getter implements. In this case we just return
    # some stored value
    return self._read_position

async def _set_position(self, position):
    # position is guaranteed to be in the units set by the respective
    # Quantity. In this case just store the desired position in a
    # private variable.
    self._read_position = position
```

We guarantee that setters which implement a *Quantity*, like the `_set_position()` above, obtain the value in the exact same units as they were specified in the respective *Quantity* they implement. E.g. if the above `_set_position()` implemented a quantity with units set in kilometers, the *position* of the `_set_position()` will also be in kilometers. On the other hand the getters do not need to return the exact same quantity but the value must be compatible, so the above `_get_position()` could return millimeters and the user would get the value in kilometers, as defined in the respective *Quantity*.

Parameter setters can be cancelled by hitting *ctrl-c* or *ctrl-k*. If you want a parameter to make some cleanup action after *ctrl-c* is pressed, you should catch the `asyncio.CancelledError` exception, for the motor above you can write:

```
async def _set_position(self, position):
    try:
        self._read_position = position
    except asyncio.CancelledError:
        # cleanup action goes here
        raise   # re-raise the exception if needed
```

And you are guaranteed that when you interrupt the setter the motor stops moving.

Creating a device class

Defining a new device class involves adding a new package to the `concert/devices` directory and adding a new `base.py` class that inherits from *Device* and defines necessary *Parameter* and *Quantity* objects.

In this exercise, we will add a new pump device class. From an abstract point of view, a pump is characterized and manipulated in terms of the volumetric flow rate, e.g. how many cubic millimeters per second of a medium is desired.

First, we create a new `base.py` into the new `concert/devices/pumps` directory and import everything that we need:

```
from concert.quantities import q
from concert.base import Quantity
from concert.devices.base import Device
```

The *Device* handles the nitty-gritty details of messaging and parameter handling, so our base pump device must inherit from it. Furthermore, we have to specify which kind of parameters we want to expose and how we get the values for the parameters (by tying them to getter and setter callables):

```
class Pump(Device):

    flow_rate = Quantity(q.m**3 / q.s,
                         lower=0 * q.m**3 / q.s, upper=1 * q.m**3 / q.s,
                         help="Flow rate of the pump")

    async def __ainit__(self):
        await super(Pump, self).__ainit__()
```

The *flow_rate* parameter can only receive values from zero to one cubic meter per second.

We didn't specify explicit *fget* and *fset* functions, which is why implicit setters and getters called *_set_flow_rate* and *_get_flow_rate* are installed. The real devices then need to implement these. You can however, also specify explicit setters and getters in order to hook into the get and set process:

```
class Pump(Device):

    async def __ainit__(self):
        await super(Pump, self).__ainit__()

    async def _intercept_get_flow_rate(self):
        return await self._get_flow_rate() * 10

    flow_rate = Quantity(q.m**3 / q.s,
                         fget=_intercept_get_flow_rate)
```

Be aware, that in this case you have to list the parameter *after* the functions that you want to refer to.

In case you want to specify the name of the accessor function yourself and rely on implementation by subclasses, you have to raise an *AccessorNotImplementedError*:

```
from concert.base import AccessorNotImplementedError

class Pump(Device):

    ...

    async def _set_flow_rate(self, flow_rate):
        raise AccessorNotImplementedError
```

State machine

A formally defined finite state machine is necessary to ensure and reason about correct behaviour. Concert provides an implicitly defined, decorator-based state machine. The machine can be used to model devices which support hardware state reading but also the ones which don't, thanks to the possibility to store the state in the device itself. To use the state machine you need to declare a *State* object in the base device class and apply the *check()* decorator on each method that changes the state of a device. If you are implementing a device which can read the hardware state you need to define the `_get_state` method. If you are implementing a device which does not support hardware state reading then you need to redefine the *State* in such a way that it has a default value (see the code below) and you can ensure it is changed by respective methods by using the *transition()* decorator on such methods, so that you can keep track of state changes at least in software and comply with transitioning. Examples of such devices could look as follows:

```
from concert.base import Quantity, State, transition, check


class BaseMotor(Device):

    """A base motor class."""

    state = State()
    position = Quantity(q.m)

    @check(source='standby', target='moving')
    async def start(self):
        ...

    async def _start(self):
        # the actual implementation of starting something
        ...


class Motor(BaseMotor):

    """A motor with hardware state reading support."""

    ...

    async def _start(self):
        # Implementation communicates with hardware
        ...

    async def _get_state(self):
        # Get the state from the hardware
        ...


class StatelessMotor(BaseMotor):

    """A motor which doesn't support state reading from hardware."""

    # we have to specify a default value since we cannot get it from
    # hardware
    state = State(default='standby')
```

(continues on next page)

(continued from previous page)

```
    ...

    @transition(target='moving')
    async def _start(self):
        ...
```

The example above explains two devices with the same functionality, however, one supports hardware state reading and the other does not. When they want to `start` the state is checked before the method is executed and afterwards. By checking we mean the current state is checked against the one specified by `source` and the state after the execution is checked against `target`. The `Motor` represents a device which supports hardware state reading. That means all we have to do is to implement `_get_state`. The `StatelessMotor`, on the other hand, has no way of determining the hardware state, thus we need to keep track of it in software. That is achieved by the *transition()* which sets the device state after the execution of the decorated function to `target`. This way the `start` method can look the same for both devices.

Besides single state strings you can also add lists of strings and a catch-all * state that matches all states.

There is no explicit error handling implemented for devices which support hardware state reading but it can be easily modeled by adding error states and reset functions that transition out of them. In case the device does not support state reading and it runs into an error state all you need to do is to raise a *StateError* exception, which has a parameter `error_state`. The exception is caught by *transition()* and the `error_state` parameter is used for setting the device state.

Parameters

In case changing a parameter value causes a state transition, add a *check()* to the *Quantity* object or to the *Parameter* object:

```
class Motor(Device):

    state = State(default='standby')

    velocity = Quantity(q.m / q.s,
                        check=check(source='*', target='moving'))

    foo = Parameter(check=check(source='*', target='*'))
```

Limits

Quantity instances can have user-defined or external limits (e.g. read from a controller). There are `Quantity.lower` and `Quantity.upper` limits and they are obtained in the following way. If `external_lower_getter()` function is specified in the constructor of the quantity, it is used to get the lower limit. If it is not, then the user-defined limit is returned, and that is done either via the `user_lower_getter()` function if specified in the constructor of the quantity, or via the value saved in the quantity, set previousy by `QuantityValue.set_lower()`. The setter calls the `user_lower_setter()` if specified, otherwise just saves the value in a variable inside the quantity. The user-defined getters and setters are useful for invoking mechanisms beyond concert, e.g. updating the limits in a Tango database. The limits can be locked in a similar way to parameter locking.

Creating a experiment class

A new Experiment inherits from *Experiment*. Like the *Device* an experiment class can also hold *Quantity* and *Parameter*. The logger from the *Experiment* will automatically write the values of these in the experiments log file. It also has a state parameter, showing the current experiments state.

Each experiment consist of a set of `Acquisitions`, each generating images. An example experiment with one `Acquisitions` can look like this:

```
class MyExperiment(Experiment):
    num_images = Parameter(help="number of images to acquire")

    async def __ainit__(self, camera, walker):
        self._num_images = 5
        self._camera = camera
        image_acquisition = Acquisition("images", self._acquire_images)
        await super().__init__(acquisitions=[image_acquisition], walker=walker)

    async def _get_num_images(self):
        return self._num_images

    async def _set_num_images(self, n):
        self._num_images = int(n)

    async def _acquire_images(self):
        await self._camera.set_trigger_source("AUTO")
        async with self._camera.recording():
            for i in range(await self.get_num_images()):
                yield await self._camera.grab()
```

2.1.2 Extensions

Concert allows third-party extensions to reside under a common namespace `concert.third.*` similar to the Flask extension system. To achieve this, extensions must be modules or packages named `concert_name` and be installed with setuptools like this:

```
from setuptools import setup

setup(
    name='Concert-Foo',
    version='1.0',
    url='...',
    author='...',
    py_modules=['concert_foo'],
    zip_safe=False,
    install_requires=[
        'concert',
    ]
)
```

After successful installation, the user can import a third-party extension simply like this:

```
from concert.third.foo import SomeClass, some_func
```

2.1.3 Contributing

Reporting bugs

Any bugs concerning the Concert core library and script should be reported as an issue on the GitHub issue tracker.

Fixing bugs or adding features

Bug fixes and new features **must** be in pull request form. Pull request commits should consist of single logical changes and bear a clear message respecting common commit message conventions. Before the change is merged eventually it must be rebased against master.

Bug fixes must come with a unit test that will fail on the bug and pass with the fix. If an issue exists reference it in the branch name and commit message, e.g. `fix-92-remove-foo` and "Fix #92: Remove foo".

New features **must** follow PEP 8 and must be documented thoroughly.

2.2 API reference

2.2.1 Core objects

Parameters

class concert.base.**Parameter**(*fget=None*, *fset=None*, *fget_target=None*, *data=None*, *check=None*, *help=None*)

A parameter with getter and setter.

Parameters are similar to normal Python properties and can additionally trigger state checks. If *fget* or *fset* is not given, you must implement the accessor functions named *_set_name* and *_get_name*:

```
from concert.base import Parameter, State, check

class SomeClass(object):

    state = State(default='standby')
    param = Parameter(check=check(source='standby', target=['standby', 'moving']))

    def _set_param(self, value):
        pass

    def _get_param(self):
        pass
```

When a *Parameter* is attached to a class, you can modify it by accessing its associated *ParameterValue* with a dictionary access:

```
obj = SomeClass()
print(obj['param'])
```

fget is a callable that is called when reading the parameter. *fset* is called when the parameter is written to. *fget_target* is a getter for the target value. *fget*, *fset*, *fget_target* must be member functions of the corresponding Parameterizable object.

data is passed to the state check function.

check is a `check()` that changes states when a value is written to the parameter.

help is a string describing the parameter in more detail.

class concert.base.**ParameterValue**(*instance*, *parameter*)

Value object of a `Parameter`.

await get(*wait_on=None*)

Get coroutine obtaining the concrete *value* of this object.

If *wait_on* is not None, it must be an awaitable on which this method waits.

await get_target(*wait_on=None*)

Get coroutine obtaining target value of this object.

If *wait_on* is not None, it must be an awaitable on which this method waits.

lock(*permanent=False*)

Lock parameter for writing. If *permament* is True the parameter cannot be unlocked anymore.

property locked

Return True if the parameter is locked for writing.

await restore()

Restore the last value saved with `ParameterValue.stash()`.

If the parameter can only be read or no value has been saved, this operation does nothing.

await set(*value*, *wait_on=None*)

Set concrete *value* on the object.

If *wait_on* is not None, it must be an awaitable on which this method waits.

await stash()

Save the current value internally on a growing stack.

If the parameter is writable the current value is saved on a stack and to be later retrieved with `ParameterValue.restore()`.

unlock()

Unlock parameter for writing.

await wait(*value*, *sleep_time=<Quantity(0.1, 'second')>*, *timeout=None*)

Wait until the parameter value is *value*. *sleep_time* is the time to sleep between consecutive checks. *timeout* specifies the maximum waiting time.

property writable

Return True if the parameter is writable.

class concert.base.**Quantity**(*unit*, *fget=None*, *fset=None*, *fget_target=None*, *lower=None*, *upper=None*, *data=None*, *check=None*, *external_lower_getter=None*, *external_upper_getter=None*, *user_lower_getter=None*, *user_lower_setter=None*, *user_upper_getter=None*, *user_upper_setter=None*, *help=None*)

Bases: *Parameter*

A *Parameter* associated with a unit.

fget, *fset*, *data*, *check* and *help* are identical to the *Parameter* constructor arguments.

unit is a Pint quantity. *lower* and *upper* denote soft limits between the *Quantity* values can lie.

class concert.base.**QuantityValue**(*instance*, *quantity*)

Bases: *ParameterValue*

await get(*wait_on=None*)

Get coroutine obtaining the concrete *value* of this object.

If *wait_on* is not None, it must be an awaitable on which this method waits.

lock_limits(*permanent=False*)

Lock limits, if *permanent* is True the limits cannot be unlocked anymore.

await set(*value*, *wait_on=None*)

Set concrete *value* on the object.

If *wait_on* is not None, it must be an awaitable on which this method waits.

unlock_limits()

Unlock limits.

await wait(*value*, *eps=None*, *sleep_time=<Quantity(0.1, 'second')>*, *timeout=None*)

Wait until the parameter value is *value*. *eps* is the allowed discrepancy between the actual value and *value*. *sleep_time* is the time to sleep between consecutive checks. *timeout* specifies the maximum waiting time.

Collection of parameters

class concert.base.**Parameterizable**(*self*)

Collection of parameters.

For each class of type *Parameterizable*, *Parameter* can be set as class attributes

