Scan Scripts Documentation

Release 0.4.0

Argonne National Laboratory

Sep 25, 2018

Contents

This GitHub repository provides a framework for running experiment scripts at various beamline at the Advanced Photon Source.

CHAPTER 1

Content

1.1 Install

This section covers the basics of how to download and install ScanScripts.We recommend you to install the Anaconda Python distribution.

Contents:

1.1.1 Installing from source

Clone the ScanScripts from GitHub repository:

```
git clone https://github.com/tomography/scanscripts.git project
```

then:

```
cd project
python setup.py install
```

1.1.2 Beamline Configuration

The scanscripts library looks for a file in the top director (eg `~/TXM/scanscripts`) called `beamline_config.conf`. This file should contain configuration details for how the beamline is setup. This allows easy configuration changes without having to modify library code. See the documentation for each beamline for more details on which options are supported:

- *Sector 32-ID Configuration*

1.2 Sector 32-ID Scripts

1.2.1 General Features

All the scan scripts below can be executed in one of three ways.

1. Through the `tomography.sh` graphical user interface (GUI)
2. From the command line interface (CLI)
3. Directly from a python interpreter

The mechanisms behind the GUI and command-line interfaces are identical. Every argument in the GUI parameter panel is also present as a long argument on the command-line:

```
$ energy-scan --Energy_End 8.5 --Energy_Start 8.3 --ExposureTime 1.5 --SampleXOut 0.1
```

The programatic python versions start with `run_`. They often have slightly differet parameters to the GUI/CLI implementation, allowing for more precise control.

```
>>> import aps_32id
>>> import numpy as np
>>> aps_32id.run_energy(energies=np.linspace(8.3, 8.5, num=101))
```

Logging

These scripts (except for `move_energy`) uses the standard library `logging` module to save logs with file names matching the HDF5 data files. The default level is `logging.INFO`, but this can be changed by using the `Log_Level` variable:

```
$ energy-scan --Log_Level 10
```

or the `log_level` parameter:

```
>>> import numpy as np
>>> import logging
>>> import aps_32id
>>> aps_32id.run_energy_scan(energies=np.linspace(8.3, 8.5, 100), log_level=logging.
↪DEBUG)
```

The log levels are the same as those defined in the logging module. They get set to the root logger, so logging.UNSET results in all messages being sent through. The special value -1 causes no changes to the logging configuration.

Table 1: Logging levels for the `Log_Level` variable

Level	Value
(no change)	-1
logging.UNSET	0
logging.DEBUG	10
logging.INFO	20
logging.WARNING	30
logging.ERROR	40
logging.CRITICAL	50

1.2.2 Move Energy

GUI:	`run/move_energy.py`
Command-line:	`$ move-energy`
Python:	`>>> aps_32id.move_energy()`

The `move_energy` script provides a way to change the energy of the beamline. If the parameter `constant_mag` is truthy, the detector will move to maintain a constant level of magnification. The equivalent function `move_energy()` can be used programatically.

1.2.3 Energy Scan

GUI:	`run/energy_scan.py`
Command-line:	`$ energy-scan`
Python:	`>>> aps_32id.run_energy_scan()`

The `energy_scan` script collects 2D frames over a range of energies, as well as the corresponding flat-field and dark-field images. The equivalent function `run_energy_scan()` lets this script be called programatically. The variable dictionary contains parameters for `Energy_Start`, `Energy_End` and `Energy_Step`. If more control is needed (eg, non-evenly spaced energies), then the function should be used with the `energies` argument. The helper function `energy_range()` allows easy construction of a unique list of energies.

```python
from aps_32id import run_energy_scan
from scanlib import energy_range
import numpy as np

# Create a list of energies from energy ranges
energies = energy_range(
    # (start, end, step)
    (8250, 8290, 10),
    (8290, 8300, 2),
    (8300, 8380, 1),
    (8380, 8500, 10),
)

# Describe position for sample and flat-field frames
# (x, y, z, θ°)
out_pos = (0.2, None, None, 0)
sample_pos = (0, None, None, 0)

# Execute the scan
run_energy_scan(energies=energies, out_pos=out_pos, sample_pos=sample_pos)
```

1.2.4 Tomography Step Scan

GUI:	`run/tomo_step_scan.py`
Command-line:	`$ tomo-step-scan`
Python:	`>>> aps_32id.run_tomo_step_scan()`

