Step Scanning and Data Acquisition with Python and Epics Channel Access

Release 1.0

Matthew Newville

May 07, 2012

CONTENTS

1 StepScan Downloading and Installation **3**
- 1.1 Pre-requisites . . . 3
- 1.2 Downloads . . . 3
- 1.3 Development Version . . . 3
- 1.4 Installation . . . 3

2 Step Scan Concepts **5**
- 2.1 Positioners . . . 5
- 2.2 Triggers . . . 5
- 2.3 Counters . . . 6
- 2.4 Detectors . . . 6
- 2.5 Extra PVs . . . 6
- 2.6 Breakpoints . . . 6
- 2.7 Pre- and Post-Scan functions . . . 6

3 Using StepScan **7**
- 3.1 Defining a New Scan . . . 7
- 3.2 Running a Scan . . . 7

4 Spec-like Scans with StepScan **9**
- 4.1 Setting up and Configuring the spec emulation layer . . . 9
- 4.2 Using the spec emulation layer . . . 10

5 Using the StepScan GUI **11**

Python Module Index **13**

Python Module Index **15**

Index **17**

Epics StepScan is a library and (soon...) set of command-line and GUI applications for data acquisition with Epics Channel Access using Python. StepScan allows you to define and run step scans, in which a motorized stage or other variable is changed through a set of pre-defined positions (or steps), and a set of detectors are measured at each position. By using Epics Channel Access, nearly any motors, detectors, or Epics Process Variable can be scanned or counted in a StepScan. Using concepts are borrowed from the Epics SScan record and from commonly used Spec software, simple to moderately complex step scans can be built and run. With StepScan, the data collection happens in the python client, so that many other python libraries and environments can be coupled with Epics StepScan.

Though StepScan allows nearly any Process Variable to be scanned or counted during a scan, and thus allows scans to become quite complex, the StepScan library and applications allow simple scans to still be simple to define and run. A description of some terms in the *Step Scan Concepts* section is given to clarify the concepts used in StepScan and this documentation. Examples of simple scans and some more complex scans are given.

CHAPTER

ONE

STEPSCAN DOWNLOADING AND INSTALLATION

1.1 Pre-requisites

The StepScan library needs pyepics, a Python library for Epics Channel Access, and, in fact, is an add-on package to that library. StepScan also requires a few near-standard python modules: numpy, scipy, sqlalchemy, and wxPython. In order to save data to the HDF5, the hdf5 library, and the h5py module are also required.

1.2 Downloads

The latest stable version is available from PyPI or CARS (Univ of Chicago):

Download Option	Python Versions	Location
Source Kit	2.6, 2.7, 3.2	• stepscan-0.2.tar.gz (PyPI) • stepscan-0.2.tar.gz (CARS)

Installers for Windows will be made available soon.

If you have Python Setup Tools installed, you can download and install with:

```
easy_install -U stepscan
```

1.3 Development Version

To get the latest development version, use:

```
git clone http://github.com/pyepics/stepscan.git
```

1.4 Installation

To install from source, use:

```
python setup.py install
```

from the StepScan folder.

1.4.1 Running the StepScan GUI

Soon, this will work:

    python stepscan_gui.py

or click on the icon.

1.4.2 Using StepScan from Python

To use StepScan from Python scripts, use:

```
from epics import stepscan
```

CHAPTER

TWO

STEP SCAN CONCEPTS

A step scan is simply a **for loop** that iterates a collection of **Positioners** (any Epics PV – not restricted to Motors) through a set of pre-determined position values. At each position, a set of detectors are **triggered** (which is to say, collection is started), and then waited upon until they announce they are complete. At that point a collection of **Counters** (again, any Epics PV, though for those holding waveform values see the notes below) are read, and the loop starts over, moving on to the next point in the list of positions.