```
class Device(Parameterizable):

    def get_something(self):
        return 'something'

    something = Parameter(get_something)
```

There is a simple *Parameter* and a parameter which models a physical quantity *Quantity*.

A *Parameterizable* is iterable and returns its parameters of type *ParameterValue* or its subclasses

```
for param in device:
    print("name={}".format(param.name))
```

To access a single name parameter object, you can use the `[]` operator:

```
param = device['position']
```

If the parameter name does not exist, a *ParameterError* is raised.

Each parameter value is accessible as a property. If a device has a position it can be read and written with:

```
param.position = 0 * q.mm
print param.position
```

install_parameters(*params*)

Install parameters at run-time.

params is a dictionary mapping parameter names to *Parameter* objects.

lock(*permanent=False*)

Lock all the parameters for writing. If *permanent* is True, the parameters cannot be unlocked anymore.

await restore()

Restore all parameters saved with *Parameterizable.stash()*.

await stash()

Save all writable parameters that can be restored with *Parameterizable.restore()*.

The values are stored on a stacked, hence subsequent saved states can be restored one by one.

unlock()

Unlock all the parameters for writing.

State machine

class concert.base.**State**(*default=None*, *fget=None*, *fset=None*, *data=None*, *check=None*, *help=None*)

Finite state machine.

Use this on a class, to keep some sort of known state. In order to enforce restrictions, you would decorate methods on the class with *check()*:

```
class SomeObject(object):

    state = State(default='standby')

    @check(source='*', target='moving')
    def move(self):
        pass
```

In case your device doesn't provide information on its state you can use the *transition()* to store the state in an instance of your device:

```
@transition(immediate='moving', target='standby')
def _set_some_param(self, param_value):
    # when the method starts device state is set to *immediate*
    # long operation goes here
    pass
    # the state is set to *target* in the end
```

Accessing the state variable will return the current state value, i.e.:

```
obj = SomeObject()
assert obj.state == 'standby'
```

The state cannot be set explicitly by:

```
obj.state = 'some_state'
```

but the object needs to provide methods which transition out of states, the same holds for transitioning out of error states. If the `_get_state()` method is implemented in the device it is always used to get the state, otherwise the state is stored in software.

fget is a callable that is called when reading the parameter. *fset* is called when the parameter is written to. *fget_target* is a getter for the target value. *fget*, *fset*, *fget_target* must be member functions of the corresponding Parameterizable object.

data is passed to the state check function.

check is a *check()* that changes states when a value is written to the parameter.

help is a string describing the parameter in more detail.

concert.base.**check**(*source='*'*, *target='*'*)

Decorates a method for checking the device state.

source denotes the source state that must be present at the time of invoking the decorated method. *target* is the state that the state object will be after successful completion of the method or a list of possible target states.

concert.base.**transition**(*immediate=None*, *target=None*)

Change software state of a device to *immediate*. After the function execution finishes change the state to *target*. On `asyncio.CancelledError`, state is set to *target* and cleanup logic must take place in the callable to be wrapped.

Devices

class concert.devices.base.**Device**(*self*)

Bases: *Parameterizable*

A *Device* provides locked access to a real-world device.

It implements the context protocol to provide locking:

```
async with device:
    # device is locked
    await device.set_parameter(1 * q.m)
    ...

# device is unlocked again
```

await emergency_stop()

Emergency stop.

Asynchronous execution

exception concert.coroutines.base.**WaitError**

Raised on busy waiting timeouts

concert.coroutines.base.**background**(*coroutine*)

Same as *start()*, just meant to be used as a decorator.

concert.coroutines.base.**broadcast**(*producer*, **consumers*)

Feed *producer* to all *consumers*.

await concert.coroutines.base.**ensure_coroutine**(*func*, **args*, ***kwargs*)

func(*args, **kwargs) returns an awaitable which is wrapped here into a real coroutine. This is useful for turuning futures from other libraries, like Tango, into real coroutines.

await concert.coroutines.base.**feed_queue**(*producer*, *func*, **args*)

Feed function *func* with items from *producer* in a separete thread. The signatute must be func(queue, *args) where elements in the queue are instances of *concert.helpers.PrioItem*.

concert.coroutines.base.**get_event_loop**()

Get asyncio's event loop.

concert.coroutines.base.**run_in_executor**(*func*, **args*)

Run a blocking function *func* with signature func(*args) in an executor.

concert.coroutines.base.**run_in_loop**(*coroutine*, *error_msg_if_running=None*)

Wrap *coroutine* into a *asyncio.Task*, run it in the current loop, block until it finishes and return the result. On KeyboardInterrupt, the task is cancelled. Raise RuntimeError with message *error_msg_if_running* in case the loop is already running, otherwise Python will take care of the error reporting.

concert.coroutines.base.**start**(*coroutine*)

Wrap *coroutine* into a task and start its execution right away. The returned task will also be cancellable by ctrl-k.

await concert.coroutines.base.**wait_until**(*condition*, *sleep_time=<Quantity(0.1, 'second')>*, *timeout=None*)

Wait until a callable *condition* returns True. *sleep_time* is the time to sleep between consecutive checks of *condition*. If *timeout* is given and the *condition* doesn't return True within the time specified by it a `WaitingError` is raised.

Exceptions

class concert.base.**UnitError**

Raised when an operation is passed value with an incompatible unit.

class concert.base.**LimitError**

Raised when an operation is passed a value that exceeds a limit.

class concert.base.**ParameterError**(*parameter*)

Raised when a parameter is accessed that does not exists.

class concert.base.**AccessorNotImplementedError**

Raised when a setter or getter is not implemented.

class concert.base.**ReadAccessError**(*parameter*)

Raised when user tries to read a parameter that cannot be read.

class concert.base.**WriteAccessError**(*parameter*)

Raised when user tries to read a parameter that cannot be read.

class concert.base.**StateError**(*error_state*, *msg=None*)

Raised in state check functions of devices.

Configuration

concert.config.**MOTOR_VELOCITY_SAMPLING_TIME**

Time step for calculation of motor velocity by measuring two postion values. Longer values will create more acurate results but reading the velocity will take more time.

concert.config.**PROGRESS_BAR**

Turn on progress bar by long-lasting operations if tqdm package is present

2.2.2 Sessions

concert.session.utils.**abort_awaiting**(*background=False*, *skip=None*)

Abort task currently being awaited in the session. Return True if there is a task being awaited, otherwise False. This function does not touch tasks running in the background unless *background* is True, in which case it cancels all awaitables.

await concert.session.utils.**check_emergency_stop**(*check*, *poll_interval=0.1 * q.s*, *exit_session=False*)

If a callable *check* returns True abort is called. Then until it clears to False nothing is done and then the process begins again. *poll_interval* is the interval at which *check* is called. If *exit_session* is True the session exits when the emergency stop occurs.

concert.session.utils.**code_of**(*func*)

Show implementation of *func*.

concert.session.utils.**ddoc**()

Render device documentation.

concert.session.utils.**dstate**()

Render device state in a table.

concert.session.utils.**get_default_table**(*field_names*, *widths=None*)

Return a prettytable styled for use in the shell. *field_names* is a list of table header strings.

concert.session.utils.**pdoc**(*hide_blacklisted=True*)

Render process documentation.

2.2.3 Networking

Networking package facilitates all network connections, e.g. sockets and Tango.

Socket Connections

class concert.networking.base.**SocketConnection**(*host*, *port*, *return_sequence='\n'*)

A two-way socket connection. *return_sequence* is a string appended after every command indicating the end of it, the default value is a newline (n).

await close()

Close connection.

await connect()

Open connection.

await execute(*data*, *num=1024*)

Execute command and wait for response (coroutine-safe, not thread-safe). Read *num* bytes from the socket.

await recv(*num=1024*)

Read *num* bytes from the socket. The result is first stripped from the trailing return sequence characters and then returned.

await send(*data*)

Send *data* to the peer. The return sequence characters are appended to the data before it is sent.

TANGO

Tango devices are interfaced by PyTango, one can obtain the DeviceProxy by the `get_tango_device()` function.

concert.networking.base.**get_tango_device**(*uri*, *peer=None*, *timeout=<Quantity(10, 'second')>*)

Get a Tango device by specifying its *uri*. If *peer* is given change the tango_host specifying which database to connect to. Format is host:port as a string. *timeout* sets the device's general timeout. It is converted to milliseconds, converted to integer and then the tango device's *set_timout_millis* is called with the converted integer value.

2.2.4 Helpers

class concert.helpers.**Bunch**(*values*)

Encapsulate a list or dictionary to provide attribute-like access.

Common use cases look like this:

```
d = {'foo': 123, 'bar': 'baz'}
b = Bunch(d)
print(b.foo)
>>> 123

l = ['foo', 'bar']
b = Bunch(l)
print(b.foo)
>>> 'foo'
```

class concert.helpers.**ImageWithMetadata**(*input_array*, *metadata: dict | None = None*)

Subclass of numpy.ndarray with a metadata dictionary to hold images its metadata.

class concert.helpers.**PrioItem**(*priority: int*, *data: Any*)

To be used in combination with *queue.PriorityQueue*.

concert.helpers.**arange**(*start*, *stop*, *step*)

This function wraps numpy.arange but strips the units before and adds the unit later at the numpy.array.

Parameters

- **start** (*concert.quantities.q.Quantity*) –
- **stop** (*concert.quantities.q.Quantity*) –
- **step** (*concert.quantities.q.Quantity*) –

Returns

class concert.helpers.**expects**(**args*, ***kwargs*)

Decorator which determines expected arguments for the function and also check correctness of given arguments. If input arguments differ from expected ones, exception *TypeError* will be raised.

For numeric arguments use *Numeric* class with 2 parameters: dimension of the array and units (optional). E.g. "Numeric (1)" means function expects one number or "Numeric (2, q.mm)" means function expects expression like [4,5]*q.mm

Common use case looks like this:

```
from concert.helpers import Numeric

@expects(Camera, LinearMotor, pixelsize = Numeric(2, q.mm))
def foo(camera, motor, pixelsize = None):
    pass
```

concert.helpers.**is_iterable**(*item*)

Is *item* iterable or not.

concert.helpers.**linspace**(*start*, *stop*, *num*, *endpoint=True*)

This function wraps numpy.linspace but strips the units before and adds the unit later at the numpy.array.

Parameters

- **start** (*concert.quantities.q.Quantity*) – First value
- **stop** (*concert.quantities.q.Quantity*) –
- **num** (*int*) –
- **endpoint** (*bool*) –

Returns

numpy.array with the length *num* and entries equally distributed within *start* and *stop*.

concert.helpers.**measure**(*func=None*, *return_result=False*)

Measure and print execution time of *func*.

If *return_result* is True, the decorated function returns a tuple consisting of the original return value and the measured time in seconds.

concert.helpers.**memoize**(*func*)

Memoize the result of *func*.

Remember the result of *func* depending on its arguments. Note, that this requires that the function is free from any side effects, e.g. returns the same value given the same arguments.

2.2.5 Storage

Storage implementations.

class concert.storage.**DirectoryWalker**(*writer=<class 'concert.writers.TiffWriter'>*, *dsetname='frame_{:>06}.tif'*, *start_index=0*, *bytes_per_file=0*, *root=None*, *log=None*, *log_name='experiment.log'*)

A DirectoryWalker moves through a file system and writes flat files using a specific filename template.

Use *writer* to write data to files with filenames with a template from *dsetname*. *start_index* specifies the number in the first file name, e.g. for the default *dsetname* and *start_index* 100, the first file name will be frame_000100.tif.

exists(**paths*)

Check if *paths* exist.

class concert.storage.**DummyWalker**(*root=''*)

Constructor. *root* is the topmost level of the data structure.

exists(**paths*)

Return True if path from current position specified by a list of *paths* exists.

exception concert.storage.**StorageError**

Exceptions related to logical issues with storage.

class concert.storage.**Walker**(*root*, *dsetname='frames'*, *log=None*, *log_handler=None*)

A Walker moves through an abstract hierarchy and allows to write data at a specific location.

Constructor. *root* is the topmost level of the data structure.

ascend()

Ascend from current depth and return *self*.

create_writer(*producer*, *name=None*, *dsetname=None*)

Create a writer coroutine for writing data set *dsetname* with images from *producer* inside. If *name* is given, descend to it first and once the writer is created ascend back. This way, the writer can operate in *name* and the walker can be safely used to move around and create other writers elsewhere while the created writer is working. The returned coroutine is not guaranteed to be wrapped into a `asyncio.Task`, hence to be started immediately. This function also does not block after creating the writer. This is useful for splitting the preparation of writing (creating directories, ...) and the I/O itself.

property current

Return current position.

descend(*name*)

Descend to *name* and return *self*.

exists(**paths*)

Return True if path from current position specified by a list of *paths* exists.

home()

Return to root.

await write(*producer*, *dsetname=None*)

Create a coroutine for writing data set *dsetname* with images from *producer*. The execution starts immediately in the background and await will block until the images are written.

concert.storage.**create_directory**(*directory*, *rights=488*)

Create *directory* and all paths along the way if necessary.

concert.storage.**read_image**(*filename*)

Read image from file with *filename*. The file type is detected automatically.

concert.storage.**read_tiff**(*file_name*)

Read tiff file from disk by `tifffile` module.

concert.storage.**split_dsetformat**(*dsetname*)

Strip *dsetname* off the formatting part wihch leaves us with the data set name.

concert.storage.**write_images**(*pqueue*, *writer=<class 'concert.writers.TiffWriter'>*, *prefix='image_{:>05}.tif'*, *start_index=0*, *bytes_per_file=0*)

write_images(pqueue, writer=TiffWriter, prefix="image_{:>05}.tif", start_index=0, bytes_per_file=0)

Write images on disk with specified *writer* and file name *prefix*. Write to one file until the *bytes_per_file* bytes has been written. If it is 0, then one file per image is created. *writer* is a subclass of `writers.ImageWriter`. *start_index* specifies the number in the first file name, e.g. for the default *prefix* and *start_index* 100, the first file name will be image_00100.tif. If *prefix* is not formattable images are appended to the filename specified by *prefix*.

concert.storage.**write_libtiff**(*file_name*, *data*)

Write a TIFF file using pylibtiff. Return the written file name.

concert.storage.**write_tiff**(*file_name*, *data*)

The default TIFF writer which uses `tifffile` module. Return the written file name.

2.2.6 Device classes

Cameras

A *Camera* can be set via the device-specific properties that can be set and read with `Parameter.set()` and `Parameter.get()`. Moreover, a camera provides means to

- *start_recording()* frames,
- *stop_recording()* the acquisition,
- *trigger()* a frame capture and
- *grab()* to get the last frame.

Camera triggering is specified by the `trigger_source` parameter, which can be one of

- `camera.trigger_sources.AUTO` means the camera triggers itself automatically, the frames start being recorded right after the `start_recording()` call and stop being recorded by `stop_recording()`
- `Camera.trigger_sources.SOFTWARE` means the camera needs to be triggered by the user by *trigger()*. This way you have complete programatic control over when is the camera triggered, example usage:

```
camera.trigger_source = camera.trigger_sources.SOFTWARE
start_recording(camera)
trigger(camera)
long_operation()
# Here we get the frame from before the long operation
grab(camera)
```

- `Camera.trigger_sources.EXTERNAL` is a source when the camera is triggered by an external low-level signal (such as TTL). This source provides very precise triggering in terms of time synchronization with other devices

To setup and use a camera in a typical environment, you would do:

```
import numpy as np
from concert.devices.cameras.uca import Camera

camera = Camera('pco')
camera.trigger_source = camera.trigger_sources.SOFTWARE
camera.exposure_time = 0.2 * q.s
start_recording(camera)
trigger(camera)
data = grab(camera)
stop_recording(camera)

print("mean=%f, stddev=%f" % (np.mean(data), np.std(data)))
```

You can apply primitive operations to the frames obtained by *Camera.grab()* by setting up a `Camera.convert` attribute to some callable which takes just one argument which is the grabbed frame. The callable is applied to the frame and the converted one is returned by *Camera.grab()*. You can do:

```
import numpy as np
from concert.devices.cameras.dummy import Camera

camera = Camera()
camera.convert = np.fliplr
# The frame is left-right flipped
grab(camera)
```

class concert.devices.cameras.base.**BufferedMixin**(*self*)

Bases: *Device*

A camera that stores the frames in an internal buffer

class concert.devices.cameras.base.**Camera**(*self*)

Bases: *Device*

Base class for remotely controllable cameras.

frame-rate

Frame rate of acquisition in q.count per time unit.

await grab() → *ImageWithMetadata*

Return a concert.storage.ImageWithMetadata (subclass of np.ndarray) with data of the current frame.

async with recording()

A context manager for starting and stopping the camera.

In general it is used with the `async with` keyword like this:

```
async with camera.recording():
    frame = await camera.grab()
```

await start_recording()

Start recording frames.

await stop_recording()

Stop recording frames.

async for ... in stream()

Grab frames continuously yield them. This is an async generator.

await trigger()

Trigger a frame if possible.

exception concert.devices.cameras.base.**CameraError**

Bases: Exception

Camera specific errors.

class concert.devices.cameras.uca.**Camera**(*self*, *name*, *params=None*)

libuca-based camera.

All properties that are exported by the underlying camera are also visible.

await __ainit__(*name*, *params=None*)

Create a new libuca camera.

The *name* is passed to the uca plugin manager.

Raises

CameraError – In case camera *name* does not exist.

await grab(*index=None*)

Return a concert.storage.ImageWithMetadata (subclass of np.ndarray) with data of the current frame.

write(*name*, *data*)

Write NumPy array *data* for *name*.

class concert.devices.cameras.dummy.**Camera**(*self*, *background=None*, *simulate=True*)

A simple dummy camera.

await __ainit__(*background=None*, *simulate=True*)

background can be an array-like that will be used to generate the frame when calling grab. If *simulate* is True the final image intensity will be scaled based on exposure time and poisson noise will be added. If *simulate* is False, the background will be returned with no modifications to it.

Grippers

A gripper can grip and release objects.

class concert.devices.grippers.base.**Gripper**(*self*)

Bases: *Device*

Base gripper class.

await grip()

Grip an object.

await release()

Release an object.

I/O

class concert.devices.io.base.**Signal**(*self*)

Bases: *Device*

Base device for binary signals, e.g. TTL trigger signals and similar.

await off()

Switch the signal off.

await on()

Switch the signal on.

await trigger(*duration=10 * q.ms*)

Generate a trigger signal of *duration*.

class concert.devices.io.base.**IO**(*self*)

Bases: *Device*

The IO device consists of ports which can be readable, writable or both.

property ports

Port IDs used by *read_port()* and *write_port()*

await read_port(*port*)

Read a *port*.

await write_port(*port*, *value*)

Write a *value* to the *port*.

class concert.devices.io.dummy.**IO**(*self*, *port_value=0*)

Dummy I/O device implementation.

Lightsources

class concert.devices.lightsources.base.**LightSource**(*self*)

Bases: *Device*

A base LightSource class.

class concert.devices.lightsources.dummy.**LightSource**(*self*)

A dummy light source

Monochromators

class concert.devices.monochromators.base.**Monochromator**(*self*)

Bases: *Device*

Monochromator device which is used to filter the beam in order to get a very narrow energy bandwidth.

energy

Monochromatic energy in electron volts.

wavelength

Monochromatic wavelength in meters.

class concert.devices.monochromators.doublemonochromator.**Monochromator**(*self*, *motor_2*)

Bases: *Monochromator*

Base implementation of a monochromator with the ability to scan a second crystal/multilayer.

await **__ainit__**(*motor_2*)

Parameters
motor_2 (concert.devices.motors.base.RotationMotor) – Motor controlling the tilt of the second crystal or multilayer

get_last_tune_scan()

Shows a plot of the last tuning scan.

await **scan_bragg_angle**(*diode*, *plot_callback=None*, *n_points=50*, *tune_range=<Quantity(0.025, 'degree')>*, *center_point=None*)

Scans the second crystal or multilayer. After the scan, the motor is moved back to its initial position.

A scan can be shown afterwards with show_tune_scan(). To move the motor to the maximum call `select_maximum()` and to go to the center of mass `select_center_of_mass()`.

Parameters

- **diode** (*concert.devices.photodiodes.base.Diode*) – Diode to measure the intensity
- **plot_callback** – Function to plot the scanned intensity. Could be an instance of a PyplotViewer. If set to *None* the values of the scan are returned.
- **n_points** (*int*) – Number of points, equally distributed between the current_angle - tune_range/2 and current_angle + tube_range/2
- **tune_range** (*q.deg*) – Range to scan for maximum.
- **center_point** (*q.deg*) – central point around which the scan is performed. If set to *None* the current position of the scanning motor is used.

await **select_center_of_mass**()

Moves bragg2 to the center of mass of the last tuning scan.

await **select_maximum**()

Moves bragg2 to the maximum of the last tuning scan.

class concert.devices.monochromators.dummy.**Monochromator**(*self*)

Monochromator class implementation.

class concert.devices.monochromators.dummy.**DoubleMonochromator**(*self*)

Double monochromator implementation

await **__ainit__**()

Parameters
motor_2 (concert.devices.motors.base.RotationMotor) – Motor controlling the tilt of the second crystal or multilayer

Motors

Linear

Linear motors are characterized by moving along a straight line.

class concert.devices.motors.base.**LinearMotor**(*self*)

Bases: _PositionMixin

One-dimensional linear motor.

position

Position of the motor in length units.

class concert.devices.motors.base.**ContinuousLinearMotor**(*self*)

Bases: *LinearMotor*, _VelocityMixin

One-dimensional linear motor with adjustable velocity.

velocity

Current velocity in length per time unit.

class concert.devices.motors.dummy.**LinearMotor**(*self*, *position=None*, *upper_hard_limit=None*, *lower_hard_limit=None*)

A linear step motor dummy.

class concert.devices.motors.dummy.**ContinuousLinearMotor**(*self*, *position=None*, *upper_hard_limit=None*, *lower_hard_limit=None*)

A continuous linear motor dummy.

Rotational

Rotational motors are characterized by rotating around an axis.

class concert.devices.motors.base.**RotationMotor**(*self*)

Bases: _PositionMixin

One-dimensional rotational motor.

position

Position of the motor in angular units.

class concert.devices.motors.base.**ContinuousRotationMotor**(*self*)

Bases: *RotationMotor*, _VelocityMixin

One-dimensional rotational motor with adjustable velocity.

velocity

Current velocity in angle per time unit.

class concert.devices.motors.dummy.**RotationMotor**(*self*, *upper_hard_limit=None*, *lower_hard_limit=None*)

A rotational step motor dummy.

class concert.devices.motors.dummy.**ContinuousRotationMotor**(*self*, *position=None*, *upper_hard_limit=None*, *lower_hard_limit=None*)

A continuous rotational step motor dummy.

Axes

An axis is a coordinate system axis which can realize either translation or rotation, depending by which type of motor it is realized.

class concert.devices.positioners.base.**Axis**(*self*, *coordinate*, *motor*, *direction=1*, *position=None*)

Bases: AsyncObject

An axis represents a Euclidean axis along which one can translate or around which one can rotate. The axis *coordinate* is a string representing the Euclidean axis, i.e. 'x' or 'y' or 'z'. Movement is realized by a *motor*. An additional *position* argument is necessary for calculatin more complicated motion types, e.g. rotation around arbitrary point in space. It is the local position with respect to a *concert.devices.positioners.base.Positioner* in which it is placed.

await get_position()

Get position asynchronously with respect to axis direction.

await set_position(*position*)

Set the *position* asynchronously with respect to axis direction.

Photodiodes

Photodiodes measure light intensity.

class concert.devices.photodiodes.base.**PhotoDiode**(*self*)

Bases: *Device*

Impementation of photo diode with V output signal

class concert.devices.photodiodes.dummy.**PhotoDiode**(*self*)

A dummy photo diode

Positioners

Positioner is a device consisting of more *concert.devices.positioners.base.Axis* instances which make it possible to specify a 3D position and orientation of some object.

class concert.devices.positioners.base.**Positioner**(*self*, *axes*, *position=None*)

Bases: *Device*

Combines more motors which move to form a complex motion. *axes* is a list of *Axis* instances. *position* is a 3D vector of coordinates specifying the global position of the positioner.

If a certain coordinate in the positioner is missing, then when we set the position or orientation we can specify the respective vector position to be zero or numpy.nan.

await back(*value*)

Move back by *value*.

await down(*value*)

Move down by *value*.

await forward(*value*)

Move forward by *value*.

await left(*value*)

Move left by *value*.

await move(*position*)

Move by specified *position*.

await right(*value*)

Move right by *value*.

await rotate(*angles*)

Rotate by *angles*.

await up(*value*)

Move up by *value*.

class concert.devices.positioners.dummy.**Positioner**(*self*, *position=None*)

A dummy positioner.

Imaging Positioners

Imaging positioner is a positioner capable of moving in *x* and *y* directions by the given amount of pixels.

class concert.devices.positioners.imaging.**Positioner**(*self*, *axes*, *detector*, *position=None*)

Bases: *Positioner*

A positioner which takes into account a detector with some pixel size. This way the user can specify the movement in pixels.

await move(*position*)

Move by specified *position* which can be given in meters or pixels.

class concert.devices.positioners.dummy.**ImagingPositioner**(*self*, *detector=None*, *position=None*)

A dummy imaging positioner.

Pumps

class concert.devices.pumps.base.**Pump**(*self*)

Bases: *Device*

A pumping device.

await start()

Start pumping.

await stop()

Stop pumping.

class concert.devices.pumps.dummy.**Pump**(*self*)

A dummy pump.

Sample changers

class concert.devices.samplechangers.base.**SampleChanger**(*self*)

Bases: *Device*

A device that moves samples in and out from the sample holder.

Scales

class concert.devices.scales.base.**Scales**(*self*)

Bases: *Device*

Base scales class.

class concert.devices.scales.base.**TarableScales**(*self*)

Bases: *Scales*

Scales which can be tared.

await tare()

Tare the scales.

class concert.devices.scales.dummy.**Scales**(*self*)

A dummy scale.

Shutters

class concert.devices.shutters.base.**Shutter**(*self*)

Bases: *Device*

Shutter device class implementation.

await close()

Close the shutter.

await open()

Open the shutter.

class concert.devices.shutters.dummy.**Shutter**(*self*)

A dummy shutter that can be opened and closed.

Storage rings

class concert.devices.storagerings.base.**StorageRing**(*self*)

Bases: *Device*

Read-only access to storage ring information.

current

Ring current

energy

Ring energy

lifetime

Ring lifetime in hours

class concert.devices.storagerings.dummy.**StorageRing**(*self*)

A storage ring dummy.

X-ray tubes

class concert.devices.xraytubes.base.**XRayTube**(*self*)

Bases: *Device*

A base x-ray tube class.

await off()

Disables the x-ray tube.

await on()

Enables the x-ray tube.

2.2.7 Processes

Scanning

async for ... in concert.processes.common.**scan**(*params*, *values*, *feedback*, *go_back=False*)

Multi-dimensional scan of *concert.base.Parameter* instances *params*, which are set to *values*. *feedback* is a coroutine function without parameters called after every iteration. If *go_back* is True, the original parameter values are restored at the end.

If *params* is just one parameter and *values* is one list of values, perform a 1D scan. In this case tuples (x, y) are returned where x are the individual elements from the list of *values* and y = feedback() is called after every value setting.

If *params* is a list of parameters and *values* is a list of lists, assign values[i] to params[i] and do a multi-dimensional scan, where last parameter changes the fastest (in other words a nested scan of all parameters, where *feedback* is called for every combination of parameter values. The combinations are obtained as a cartesian product of the *values*. For example, scanning camera exposure times and motor positions with values=[[1, 2] * q.s, [3, 5] * q.mm], would result in this:

```
[((1 * q.s, 3 * q.mm), feedback()), ((1 * q.s, 5 * q.mm), feedback()),
 ((2 * q.s, 3 * q.mm), feedback()), ((2 * q.s, 5 * q.mm), feedback())]
```

In general, for n parameters and lists of values, returned are tuples ((x_0, . . . , x_{n-1}), y), where y = feedback() is called after every value setting (any parameter change). Parameter setting occurs in parallel, is waited for and then *feedback* is called.

A simple 1D example:

```
async for vector in scan(camera['exposure_time'], np.arange(1, 10, 1) * q.s,␣
↪feedback):
    print(vector) # prints (1 * q.s, feedback()) and so on
```

2D example:

```
params = [camera['exposure_time'], motor['position']]
values = [np.arange(1, 10, 1) * q.s, np.arange(5, 15, 2) * q.mm]
async for vector in scan(params, values, feedback):
    print(vector) # prints ((1 * q.s, 5 * q.mm), feedback()) and so on
```

async for ... in concert.processes.common.**ascan**(*param*, *start*, *stop*, *step*, *feedback*, *go_back=False*, *include_last=True*)

A convenience function to perform a 1D scan on parameter *param*, scan from *start* value to *stop* with *step*. *feedback* and *go_back* are the same as in the *scan()*. If *include_last* is True, the *stop* value will be included in the created values This function just computes the values from *start*, *stop*, *step* and then calls *scan()*:

```
scan(param, values, feedback=feedback, go_back=go_back))
```

async for ... in concert.processes.common.**dscan**(*param*, *delta*, *step*, *feedback*, *go_back=False*, *include_last=True*)

A convenience function to perform a 1D scan on parameter *param*, scan from its current value to some *delta* with *step*. *feedback* and *go_back* are the same as in the *scan()*. This function just computes the start and stop values and calls *ascan()*:

```
start = await param.get()
ascan(param, start, start + delta, step, feedback, go_back=go_back)
```

Focusing

await concert.processes.common.**focus**(*camera*, *motor*, *measure=<function std>*, *opt_kwargs=None*, *plot_callback=None*, *frame_callback=None*)

Focus *camera* by moving *motor*. *measure* is a callable that computes a scalar that has to be maximized from an image taken with *camera*. *opt_kwargs* are keyword arguments sent to the optimization algorithm. *plot_callback* is (x, y) values, where x is the iteration number and y the metric result. *frame_callback* is a coroutine function fed with the incoming frames.

This function is returning a future encapsulating the focusing event. Note, that the camera is stopped from recording as soon as the optimal position is found.

Alignment

await concert.processes.common.**align_rotation_axis**(*camera*, *rotation_motor*, *x_motor=None*, *z_motor=None*, *get_ellipse_points=<function find_needle_tips>*, *num_frames=10*, *metric_eps=None*, *position_eps=<Quantity(0.1, 'degree')>*, *max_iterations=5*, *initial_x_coeff=<Quantity(1, 'dimensionless')>*, *initial_z_coeff=<Quantity(1, 'dimensionless')>*, *shutter=None*, *flat_motor=None*, *flat_position=None*, *y_0=0*, *y_1=None*, *get_ellipse_points_kwargs=None*, *frame_consumers=None*)

align_rotation_axis(camera, rotation_motor, x_motor=None, z_motor=None, get_ellipse_points=find_needle_tips, num_frames=10, metric_eps=None, position_eps=0.1 * q.deg,

max_iterations=5, initial_x_coeff=1 * q.dimensionless, initial_z_coeff=1 * q.dimensionless, shutter=None, flat_motor=None, flat_position=None, y_0=0, y_1=None, get_ellipse_points_kwargs=None, frame_consumers=None)

Align rotation axis. *camera* is used to obtain frames, *rotation_motor* rotates the sample around the tomographic axis of rotation, *x_motor* turns the sample around x-axis, *z_motor* turns the sample around z-axis.

get_ellipse_points is a function with one positional argument, a set of images. It computes the ellipse points from the sample positions as it rotates around the tomographic axis. You can use e.g. *concert.imageprocessing.find_needle_tips()* and *concert.imageprocessing.find_sphere_centers()* to extract the ellipse points from needle tips or sphere centers. You can pass additional keyword arguments to the *get_ellipse_points* function in the *get_ellipse_points_kwargs* dictionary.

num_frames defines how many frames are acquired and passed to the *measure*. *metric_eps* is the metric threshold for stopping the procedure. If not specified, it is calculated automatically to not exceed 0.5 pixels vertically. If *max_iterations* is reached the procedure stops as well. *initial_[x|z]_coeff* is the coefficient applied` to the motor motion for the first iteration. If we move the camera instead of the rotation stage, it is often necessary to acquire fresh flat fields. In order to make an up-to-date flat correction, specify *shutter* if you want fresh dark fields and specify *flat_motor* and *flat_position* to acquire flat fields. Crop acquired images to *y_0* and *y_1*. *frame_consumers* are coroutine functions which will be fed with all acquired frames.

The procedure finishes when it finds the minimum angle between an ellipse extracted from the sample movement and respective axes or the found angle drops below *metric_eps*. The axis of rotation after the procedure is (0,1,0), which is the direction perpendicular to the beam direction and the lateral direction. *x_motor* and *z_motor* do not have to move exactly by the computed angles but their relative motion must be linear with respect to computed angles (e.g. if the motors operate with steps it is fine, also rotation direction does not need to be known).

2.2.8 Coroutines

Sinks

class concert.coroutines.sinks.**Accumulate**(*shape=None*, *dtype=None*, *reset_on_call=True*)

Accumulate items in a list or a numpy array if *shape* is given, *dtype* is the data type. If *reset_on_call* is True, the saved values will be overwritten every time the accumulator is called, otherwise they will be appended.

class concert.coroutines.sinks.**Result**

The object is callable and when called it becomes a coroutine which accepts items and stores them in a variable which allows the user to obtain the last stored item at any time point.

await concert.coroutines.sinks.**null**(*producer*)

A black-hole.

Filters

class concert.coroutines.filters.**Timer**

Timer object measures execution times of coroutine-based workflows. It measures the time from when this object receives data until all the subsequent stages finish.

property duration

All iterations summed up.

property mean

Mean iteration execution time.

reset()

Reset the timer.

async for ... in concert.coroutines.filters.**absorptivity**(*producer*)

Get the absorptivity from a flat corrected stream of images. The intensity after the object is defined as $I = I_0 \cdot e^{-\mu t}$ and we extract the absorptivity μt from the stream of flat corrected images I/I_0.

async for ... in concert.coroutines.filters.**average_images**(*producer*)

Average images as they come from *producer*.

async for ... in concert.coroutines.filters.**downsize**(*producer*, *x_slice=None*, *y_slice=None*, *z_slice=None*)

Downsize images in 3D. Every argument is either a tuple (start, stop, step). *x_slice* operates on image width, *y_slice* on its height and *z_slice* on the incoming images, i.e. it creates the third time dimension.

Note: the *start* index is included in the data and the *stop* index is excluded.

async for ... in concert.coroutines.filters.**flat_correct**(*flat*, *producer*, *dark=None*)

Flat correcting coroutine which takes a *flat* field and a *dark* field (if given) from *producer* and calculates a flat corrected radiograph.

async for ... in concert.coroutines.filters.**stall**(*producer*, *per_shot=10*, *flush_at=None*)

Send items once enough is collected from *producer*. Collect *per_shot* items. The incoming data might represent a collection of some kind. If the last item is supposed to be sent regardless the current number of collected items, use *flush_at* by which you specify the collection size and every time the current item *counter* % *flush_at* == 0 the item is sent.

2.2.9 Optimization

Optimization is a procedure to iteratively find the best possible match to

$$y = f(x).$$

This module provides execution routines and algorithms for optimization.

exception concert.optimization.**OptimizationError**

Optimization-related errors.

await concert.optimization.**halver**(*function*, *x_0*, *initial_step=None*, *epsilon=None*, *max_iterations=100*)

Halving the interval, evaluate *function* based on *param*. Use *initial_step*, *epsilon* precision and *max_iterations*.

await concert.optimization.**optimize**(*function*, *x_0*, *algorithm*, *alg_args=()*, *alg_kwargs=None*, *callback=None*)

Optimize y = await *function* (x), so *function* must be a coroutine function. *x_0* is the initial guess. *algorithm* is the optimization algorithm to be used:

```
algorithm(x_0, *alg_args, **alg_kwargs)
```

callback is a callable called with all the (x, y) values as they are obtained.

await concert.optimization.**optimize_parameter**(*parameter*, *feedback*, *x_0*, *algorithm*, *alg_args=()*, *alg_kwargs=None*, *callback=None*)

Optimize *parameter* and use the *feedback* (a coroutine function) as a result. Other arguments are the same as by *optimize()*. The function to be optimized is determined as follows:

```
await parameter.set(x)
y = await feedback()
```

callback is the same as by *optimize()*.

await concert.optimization.**scipy_minimize**(*func*, *x_0*, ***kwargs*)

Use scipy.optimize.minimize(), *func* is a coroutine function, the translation to scipy is taken care of here. *x_0* is the initial guess and *kwargs* are passed to *minimize*.

2.2.10 Image processing

Image processing module for manipulating image data, e.g. filtered backprojection, flat field correction and other operations on images.

concert.imageprocessing.**center_of_mass**(*frame*)

Calculates the center of mass of the whole frame wheighted by value.

concert.imageprocessing.**center_of_points**(*points*)

Find a simplified center of mass withouth point-weighing from a set of *points*.

concert.imageprocessing.**compute_pearson_correlation_coefficient**(*first*, *second*, *dx*, *dy*)

Compute Pearson correlation coefficient. Image *second* is shifted by *dx* and *dy* pixels and the correlation is computed with respect to image *first*. Both images are cropped with respect to the *dx*, *dy* shift in order not to correlate regions overflowing over image edges.

concert.imageprocessing.**compute_rotation_axis**(*first_projection*, *last_projection*)

Compute the tomographic rotation axis based on cross-correlation technique. *first_projection* is the projection at 0 deg, *last_projection* is the projection at 180 deg.

concert.imageprocessing.**correlate**(*first*, *second*, *first_y=0*, *second_y=0*, *overlap_height=None*, *supersampling=1*)

Correlate *first* and *second* image, use *supersampling* for sub-pixel precision. Crop first image vertically to (*first_y*, *first_y* + *overlap_height*) and second to (*second_y*, *second_y* + *overlap_height*).

concert.imageprocessing.**filter_low_frequencies**(*data*, *fwhm=32.0*)

Filter low frequencies in 1D *data*. *fwhm* is the FWHM of the gaussian used to filter out low frequencies in real space. The window is then computed as fft(1 - gauss).

concert.imageprocessing.**find_needle_tip**(*image*)

Extract needle tip from *image*.

await concert.imageprocessing.**find_needle_tips**(*producer*)

Get sample tips in images from *producer*.

await concert.imageprocessing.**find_sphere_centers**(*producer*, *supersampling=1*, *correlation_threshold=None*)

Get sphere centers in images from *producer*. by finding the image with the largest portion of a sphere inside (the sphere may partially go out of the FOV) and correlate other images with the found one, from which relative shifts are computed and converted to absolute sphere centers. This is done by first computing the center of mass of the best image and then subtracting the respective shifts. Use *supersampling* for sub-pixel precision and filter out the centers for which the correlation coefficient computed by *compute_pearson_correlation_coefficient()* is worse than *correlation_threshold*. The correlation coefficient is computed by shifting an image based on the shift found by correlation and computing the correlation coefficient of such shifted image with respect to the best one.

await concert.imageprocessing.**find_sphere_centers_by_mass**(*producer*, *border_crossing_ok=True*)

Get sphere centers in images from *producer* by computing their center of mass. The images must be absorption images. If *border_crossing_ok* is False skip images where sphere goes outside the field of view.

concert.imageprocessing.**flat_correct**(*radio*, *flat*, *dark=None*)

Flat field correction of a radiograph *radio* with *flat* field. If *dark* field is supplied it is taken into account as well.

concert.imageprocessing.**normalize**(*image*, *minimum=0.0*, *maximum=1.0*)

Normalize *image* intensities to start at *minimum* and end at *maximum*.

concert.imageprocessing.**ramp_filter**(*width*)

Get a 1D ramp filter for filtering sinogram rows.

concert.imageprocessing.**segment_convex_object**(*image*)

Extract convex object from *image* (e.g. needle or sphere). It doesn't matter if object is brigher or darker than the background (e.g. non flat corrected radiograph on input).

2.2.11 Experiments

There are abstract implementations for radiography, stepped tomography, continuous tomography, stepped spiral tomography and continuous spiral tomography.

All of them implement *Acquisition* for dark images (without beam), flat field images (with beam, but sample moved to Experiment.flatfield_position) and projections of the sample according to the measurement scheme.

In each acquisition generator the functions `_prepare_flats()`, `_finish_flats()`, `_prepare_darks()`, `_finish_darks()`, `_prepare_radios()`, `_finish_radios()` are called. Overwriting them allows an easy way to implement special features within the experiments.

To use the classes one has to implement the *start_sample_exposure()* and *stop_sample_exposure()* accordingly (see `concert.experiments.synchrotron.SynchrotronMixin` as an example).

For special cameras the generator `_produce_frames()` can be overwritten.

Radiography

class concert.experiments.imaging.**Radiography**(*self*, *walker*, *flat_motor*, *radio_position*, *flat_position*, *camera*, *num_flats*, *num_darks*, *num_projections*, *separate_scans=True*)

Bases: *Experiment*

Radiography experiment

This records dark images (without beam) and flat images (with beam and without the sample) as well as the projections with the sample in the beam.

await **__ainit__**(*walker*, *flat_motor*, *radio_position*, *flat_position*, *camera*, *num_flats*, *num_darks*, *num_projections*, *separate_scans=True*)

Parameters

- **walker** (*concert.storage.Walker*) – Walker for storing experiment data.
- **flat_motor** – Motor for moving sample in and out of the beam. Must feature a 'position' property.
- **radio_position** – Position of *flat_motor* that the sample is positioned in the beam. Unit must be the same as flat_motor['position'].

- **flat_position** – Position of *flat_motor* that the sample is positioned out of the beam. Unit must be the same as flat_motor['position'].
- **camera** (concert.devices.cameras.base.Camera) – Camera to acquire the images.
- **num_flats** (*int*) – Number of images for flatfield correction.
- **num_darks** (*int*) – Number of images for dark correction.
- **num_projections** (*int*) – Number of projections.

num_darks

Number of images acquired for dark correction.

num_flats

Number of images acquired for flatfield correction.

num_projections

Number of projection images.

num_projections_total

Total number of projections. For most of the experiments this is the same as the number of projections.

await prepare()

Gets executed before every experiment run.

await start_sample_exposure()

This function must implement in a way that the sample is exposed by radiation, like opening a shutter or starting an X-ray tube.

await stop_sample_exposure()

This function must implement in a way that the sample is not exposed by radiation, like closing a shutter or switching off an X-ray tube.

Tomography

class concert.experiments.imaging.**Tomography**(*self*, *walker*, *flat_motor*, *tomography_motor*, *radio_position*, *flat_position*, *camera*, *num_flats=200*, *num_darks=200*, *num_projections=3000*, *angular_range=<Quantity(180, 'degree')>*, *start_angle=<Quantity(0, 'degree')>*, *separate_scans=True*)

Bases: *Radiography*

Abstract implementation of a tomography experiment.

await __ainit__(*walker*, *flat_motor*, *tomography_motor*, *radio_position*, *flat_position*, *camera*, *num_flats=200*, *num_darks=200*, *num_projections=3000*, *angular_range=<Quantity(180, 'degree')>*, *start_angle=<Quantity(0, 'degree')>*, *separate_scans=True*)

Parameters

- **walker** (concert.storage.Walker) – Walker for storing experiment data.
- **flat_motor** – Motor for moving sample in and out of the beam. Must feature a 'position' property.
- **tomography_motor** (concert.devices.motors.base.RotationMotor) – Rotation-Motor for tomography.

- **radio_position** – Position of *flat_motor* that the sample is positioned in the beam. Unit must be the same as flat_motor['position'].
- **flat_position** – Position of *flat_motor* that the sample is positioned out of the beam. Unit must be the same as flat_motor['position'].
- **camera** (concert.devices.cameras.base.Camera) – Camera to acquire the images.
- **num_flats** (*int*) – Number of images for flatfield correction.
- **num_darks** (*int*) – Number of images for dark correction.
- **num_projections** (*int*) – Number of projections.
- **angular_range** (*q.deg*) – Range for the scan of the *tomography_motor*.
- **start_angle** (*q.deg*) – Start position of *tomography_motor* for the first projection.

angular_range

Range for scanning the *tomography_motor*.

start_angle

Initial position of the *tomography_motor*.

Stepped tomography

class concert.experiments.imaging.**SteppedTomography**(*self*, *walker*, *flat_motor*, *tomography_motor*, *radio_position*, *flat_position*, *camera*, *num_flats=200*, *num_darks=200*, *num_projections=3000*, *angular_range=<Quantity(180, 'degree')>*, *start_angle=<Quantity(0, 'degree')>*, *separate_scans=True*)

Bases: *Tomography*

Stepped tomography experiment

await __ainit__(*walker*, *flat_motor*, *tomography_motor*, *radio_position*, *flat_position*, *camera*, *num_flats=200*, *num_darks=200*, *num_projections=3000*, *angular_range=<Quantity(180, 'degree')>*, *start_angle=<Quantity(0, 'degree')>*, *separate_scans=True*)

Parameters

- **walker** (concert.storage.Walker) – Walker for storing experiment data.
- **flat_motor** – Motor for moving sample in and out of the beam. Must feature a 'position' property.
- **radio_position** – Position of *flat_motor* that the sample is positioned in the beam. Unit must be the same as flat_motor['position'].
- **flat_position** – Position of *flat_motor* that the sample is positioned out of the beam. Unit must be the same as flat_motor['position'].
- **camera** (*concert.devices.camera.base.Camera*) – Camera to acquire the images.
- **num_flats** (*int*) – Number of images for flatfield correction.
- **num_darks** (*int*) – Number of images for dark correction.
- **num_projections** (*int*) – Number of projections.

- **angular_range** (*q.deg*) – Range for the scan of the *tomography_motor*.
- **start_angle** (*q.deg*) – Start position of *tomography_motor* for the first projection.

Continuous tomography

class concert.experiments.imaging.**ContinuousTomography**(*self*, *walker*, *flat_motor*, *tomography_motor*, *radio_position*, *flat_position*, *camera*, *num_flats=200*, *num_darks=200*, *num_projections=3000*, *angular_range=<Quantity(180, 'degree')>*, *start_angle=<Quantity(0, 'degree')>*, *separate_scans=True*)

Bases: *Tomography*

Continuous Tomography