The `tomo_step_scan` script collects a tomogram as well as flat-field and dark-field images. The variable dictionary entries `SampleStart_Rot`, `SampleEnd_Rot`, `Projections` control which angles get run. If more control is needed, the `run_tomo_step_scan()` function with the `angles` parameter can be used. It is not a requirement that the angles be equally spaced.

```python
import numpy as np

from aps_32id import run_tomo_step_scan

# Create the list of angles to scan
angles = np.linspace(0, 180, 361)

# Describe positions for sample and white-field position
# (x, y, z, θ°)
out_pos = (0.2, None, None, 0)
sample_pos = (0, None, None, 0)

# Execute the scan
run_tomo_step_scan(angles=angles, sample_pos=sample_pos, out_pos=out_pos)
```

1.2.5 Tomography Fly Scan

GUI:	`run/tomo_fly_scan.py`
Command-line:	`$ tomo-fly-scan`
Python:	`>>> aps_32id.run_tomo_fly_scan()`

The `tomo_fly_scan` script is similar to `tomo_step_scan` except it does not come to a complete stop when collecting projection. The timing must be uniform, so only equally spaced angles are allowed, even in the python function form.

1.2.6 Mosaic Tomography Fly Scan

Warning: This function has not yet replaced the "old style" script at the beamline.

The `mosaic_tomo_fly_scan` script and `mosaic_tomo_fly_scan()` are similar to `tomo_step_scan` except multiple fields of view are collected.

1.2.7 Roll-Your-Own Scripts

Those with a sense of adventure can write their own scripts for Sector 32. It's highly recommended to become familiar with the *Sector 32-ID TXM* and *Examples* pages.

1.3 Sector 32-ID TXM

Note: This code is under active development and may change at any time. If you encounter issues, or documentation bugs, please submit an issue.

This page describes the features of the `aps_32id.txm.NanoTXM` class, and a few supporting classes. The `NanoTXM` class is the primary interface for controlling the Transmission X-ray Microscope (TXM) at beamline 32-ID-C. There is also a complimentary `aps_32id.txm.MicroTXM`.

A **core design goal** is to keep as much of the complexity in the `NanoTXM` class, which leaves the scripts to handle high-level details. It also allows for better unit and integration testing. When creating new scripts, it is recommended to **put all interactions to process variables (PVs) in methods of the `NanoTXM` class**. This may seem silly for single PV situations, but will make the script more readable. A hypothetical example:

```
# Not readable at all: what does that address even mean??
PV('32idcTXM:SG_RdCntr:reset.PROC').put(1, wait=True)

# Better, but still not great: what does 1 mean?
txm.Reset_Theta = 1

# Best, even though this method definition would only have one line
txm.reset_theta()
```

1.3.1 Sector 32-ID Configuration

The following configuration options can be set in the `beamline_config.conf` file under the `[32-ID-C]` heading:

has_permit (yes|no) If `has_permit` is "no", then the script will not attempt to change the X-ray source, monochromator, shutters, etc. This allows testing of scripts while the B-hutch is operating without risking interferance.

stage (NanoTXM|MicroCT) Controls which stage/optics/shutters to use for manipulating the sample. `MicroCT` uses the front stage and `NanoTXM` uses the rear stage.

zone_plate_drn (int) The width, in nm, of the outermost zone of the zone-plate of the zone-plate (Δr_n).

zone_plate_diameter (int) The total diameter, in µm, of the zone-plate.

zone_plate_drift_x (float) Adjusts the zoneplate x position by this amount for every unit change of zoneplate z. When properly set, this will keep the sample centered when changing energy.

zone_plate_drift_y (float) Adjusts the zoneplate y position by this amount for every unit change of zoneplate z. When properly set, this will keep the sample centered when changing energy.

```
[32-ID-C]
has_permit = True
# Either NanoTXM or MicroCT
stage = NanoTXM
# Correct for zoneplate drift when changing energies
zone_plate_drn = 50
zone_plate_diameter = 180
zone_plate_drift_x = 0.
zone_plate_drift_y = 0.
```

Internally, these options are parsed in `aps_32id.txm.txm_config()` using the standard library's configparser package. To make scripts easier to read, it is best to read the configuration only inside methods of `NanoTXM` (or subclasses). The configuration values can be read in the following manner:

```
cfg = txm_config()['32-ID-C']
zp_drn = cfg.getfloat('zone_plate_drn')
has_permit = cfg.getboolean('has_permit')
```

1.3.2 Stopping Scans Gracefully

When a scan script ends, we want the **instrument to return to a usable configuration** even if an exception occurred. Using the `run_scan()` context manager, this becomes easy. At the start of the context, this manager saves certain configuration details about instrument; when exiting the context for any reason the configuration is restored, the CCD is set to "continuous mode", and any extra logging is stopped:

```
import logging
import aps_32id

txm = aps_32id.NanoTXM()

with txm.run_scan():
    # Setup the microscope as desired
    txm.setup_hdf_writer()
    txm.start_logging(logging.INFO)
    txm.setup_detector()
    # Now do experiment stuff
```