A StepScan also contains several other components::

- extra_pvs: a set of PVs to record once (or a few times) for each scan, but not at every point in the scan.
- breakpoints: a series of points in the scan at which data collected so far is dumped to file, and perhaps other tasks are done.
- pre_scan: a set of functions to call just prior to beginning the scan.
- post_scan: a set of functions to call just after the scan completes.
- at_break: a set of functions to call at each breakpoint (after the data is written to disk).

2.1 Positioners

Positioners are what is changed in the scan – the dependent variables. The can be represented by any Epics PV, such as Motors, temperatures, currents, count times, or a combination of these. Scans can have multiple positioners, either moving in tandem to make a complex motion in the positioner space, or independently to make a mesh in the positioner space.

In addition to a PV, each positioner will have an array of positions which it should take during the scan. The Positioner holds the full array positions. There are methods available for creating this from Start, Stop, Step, and Npts parameters for simple linear scans, but you can also give it a non-uniform array,

If you have multiple positioners, they must all have an array of positions that is the same length.

2.2 Triggers

A Trigger is something that starts a detector counting. The general assumption is that these start integrating detectors with a finite count time (which may have been set prior to the scan, or set as one of the Positioners). The scan waits for these triggers to announce that they have completed (ie, the Epics "put" has completed). That means that not just any PV can be used as a Trigger – you really need a trigger that will finish.

For many detectors using standard Epics Records (Scalers, MCAs, multi-element MCAs, AreaDetectors), using a Detector (see below) will include the appropriate Trigger.

2.3 Counters

A Counter is any PV that you wish to measure at each point in the scan.

For many detectors using standard Epics Records (Scalers, MCAs, multi-element MCAs, AreaDetectors), using a Detector (see below) will automatically include a wide range of Counters.

2.3.1 Counters for waveforms

Currently, these are not directly supported, mostly as it is not yet determined how to save this data into a plain ASCII file. If more complex file formats are used, these could be supported.

Note that a Trigger for an AreaDetector can be included, which might cause it to save data through its own means.

2.4 Detectors

Detectors are essentially a combination of Triggers and Counters that represent a detector as defined by one of the common Epics detector records. These include Scalers, MCAs, multi-element MCAs, and AreaDetectors.

2.5 Extra PVs

Extra PVs are simply Epics PVs that should be recorded once (or occasionally) during a scan, but but not at every point in the scan. These might include configuration information, detector and motor settings, and ancillary data like temperatures, ring current, etc.

These should be supplied as a list of *(Label, PVname)* tuples. The values for these will be recorded at the beginning of each scan, and at each breakpoint.

2.6 Breakpoints

Breakpoints are points in the scan at which the scan pauses briefly to to read values for Extra PVs, write out these and the data collected so far to disk, and run any other user-defined functions.

A typical short, one dimensional scan will not need any breakpoints – the data will be written at the end of the scan. On the other hand, a two dimensional mesh scan may want to set a breakpoint at the end of each row, so that the data collected to date can be read from the datafile.

To set breakpoints, just put the indices of the points to break after (starting the counting from 0) into a *scan.breakpoints* list.

Users can add their own functions to be run at each breakpoint as well.

2.7 Pre- and Post-Scan functions

These functions are run prior to and just after the scan has run. Detectors and Positioners may add their own `pre_scan()` and `post_scan()` methods, for example to place a detector in the right mode for scanning.

CHAPTER

THREE

USING STEPSCAN

This section describes how to use a StepScan from python scripts. See *Step Scan Concepts* for a description of terms and basic concepts.

3.1 Defining a New Scan

To define a new scan, you create a scan object, set up the Positioners, Detectors, and Counters for the scan, add optional extra features such a as extra_pvs and breakpoints, and then run the scan.

3.2 Running a Scan

CHAPTER

FOUR

SPEC-LIKE SCANS WITH STEPSCAN

Spec is a commonly used program for collecting data at synchrotrons, as it can not only communicate with Motors and Detectors, but can also work in the geometries needed to run diffractometers, and includes a convenient macro language. Spec is not without limitations, and a reasonable goal of StepScan would be to replace some uses of Spec. Of course, Python can replace the macro language, and Epics Channel Access can control Motor and Detectors. To better enable a transition of some scanning away from Spec and toward Python, StepScan has a `spec_emulator` module that provides Spec-like functionality, with scanning methods `ascan()`, `dscan()`, `a2scan()`, `d2scan()`, `lup()`, and `mesh()` that closely match the Spec routines of the same name.