This implements a tomography with a continuous rotation of the sample. The camera must record frames with a constant rate.

await __ainit__(*walker*, *flat_motor*, *tomography_motor*, *radio_position*, *flat_position*, *camera*, *num_flats=200*, *num_darks=200*, *num_projections=3000*, *angular_range=<Quantity(180, 'degree')>*, *start_angle=<Quantity(0, 'degree')>*, *separate_scans=True*)

Parameters

- **walker** (concert.storage.Walker) – Walker for storing experiment data.
- **flat_motor** – Motor for moving sample in and out of the beam. Must feature a 'position' property.
- **radio_position** – Position of *flat_motor* that the sample is positioned in the beam. Unit must be the same as flat_motor['position'].
- **flat_position** – Position of *flat_motor* that the sample is positioned out of the beam. Unit must be the same as flat_motor['position'].
- **camera** (*concert.devices.camera.base.Camera*) – Camera to acquire the images.
- **num_flats** (*int*) – Number of images for flatfield correction.
- **num_darks** (*int*) – Number of images for dark correction.
- **num_projections** (*int*) – Number of projections.
- **angular_range** (*q.deg*) – Range for the scan of the *tomography_motor*.
- **start_angle** (*q.deg*) – Start position of *tomography_motor* for the first projection.

velocity

Velocity of the *tomography_motor* in the continuous scan.

Stepped spiral tomography

class concert.experiments.imaging.**SteppedSpiralTomography**(*self*, *walker*, *flat_motor*, *tomography_motor*, *vertical_motor*, *radio_position*, *flat_position*, *camera*, *start_position_vertical*, *sample_height*, *vertical_shift_per_tomogram*, *num_flats=200*, *num_darks=200*, *num_projections=3000*, *angular_range=<Quantity(180, 'degree')>*, *start_angle=<Quantity(0, 'degree')>*, *separate_scans=True*)

Bases: SpiralMixin, *SteppedTomography*

Stepped spiral tomography

await **__ainit__**(*walker*, *flat_motor*, *tomography_motor*, *vertical_motor*, *radio_position*, *flat_position*, *camera*, *start_position_vertical*, *sample_height*, *vertical_shift_per_tomogram*, *num_flats=200*, *num_darks=200*, *num_projections=3000*, *angular_range=<Quantity(180, 'degree')>*, *start_angle=<Quantity(0, 'degree')>*, *separate_scans=True*)

Parameters

- **walker** (concert.storage.Walker) – Walker for storing experiment data.
- **flat_motor** – Motor for moving sample in and out of the beam. Must feature a 'position' property.
- **tomography_motor** (concert.devices.motors.base.RotationMotor) – Rotation-Motor for tomography scan.
- **vertical_motor** (concert.devices.motors.base.LinearMotor) – LinearMotor to translate the sample along the tomographic axis.
- **radio_position** – Position of *flat_motor* that the sample is positioned in the beam. Unit must be the same as flat_motor['position'].
- **flat_position** – Position of *flat_motor* that the sample is positioned out of the beam. Unit must be the same as flat_motor['position'].
- **camera** (concert.devices.cameras.base.Camera) – Camera to acquire the images.
- **start_position_vertical** (*q.mm*) – Start position of *vertical_motor*.
- **sample_height** (*q.mm*) – Height of the sample.
- **vertical_shift_per_tomogram** (*q.mm*) – Distance *vertical_motor* is translated during one *angular_range*.
- **num_flats** (*int*) – Number of images for flatfield correction.
- **num_darks** (*int*) – Number of images for dark correction.
- **num_projections** (*int*) – Number of projections.
- **angular_range** (*q.deg*) – Range for the scan of the *tomography_motor*.
- **start_angle** (*q.deg*) – Start position of *tomography_motor* for the first projection.

Continuous spiral tomography

class concert.experiments.imaging.**ContinuousSpiralTomography**(*self*, *walker*, *flat_motor*, *tomography_motor*, *vertical_motor*, *radio_position*, *flat_position*, *camera*, *start_position_vertical*, *sample_height*, *vertical_shift_per_tomogram*, *num_flats=200*, *num_darks=200*, *num_projections=3000*, *angular_range=<Quantity(180, 'degree')>*, *start_angle=<Quantity(0, 'degree')>*, *separate_scans=True*)

Bases: SpiralMixin, *ContinuousTomography*

Spiral Tomography

This implements a helical acquisition scheme, where the sample is translated perpendicular to the beam while the sample is rotated and the projections are recorded.

await **__ainit__**(*walker*, *flat_motor*, *tomography_motor*, *vertical_motor*, *radio_position*, *flat_position*, *camera*, *start_position_vertical*, *sample_height*, *vertical_shift_per_tomogram*, *num_flats=200*, *num_darks=200*, *num_projections=3000*, *angular_range=<Quantity(180, 'degree')>*, *start_angle=<Quantity(0, 'degree')>*, *separate_scans=True*)

Parameters

- **walker** (concert.storage.Walker) – Walker for storing experiment data.
- **flat_motor** – Motor for moving sample in and out of the beam. Must feature a 'position' property.
- **tomography_motor** (concert.devices.motors.base.ContinuousRotationMotor) – ContinuousRotationMotor for tomography scan.
- **vertical_motor** (concert.devices.motors.base.ContinuousLinearMotor) – ContinuousLinearMotor to translate the sample along the tomographic axis.
- **radio_position** – Position of *flat_motor* that the sample is positioned in the beam. Unit must be the same as flat_motor['position'].
- **flat_position** – Position of *flat_motor* that the sample is positioned out of the beam. Unit must be the same as flat_motor['position'].
- **camera** (concert.devices.cameras.base.Camera) – Camera to acquire the images.
- **start_position_vertical** (*q.mm*) – Start position of *vertical_motor*.
- **sample_height** (*q.mm*) – Height of the sample.
- **vertical_shift_per_tomogram** (*q.mm*) – Distance *vertical_motor* is translated during one *angular_range*.
- **num_flats** (*int*) – Number of images for flatfield correction.
- **num_darks** (*int*) – Number of images for dark correction.
- **num_projections** (*int*) – Number of projections.
- **angular_range** (*q.deg*) – Range for the scan of the *tomography_motor*.

- **start_angle** (*q.deg*) – Start position of *tomography_motor* for the first projection.

2.2.12 Extensions

Concert integrates third-party software in the `ext` package. Because the dependencies of these modules are not listed as Concert dependencies, you have to make sure, that the appropriate libraries and modules are installed.

UFO Processing

class concert.ext.ufo.**FlatCorrect**(*dark*, *flat*, *absorptivity=True*, *fix_nan_and_inf=True*, *copy_inputs=False*)

Flat-field correction.

class concert.ext.ufo.**GeneralBackproject**(*args*, *resources=None*, *gpu_index=0*, *do_normalization=False*, *region=None*, *copy_inputs=False*)

One backprojector worker.

args

`GeneralBackprojectArgs` instance with arguments for reconstruction

resources

Ufo.Resources instance

gpu_index

use GPU with this index

do_normalization

do flat correction of not

region

User defined region which can override the one stored in *args*

copy_inputs

if True copy images before they are inserted into UFO

exception concert.ext.ufo.**GeneralBackprojectArgsError**

exception concert.ext.ufo.**GeneralBackprojectError**

class concert.ext.ufo.**GeneralBackprojectManager**(*self*, *args*, *average_normalization=True*, *regions=None*, *copy_inputs=False*)

Manage 3D recnstruction by back projection. The manager stores darks, flats and projections and spreads them to back projection workers in as many batches as needed in order not to overflow GPU memory.

args

`GeneralBackprojectArgs` instance with arguments for reconstruction

average_normalization

if False, use only one dark and flat image from their streams, otherwise average all of them

regions

User defined regions (batches) for reconstruction, if None, batches are determined automatically

copy_inputs

if True copy images before they are inserted into UFO

await backproject(*producer*)

Backproject projections from *producer*.

find_parameters(*parameters*, *projections=None*, *metrics=('sag',)*, *regions=None*, *iterations=1*, *fwhm=0*, *minimize=(True,)*, *z=None*, *method='powell'*, *method_options=None*, *guesses=None*, *bounds=None*, *store=True*)

Find reconstruction parameters. *parameters* (see `GeneralBackprojectArgs.z_parameters`) are the names of the parameters which should be found, *projections* are the input data and if not specified, the ones from last reconstruction are used. *z* specifies the height in which the parameter is looked for. If *store* is True, the found parameter values are stored in the reconstruction arguments. Optimization is done either brute-force if *regions* are not specified or one of the scipy minimization methods is used, see below.

If *regions* are specified, they are reconstructed for the corresponding parameters and a metric from *metrics* list is applied. Thus, first parameter in *parameters* is reconstructed within the first region in *regions* and the first metric (see `GeneralBackprojectArgs.slice_metrics`) in *metrics* is applied and so on. If *metrics* is of length 1 then it is applied to all parameters. *minimize* is a tuple specifying whether each parameter in the list should be minimized (True) or maximized (False). After every parameter is processed, the parameter optimization result is stored and the next parameter is optimized in such a way, that the result of the optimization of the previous parameter already takes place. *iterations* specifies how many times are all the parameters reconstructed. *fwhm* specifies the full width half maximum of the gaussian window used to filter out the low frequencies in the metric, which is useful when the region for a metric is large. If the *fwhm* is specified, the region must be at least 4 * fwhm large. If *fwhm* is 0 no filtering is done.

If *regions* is not specified, `scipy.minimize()` is used to find the parameter, where the optimization method is given by the *method* parameter, *method_options* are passed as *options* to the minimize function and *guesses* are initial guesses in the order of the *parameters* list. If *bounds* are given, they represent the domains where to look for parameters, they are (min, max) tuples, also in the order of the *parameters* list. See documentation of `scipy.minimize()` for the list of minimization methods which support bounds specification. In this approach only the first in *metrics* is taken into account because the optimization happens on all parameters simultaneously, the same holds for *minimize*.

property num_processed_projections

Number of projections sent to backprojectors.

property num_received_projections

Number of received projections.

reset()

Reset state, clearing all pre-processing steps but keep projections and slices intact.

await update_darks(*producer*)

Get new darks from *producer*. Immediately start the reconstruction so that averaging starts.

await update_flats(*producer*)

Get new flats from *producer*. Immediately start the reconstruction so that averaging starts.

exception concert.ext.ufo.**GeneralBackprojectManagerError**

class concert.ext.ufo.**InjectProcess**(*graph*, *get_output=False*, *output_dims=2*, *scheduler=None*, *copy_inputs=False*)

Process to inject NumPy data into a UFO processing graph.