1.3.3 Process Variables

Process variables (PVs), though the `pyepics` package are the way python controls the actuators and sensors of the instrument. There are **two ways to interact with process variables**:

1. The `pv_put()` method on a `NanoTXM` object.
2. A `TxmPV` descriptor on the `NanoTXM` class (or subclass).

The second option handles more of the underlying complexity, but understanding it requires a good grasp of the first option. The `NanoTXM.pv_put()` method is a wrapper around `pyepics.PV.put()`, and accepts similar arguments:

```
# These two sets of statements have the same effect

# Using the epics PV class
epics.PV('my_great_pv').put(1, wait=True)

# Using the TXM method
my_txm = TXM()
my_txm.pv_put('my_great_pv', 1, wait=True)
```

Behind the scenes, there is some extra magic so *the txm can coordinate PVs that work together*.

Manually supplying the PV name and options each time is cumbersome, so the `TxmPV` descriptor can be used to **define PVs at import time**. Set instances of the `TxmPV` class as attributes on a `NanoTXM` subclass, then assign and retrieve values directly from the attribute:

```
from aps_32id import NanoTXM
from scanlib import TxmPV

class ExampleTXM(NanoTXM):
    # Define a PV during import time
    my_awesome_pv = TxmPV('cryptic:pv:string', dtype=float, wait=True)
    # More PV definitions go here

# Now we can use the PV attribute of the txm class
my_txm = ExampleTXM()
```

(continues on next page)

(continued from previous page)

```
# Retrieve the current value
# Equivalent to ``float(epics.PV('cryptic:pv:string').get())``
curr_value = my_txm.my_awesome_pv
# Set the value
# Equivalent of epics.PV('cryptic:pv:string').put(2.718, wait=True)
my_txm.my_awesome_pv = 2.718
```

The advantage here is that boilerplate, such as type-casting and blocking, can be defined once then forgotten. This approach also lets you define PVs that should not be changed when the B-hutch is being operated, by passing `permit_required=True` to the TxmPV constructor. *More on this below*.

1.3.4 Waiting on Process Variables

Sometimes it is necessary to set one PV then wait on a different PV to confirm the new value. The `tomo.32id.txm.TXM.wait_pv()` method will poll a specified PV until it reaches its target value. It accepts the *attribute name* of a PV, not the actual PV name itself. It may be necessary to use the `wait=False` argument on the first PV to avoid blocking forever:

```
class MyTXM(TXM):
    motor_pv = TxmPV('txm:motorA', wait=False)
    sensor_pv = TxmPV('txm:sensorA')


txm = MyTXM()
# First set the actuator to the desired value
new_position = 3.
txm.motor_pv = new_position
# This will block until the sensor reaches the target value
tmx.wait_pv('sensor_pv', new_position)
```

1.3.5 Waiting on Multiple Process Variables

Warning: This feature should be considered experimental. It has been know to break during some operations, most notably setting the undulator gap.

By default, calling the `pv_put()` method will block execution until the `put` call has completed. This means that setting several PVs becomes a serial operation. This is the safest approach but is unnecessary in many situations. For example, setting the x, y and z stage positions can be done simultaneously. You can always use `wait=False` and handle the blocking yourself, however this is not always straight-forward and may involve messy callbacks. Using the `wait_pvs()` context manager takes care of this. Any PVs that are set inside the context will move immediately; if `block=True` (default) the manager will wait for them to finish before leaving the context.

```
txm = TXM()

# These move one at a time
txm.Motor_SampleY = 5
txm.Motor_SampleZ = 3

# This waits while both motors move simultaneously
with txm.wait_pvs():
```

(continues on next page)

(continued from previous page)

```
    txm.Motor_SampleY = 8
    txm.Motor_SampleZ = 9

# These move in the background without blocking
with txm.wait_pvs(block=False):
    txm.Motor_SampleY = 3
    txm.Motor_SampleZ = 12
```

This table describes whether if and when a process variable blocks the execution of python code and waits for the PV to achieve its target value:

Context manager	`pv_put(wait=True)`	`pv_put(wait=False)`
No context	Blocks now	No blocking
`TXM().wait_pvs`	Blocks later	No blocking
`TXM().wait_pvs(block=False)`	No blocking	No blocking