4.1 Setting up and Configuring the spec emulation layer

To emulate Spec, one creates a `SpecScan` object:

```
>>> from epicsapps.stepscan import SpecScan
>>> spec = SpecScan()
```

This creates an empty SpecScan object, which does not have any configuration information about Motors and Detectors to use during a scan. Motors can be added to the configuration with:

```
>>> spec.add_motors(x='XXX:m1', y='XXX:m2', theta='XXX:m3')
```

where you can add as many *label=PV_NAME* keyword arguments as you like. Later on, when scanning, you'll use the labels to mean the underlying PVs. As with all of StepScan, the PVs are not required to be motors.

We'll also need to add some detectors. StepScan supports many kinds of detectors, but for simple scans with point-detectors, an Epics Scaler record usually suffices, so we'll start with that:

```
>>> spec.add_detector('XXX:scaler1', kind='scaler', nchan=8, use_calc=False)
```

By saying *kind='scaler'*, StepScan will know what to use for a Trigger, how to set the dwell time, and what Counters to count – all the channels 1 through 8 that have a non-blank name will be collected. There is also the option to set *use_calc=True* to use the calc record associated with the Scan Record, which can be used to support offset values and simple calculations.

As with other parts of StepScan, we can add *extra PVs* – values to be recorded at the start of each scan, with a list of (description, pvname) tuples:

```
>>> spec.add_extra_pvs((('Ring Current', 'S:SRcurrentAI.VAL'),
                        ('Ring Lifetime', 'S:SRlifeTimeHrsCC.VAL')))
```

We haven't yet set the name of the output file. This is still a work-in-progress, but currently, the Spec file is not emulated in detail, but an ASCII file is written to for each scan. We can change this anytime with:

```
>>> spec.filename = 'myoutput.dat'
```

At this point, we have a fully configured (if minimal) Spec emulator, and can start running scans. It is likely

4.2 Using the spec emulation layer

To perform a simple absolute scan, we can use `ascan()`, which would be as simple as:

```
>>> spec.ascan('x', 0, 0.5, 51, 0.5)
```

which scans Motor 'x' between 0 and 0.5 in 21 steps, counting for 0.5 second per point.

Other supported scan routines are `dscan()`, which is like `ascan()` but with start and stop positions relative to the current position:

```
>>> spec.dscan('x', -0.1, 0.1, 41, 0.25)
```

The routines `a2scan()` and `d2scan()` simultaneously moves 2 Motors (in absolute and relative coordinates, respectively):

```
>>> spec.a2scan('x', 0, 1.0, 'y', 0, 0.2, 21, 0.5)
>>> spec.d2scan('x', -0.5, 0.5, 'y', -0.1, 0.1, 21, 0.5)
```

That is, these move along a line in 'x'-'y' space.

The routines `a3scan()` and `d3scan()` simultaneously moves 3 Motors (in absolute and relative coordinates, respectively):

```
>>> spec.a3scan('x', 0, 1.0, 'y', 0, 0.2, 'theta', 10.0, 10.1, 21, 0.5)
```

Finally, `mesh()` will make a 2-dimensional mesh, or map:

```
>>> spec.mesh('x', 10, 11, 21, theta', 10.0, 10.1, 11,  0.25)
```

This will make a 21 x 11 pixel map, moving 'x' between 10 and 11 for each 'theta' value, so that 'theta' scans slowly.

CHAPTER

FIVE

USING THE STEPSCAN GUI

PYTHON MODULE INDEX

S

spec_emulator, 9

PYTHON MODULE INDEX

S

spec_emulator, 9

INDEX

S

spec_emulator (module), 9