InjectProcess can also be used as a context manager, in which case it will call *start()* on entering the manager and *wait()* on exiting it.

graph must either be a Ufo.TaskGraph or a Ufo.TaskNode object. If it is a graph the input tasks will be connected to the roots, otherwise a new graph will be created. *scheduler* is one of the ufo schedulers, e.g. Ufo.Scheduler or Ufo.FixedScheduler.

await insert(*array*, *node=None*, *index=0*)

Insert *array* into the *node*'s *index* input.

Note: *array* must be a NumPy compatible array.

await result(*leave_index=None*)

Get result from *leave_index* if not None, all leaves if None. Returns a list of results in case *leave_index* is None or one result for the specified leave_index.

start(*arch=None*, *gpu=None*)

Run the processing in a new thread.

Use `push()` to insert data into the processing chaing and *wait()* to wait until processing has finished.

stop()

Stop input tasks.

wait()

Wait until processing has finished.

exception concert.ext.ufo.**InjectProcessError**

Errors related with *InjectProcess*.

class concert.ext.ufo.**PluginManager**

Plugin manager that initializes new tasks.

get_task(*name*, ***kwargs*)

Create a new task from plugin *name* and initialize with *kwargs*.

Viewers

Opening images in external programs.

class concert.ext.viewers.**ImageViewerBase**(*self*, *limits: str = 'stream'*, *downsampling: int = 1*, *title: str = ''*, *show_refresh_rate: bool = False*, *force: bool = False*)

Backend-free base class for displaying images.

limits

minimum and maximum gray value (black and white points). Can be a tuple (min, max), 'auto' or 'stream'. When 'auto', limits are adjusted for every shown image, when 'stream', limits are adjusted on every __call__.

downsampling

Display every n-th pixel, which can speed up the viewer

title

Image title

show_refresh_rate

Whether or not to show refresh rate text directly embedded into the displayed image

class concert.ext.viewers.**PyQtGraphViewer**(*self*, *limits: str = 'stream'*, *downsampling: int = 1*, *title: str = ''*, *show_refresh_rate: bool = False*, *force: bool = False*)

Dynamic image viewer using PyQtGraph.

class concert.ext.viewers.**PyplotImageViewer**(*self*, *imshow_kwargs: dict = None*, *fast: bool = True*, *limits: str = 'stream'*, *downsampling: int = 1*, *title: str = ''*, *show_refresh_rate: bool = False*, *force: bool = False*)

Dynamic image viewer using matplotlib.

imshow_kwargs

matplotlib's imshow keyword arguments

fast

Whether to use the fast version without colorbar

reset()

Reset the viewer's state.

class concert.ext.viewers.**PyplotViewer**(*self*, *style: str = 'o'*, *plot_kwargs: dict = None*, *autoscale: bool = True*, *title: str = ''*, *force: bool = False*)

Dynamic plot viewer using matplotlib.

style

One of matplotlib's linestyle format strings

plt_kwargs

Keyword arguments accepted by matplotlib's plot()

autoscale

If True, the axes limits will be expanded as needed by the new data, otherwise the user needs to rescale the axes

title

Plot title

reset()

Clear the plotted data.

class concert.ext.viewers.**ViewerBase**(*self*, *force: bool = False*)

A base class for data viewer which sends commands to a backend-specific updater which runs in a separate process.

pause()

Pause, no images are dispayed but image commands work.

resume()

Resume the viewer.

await show(*item*, *force=False*)

Push *item* to the queue for display in a separate proces. If *force* is True make sure the item is displayed, otherwise it may be skipped if there is something in the queue waiting to be shown.

exception concert.ext.viewers.**ViewerError**

Viewer errors.

concert.ext.viewers.**imagej**(*image*, *path='imagej'*)

Open *image* in ImageJ found by *path*.

CHAPTER

THREE

ADDITIONAL NOTES

3.1 Changelog

Here you can see the full list of changes between each Concert release.

3.1.1 Version 0.31

Released on May 11th 2022.

Features

- it is possible to use `await` statements in concert sessions outside of `async def` functions
- `AsyncObject` allows us to have async constructors `async def __ainit__`, which is very useful when we need to use async code in constructors, e.g. setting a parameter on a device
- Experiments were greatly extended by classes which can be customized and run at beam lines without many more changes
- Monochromators were enhanced
- Experiment director was introduced to take care of high-throughput measurements
- Experiments can write metadata to json
- Parameter limits getters and setters are coroutine functions
- Add ElementSelector device for having discrete choices
- `ascan` allows one to include the `stop` value in the scan
- walkers have `create_writer` method for convenience
- walkers can be used in `async with` statements for safe access from multiple concurrent coroutines
- `concert.coroutines.sinks.Accumulate` allows non-re-setting behavior between runs
- `concert.processes.common.align_rotation_axis` allows multiple frame consumers
- `concert.devices.cameras.uca.Camera` gets state from the libuca object

API breaks

- `Parameterizable` inherits from `AsyncObject` which needs an `__ainit__`

constructor, so every sub-class must change its `__init__` to `async def __ainit__`. - `run_in_loop_thread_blocking` is gone in favour of using one event loop and

`__ainit__` constructors

Notes

- we replaced Travis by CircleCI

Pull Requests

#485 from ufo-kit/circleci-project-setup #483 from ufo-kit/new-metadata #464 from ufo-kit/monochromator-enhancement #459 from ufo-kit/experiment-director #409 from ufo-kit/fix-uca-state #467 from ufo-kit/grating-inteferometry-experiment #478 from ufo-kit/hercules #473 from ufo-kit/walkers #471 from ufo-kit/ascan-include-last #470 from ufo-kit/elementselector #466 from ufo-kit/fix-tango-loop #458 from ufo-kit/new-experiments #461 from ufo-kit/async-limits

3.1.2 Version 0.30

- Simplify scans, no `Region` and no need of using `resolve` on the result

Features

- concurrency transition to asyncio, from user perspective the usage is more or less the same, e.g. `motor.move(1 * q.mm)` starts relative motion in the background, `motor.position = 1 * q.mm` sets the position in a blocking way.
- ctrl-k cancels all background commands and calls `emergency_stop` on all devices in the session
- Online reconstruction is capable of doing the flat correction with image averaging
- `concert.experiments.addons.ImageWriter` blocks for every acquisition

3.1.3 Viewers

- Subclass Parameterizeable, properties turned into parameters
- Added Simple matplotlib backend without colorbar but faster
- Added PyQtGraphViewer
- Added downsampling parameter
- Added show_refresh_rate parameter
- Parameter limits accepts “stream”, i.e. the limits are updated for the first image of __call__ and then kept the same, “auto” updates limits for every image and a tuple fixes them to min and max.

API breaks

- Removed `queue`, `sinograms`, `backproject`, `PickSlice`, `process` coroutines from `concert.coroutines.filters`
- Removed `Backproject`, `FlatCorrectedBackproject`, `FlatCorrect` (was there twice), `center_rotation_axis`, `compute_rotation_axis` from `concert.ext.ufo`
- Removed `concert.networking.aerotech` and `concert.networking.wago`
- Removed `concert.devices.motors.crio`
- Removed `concert.helpers.hasattr_raise_exceptions`
- Removed `Future.cancel_operation`, cancellation is implemented via `asyncio.CancelledError`
- Removed `concert.helpers.Region`
- **Removed devices:**
  - `concert.devices.motors.aerotech`
  - `concert.devices.io.aerotech`
  - `concert.devices.lightsources.fiberlite`
  - `concert.devices.lightsources.led.`
  - `concert.devices.photodiodes.edmundoptics`
  - `concert.devices.scales.adventurer`
  - `concert.devices.monochromators.newport.py`
- **Removed processes:**
  - `concert.processes.common.find_beam`
  - `concert.processes.common.drift_to_beam`
  - `concert.processes.common.center_to_beam`
  - `concert.processes.common.scan_param_feedback`
- Removed callbacks from `concert.processes.common.scan`
- Removed `concert.experiments.base.Acquisition.connect` for simplicity, i.e. the acquisition and consumption of data happens always at the same time, for special behavior, special classes/functions should be used
- `Device.abort` replaced by `Device.emergency_stop`
- many functions which were not asynchronous before are now, e.g. `Camera.start_recording`

3.1.4 Version 0.11

Released on April 14th 2021, which is the last release for Python 2.

Features

- 345c2e7 Add stack-based name resolution
- f8e9ec5 experiments: add per-run log handler
- 41bfa5e Add State to Experiment
- 253d4d5 Add external limits to parameter
- 40debd7 Added target value to parameter
- 6eb07fd Added LensChanger
- dc0b395 Add readers
- ec90dcd ufo: Add FlatCorrect class
- e7326bd Add dummy ImagingExperiment
- f90379d Add fwhm to find_parameters
- a8c705b imageprocessing: add filter_low_frequencies
- e181587 Add a simple online reconstruction addon
- 51d6ec2 Add ROI selection support to ImagingFileExperiment
- 2ba2fe1 add fraction and percent to quantities
- e59697b Add progress bar
- 43c4e98 Add ImagingFileExperiment
- e823b52 base and dummy attenuatorbox added
- ece80dd Add flat field correction task
- 1ec03be addons: Enable async I/O by ImageWriter
- 6535790 Add *docs* command to Concert program
- 0f91277 Add Flask-like extension system
- 6c02a84 abort on ctrl-c
- align axis: auto-compute metric_eps by default (better rotation axis alignment)

Pull Requests

- #374 from ufo-kit/calculate_motor_velocity
- #389 from ufo-kit/add-flatcorrect
- #396 from ufo-kit/fix-388
- #419 from ufo-kit/add-percent-quantity
- #436 from ufo-kit/align-rotation-axis
- #450 from ufo-kit/experiment-state
- #447 from ufo-kit/device-names
- #445 from ufo-kit/experiment-logs
- #437 from ufo-kit/add-lens-selector
- #442 from ufo-kit/always-execute-finish-in-experiment

- #443 from ufo-kit/add_enable_disable_to_motors
- #434 from ufo-kit/add-state-to-experiment
- #435 from ufo-kit/add-external-softlimits
- #439 from ufo-kit/add-state-to-monochromator
- #339 from ufo-kit/ufo-multi-gpus
- #432 from ufo-kit/fix-requirements-python2.7
- #433 from ufo-kit/fix-typo-in-experiment
- #430 from ufo-kit/cleanup-writers
- #429 from ufo-kit/add-readers
- #425 from ufo-kit/add_parameters_to_experiments
- #428 from ufo-kit/dependabot/pip/pyxdg-0.26
- #427 from ufo-kit/fix-fsm-transitioning-error
- #417 from ufo-kit/add-progressbar
- #412 from ufo-kit/enable-writer-offset
- #411 from ufo-kit/multipage-file-camera
- #410 from ufo-kit/faster-dummy-camera
- #408 from ufo-kit/write_multi
- #405 from ufo-kit/add-attenuator
- #401 from ufo-kit/fix-400
- #399 from ufo-kit/Flake8
- #397 from ufo-kit/fix-367
- #395 from ufo-kit/fix-385
- #394 from ufo-kit/docs
- #393 from ufo-kit/docs
- #391 from ufo-kit/forbid-addon-reattachment
- #387 from ufo-kit/extend-queue
- #379 from ufo-kit/fix-flatcorrectedbackproject
- #383 from ufo-kit/fix-dimax-start-recording
- #381 from ufo-kit/fix-ufo-segfault
- #370 from ufo-kit/generalize-acquisitions
- #378 from ufo-kit/print-noasync-traceback
- #377 from ufo-kit/accumulate-addon
- #376 from ufo-kit/numpy-accumulate
- #375 from ufo-kit/fix-DirectoryWalker-relative-paths
- #369 from ufo-kit/extensions

Fixes

- #400: Fix properties mixup of uca-cameras