1.3.6 Locking Shutter Permits

Sometimes it's desireable to test portions of the codebase during downtime while the B-hutch is operating. In order to do this, however, it's important to ensure that the shutters, undulator and monochromator are not changed. Using the `TxmPV` descriptors makes this easy: any PV's that should not be changed can be given the `permit_required=True` argument to their constructor:

```
class MyTXM(TXM):
    SHUTTER_OPEN = 1
    my_shutter = TxmPV('32idc:shutter', permit_required=True)

    def open_shutter(self):
        """Opens the shutter so we can science!"""
        self.my_shutter = self.SHUTTER_OPEN


# This will not do anything
my_txm = MyTXM()
my_txm.open_shutter()

# This will control the PV as expected
my_txm = MyTXM(has_permit=True)
my_txm.open_shutter()
```

Note: There is no check that the C-hutch actually *has* permission to open the shutter, etc. It's controlled only by the `has_permit` argument given to the `TXM` constructor. Please be considerate.

1.3.7 Fast Shutter

The instrument is equipped with a "fast shutter" than protects the specimen from excessive X-ray exposure. Calling `enable_fast_shutter()` turns this feature on. If using the `run_scan()` context manager (recommended), the fast shutter is automatically disabled, otherwise the `disable_fast_shutter()` method should be called to return to normal behavior. The fast shutter respects `exposure_time()` attribute.

1.4 API reference

project Modules:

1.4.1 scanscripts

aps_02bm package

Subpackages

aps_02bm.run package

Module contents

Run-time scripts for the 2-BM microscope.

Submodules

aps_02bm.macros_2bmb module

aps_02bm.mosaic_tomo_fly_scan module

aps_02bm.tomo_scan_lib module

aps_02bm.tomo_step_scan module

Module contents

aps_32id package

Subpackages

aps_32id.run package

Submodules

aps_32id.run.energy_scan module

aps_32id.run.mosaic_tomo_fly_scan module

aps_32id.run.move_energy module

aps_32id.run.tomo_fly_scan module

aps_32id.run.tomo_step_scan module

Module contents

Submodules

aps_32id.txm module

Module contents

scanlib package

Submodules

scanlib.exceptions_ module

scanlib.tomo_scan_lib module

scanlib.txm_pv module

scanlib.tools module

Module contents

presented to the user in the GUI when running this script. In the example below, Several actions take place within a `run_scan()` context manager. This ensures that the current configuration is restored after the scan.

```
#!/bin/env python
"""An example script for controlling the sector 32 ID-C microscope."""

import logging

from scanlib import update_variable_dict
from aps_32id import NanoTXM

# Prepare for logging data to a file, or whatever
log = logging.getLogger(__name__)


# A dictionary with the options that can be used when invoking this script
variableDict = {
    'Parameter A': 0.1,
    'Parameter B': 505,
    # Logging: -1=no change, 0=UNSET, 10=DEBUG, 20=INFO, 30=WARNING, 40=ERROR,␣
↪50=CRITICAL
    'Log_Level': logging.INFO,
}


def getVariableDict():
    return variableDict


def run_my_experiment(param_a, param_b, log_level=20, txm=None):
    """Separate out the work-horse code so that it can be executed
    programatically. The ``txm`` parameter is intended for testing,
    where an instance of :py:class:`tests.tools.TXMStub` is used.

    Parameters
    ==========
    param_a :
      An experimental parameter.
    param_b :
      Another experimental parameter.
    log_level : logging.INFO
      How much detail to save to the logs.
    txm : NanoTXM, optional
      A NanoTXM object that represents the X-ray microscope. Useful
      for testing.

    """
    log.debug("Starting my experiment")
    # Create a TXM object to control the instrument
    if txm is None:
        txm = new_txm()
    # Run the experiment in this context manager so it stops properly
    with txm.run_scan():
        # Setup the microscope as desired
        txm.setup_hdf_writer()
        txm.start_logging(log_level)
        txm.setup_detector()
        txm.enable_fast_shutter() # Optional: reduces beam damage
```

(continues on next page)

(continued from previous page)

```
        txm.open_shutters()
        # Now do some tomography or XANES or whatever
        pass
        # Close the shutters and shutdown
        txm.close_shutters()


def main():
    # The script was launched (not imported) so load the variable
    # dictionary from CLI parameters
    update_variable_dict(variableDict)
    # Start the experiment
    run_my_experiment(param_a=variableDict['Parameter A'],
                      param_b=variableDict['Parameter B'],
                      log_level=variableDict['Log_Level'])


if __name__ == '__main__':
    main()
```

Python Module Index

Index