- #392 and support for current Pint versions'
- #388: Fixed units checking of Numeric
- #385: Optional output of resolve without units
- #371: override grab only in uca cameras
- #301: don't print traceback for DISABLE_ASYNC

3.1.5 Version 0.10

Released on February 9th 2015.

Improvements

- Uca cameras support optional parameters.
- We added convenience functions for acquiring certain image types like dark fields, etc.
- We can determine tomographic rotation axis based on the convenince functions mentioned above.
- Hard-limit state is allowed as target in motor's *home* method.
- Added a decorator for measuring function execution time.
- Added XRayTube device.
- Added Gripper device.
- Added asynchronous grab *Camera.grab_async*.
- Added SampleChanger device.
- Parameter setters are abortable. Thanks to that we added the *abort* function to stop actions on devices. It can be used also per-device.
- Simplified *concert.base*, we don't use metaclasses anymore.
- Added *normalize* for intensity normalization.
- Added *Camera.convert* for converting images before they are returned by the camera's *grab* method (useful for flipping, etc.).
- Added a generic *process* coroutine which takes a callable and applies it to the coming data.
- We check soft limits for correct unit.
- Added EDF reading support via fabio.
- Added experiment *Addon* which operate on the data produced by an experiment (e.g. image viewing, online reconstruction, etc.).
- Added n-dimensional scans.
- Added ssh+tmux support via *concert-server* and *concert-connect*.
- Added session *export* command.
- Added session loading via *–filename*.

- Walker can write data stored in lists, not only in a coroutine way.

API breaks

- Renamed *fetch* command to *import*.
- Use positive config names (*ENABLE_* instead of *DISABLE_*).

Fixes

- Various beam time fixes from #345.
- IPython version check in #332.
- #300, #301, #306, #308, #310, #331, #353.

3.1.6 Version 0.9

Released on August 15th 2014.

Improvements

- The state machine mechanism is not special anymore but directly inherits from *Parameter*.
- Added walker mechanism to write sequence data in hierarchical structures such as directories or HDF5 files.
- The long-standing gevent integration with IPython is finished at least for IPython >= 2.0.
- Added *@expects* decorator to annotate what a function can receive.
- Added *async.resolve()* to get result of future lists.
- Added *accumulate* sink and *timer* coroutines.
- Added *Timestamp* class for PCO cameras that decodes the BCD timestamp embedded in a frame.
- Added optional *wait_on* to getter and setter of a *ParameterValue*.
- We now raise an exception in if a uca frame is not available.
- Experiments have now hooks for preparation and cleanup tasks.
- Added basic control loop classes.
- Add binary signal device class.

API breaks

- *scan* yields futures instead of returning a list
- Moved specific pco cameras to *concert.devices.cameras.pco*.
- Moved *write_images* to *concert.storage*
- Removed *base.MultiContext* and *base.Process*

Fixes

- #198, #254, #271, #277, #280, #286, #293
- The pint dependency had to be raised to 0.5.2 in order to compute sums of quantities.

3.1.7 Version 0.8

Released on April 16th 2014.

Improvements

- `concert log` can now `--follow` the current operation.
- Soft limits and parameters can be locked both temporarily and permanently.
- Added new `@quantity` decorator for simple cases.
- The `concert` binary can now be started without a session.
- Added cross-correlation tomographic axis finding.
- Added frame consumer to align_rotation_axis.
- Simplify file camera and allow resetting it
- Added ports property to the base IO device.
- Added Photodiode base device class.
- Added Fiber-Lite halogen lightsource.
- Added LEDs connected within the wago.
- Added stream coroutine to cameras.
- Added EdmundOptics photodiode.
- Added PCO.4000 camera.
- Added Wago input/output device.

API breaks

- Raise CameraError instead of ValueError
- Change Pco's freerun to stream

Fixes

- Fix FileCamera pixel units in grab
- Import GLib.GError correctly
- Make recording context exception-safe
- Fix quantity problem with recent Pint versions
- #200, #203, #206, #209, #228, #230, #245

3.1.8 Version 0.7

Released on February 17th 2014.

Improvements

- Added beam finding and centering
- `threaded` decorator uses daemonic threads
- Added `downsize`, `queue`, `stall`, `PickSlice` to coroutine filters
- Added reconstruction of the whole volume using UFO Framework
- Documentation was restructured significantly (split to usage/API)
- Added tomography helper functions
- Crio motor support continuous rotation
- `PyplotViewer` can be configured for faster drawing capabilities using `blit`
- Added dummy `Scales`
- Tests cover all devices (at least try to instantiate them)
- Added pixel units, `q.pixel` (shorthand `q.px`)
- Changed prompt color to terminal default
- Added `Positioner` device
- Added `Detector` device

API Breaks

- Finite state machine was reworked significantly
- Motors were cleaned from mixins and hard-limit was incorporated into them
- recording() context was added to cameras
- `backprojector` coroutine filter was significantly simplified
- `average_images` arguments changed
- Experiments were completely restructured based on usage of `Acquisition`
- `PyplotViewer` plotting signature changed
- Remove leftover beam line specific shutters
- Many getters/setters were replaced by properties, especially in the `concert.ext.viewers` module
- Appropriate `get_` `set_` functions were replaced by non-prefixed ones

Fixes

- #118, #128, #132, #133, #139, #148, #149, #150, #157, #159, #165, #169, #173, #174, #175, #176, #178, #179, #181, #184, #189, #192

3.1.9 Version 0.6

Released on December 10th 2013.

Improvements

- Concert now comes with an experimental gevent backend that will eventually replace the thread pool executor based asynchronous infrastructure.
- Each device can now have an explicit `State` object and `@transition` applied to function which will change the state depending on the successful outcome of the decorated function.
- 1D data plotting is implemented as `PyplotCurveViewer`.
- The `concert` binary now knows the `cp` command to make a copy of a session. The `start` command can receive a log level and with the `--non-interactive` option run a session as a script.
- Devices and parameters can store their current parameter values with `stash` and restore them later with `restore`.
- Changed the IPython prompt.
- Added the NewPort 74000 Monochromator.
- Provide a `require` function that will scream when the required Concert version is not installed.

API breaks

- `Motor` is renamed to `LinearMotor` for all devices.
- `Parameter` objects are now declared at class-level instead of at run-time within the class constructor.
- `concert.storage.create_folder` renamed to `concert.storage.create_directory`
- `concert.ext.viewers.PyplotViewer` substituted by 1D and 2D viewers `concert.ext.viewers.PyplotCurveViewer` and `concert.ext.viewers.PyplotImageViewer`
- To wait on a Future you have to call `.join` instead of `.wait`.
- Coroutine functions and decorators moved to `concert.coroutines[.base]`, asynchronous functions and decorators moved to `concert.async`.
- Removed `is_async`
- Configuration moved to `concert.config`
- Method names of `concert.ext.ufo.InjectProcess` changed.

Fixes

- #168, #166, #152, #147, #158, #150, #157, #95, #138
- Many more concerning the camera implementation.

3.1.10 Version 0.5

Released on October 31st 2013.

Improvements

- Python 3 is supported and can be tested with tox.
- Most imports are delayed in the concert binary to reduce startup time.
- We do not depend on Logbook anymore but use Python's logging module.
- Experiments can now be modelled with the `concert.experiments` module.
- `concert.ext.viewers.PyplotViewer` can be used to show 2D image data.
- Spyder command plugin is now available. That means if you have Spyder installed you can control Concert from an IDE instead of from IPython.
- Tests were restructured for easier access.

API breaks

- `concert.connections` package moved to `concert.networking` module
- Renamed `concert.helpers.multicast` to `broadcast` to reflect its true purpose.
- Session helpers such as `dstate` and `ddoc` have been moved to `concert.session.utils`.
- Frames grabbed with the libuca devices will return a copy instead of the same buffer.

Fixes:

- #106, #113 and many more which did not deserve an issue number.

3.1.11 Version 0.4

Released on October 7th 2013.

Improvements

- Tests and rotation axis alignment is faster now.
- Soft limits were added to the parameter (accessible with `.lower` and `.upper`)
- Cleaner inet connection implemention.
- Base pumps and scales were added.
- Concert no longer depends on testfixtures for running tests.
- Started work on flexible data processing schemes for light computation based on a coroutine approach.

- Integrated tifffile.py in case libtiff is not available.
- `concert mv` renames sessions.
- `@threaded` decorator can be used to run a function in its own thread.
- `Scanner` parameters can now be set in the constructor.
- Parameters can now be locked independently of the parent device. However, if done so, no one else can lock the device.
- Add `code_of` function to show the source of a function.
- Introduced coroutine based data processing facility.

API breaks

- Renamed `to_steps` to `to_device` and do not drop units
- `camera.grab` returns *None* if no data is available
- `uca.Camera` exposes the wrapped GObject camera as an attribute called `uca` instead of `camera`.
- `minimum`, `maximum` and `intervals` are now longer implemented as `Parameter` objects of `Scanner` but simple attributes.
- `asynchronous` module content has been moved to `helpers`
- Removed `Scanner` class in favor of `scan` function.

Fixes:

- Integration with all IPython releases works again.
- runtests.py returns 0 on success.
- #19, #55, #71, #78, #79

3.1.12 Version 0.3

Released on August 19th 2013.

Note: This release breaks Python 2.6 compatibility!

- `Calibration` classes moved to `concert.devices.calibration`
- Remove `concert.processes.focus` and reorganize `concert.optimization` package, the focusing can be implemented by Maximizer with a proper feedback.
- Add `--repo` parameter to the `fetch` command. With this flag, session files version controlled with Git can be imported.
- Use pint instead of quantities. pint is faster for smaller Numpy arrays, stricter and does not depend on Numpy.
- Things can now run serialized if `concert.asynchronous.DISABLE` is set to `True`.
- Restructured tests into separate directories.
- Fix PDF generation of the docs.
- Fix problem with IPython version >= 0.10.

3.1.13 Version 0.2

Released on July 14th 2013.

- Move third-party code to `concert.ext`. For example `get_tomo_scan_result` must be imported from `concert.ext.nexus`.
- Adds `concert fetch` to pull session files from remote locations.
- Code cleanup

3.1.14 Version 0.1.1

Bug fix release, released on May 25th 2013

- Fixes Python 3 support.
- Monochromator fix.

3.1.15 Version 0.1

First public release.

PYTHON MODULE INDEX

C

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

X