JOURNAL OF SYNCHROTRON RADIATION

ISSN 1600-5775

Py4Syn: Python for synchrotrons

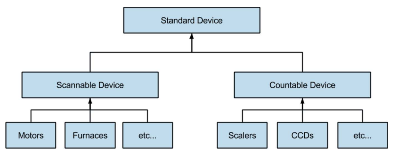
H. H. Slepicka,* H. F. Canova, D. B. Beniz and J. R. Piton

CNPEM-LNLS, Brazilian Synchrotron Light Laboratory, Campinas, São Paulo, Brazil.
*Correspondence e-mail: hhslepicka@gmail.com

Received 7 May 2015
Accepted 20 July 2015

Keywords: Python; *Py4Syn*; data acquisition; devices.

In this report, *Py4Syn*, an open-source Python-based library for data acquisition, device manipulation, scan routines and other helper functions, is presented. Driven by easy-to-use and scalability ideals, *Py4Syn* offers control system agnostic solution and high customization level for scans and data output, covering distinct techniques and facilities. Here, most of the library functionalities are described, examples of use are shown and ideas for future implementations are presented.



1. Introduction

It is widely recognized that data acquisition is one of the most important processes within any laboratory. From manually to automatically, it can be executed in many ways, using commercial and open-source frameworks. As this process is related to plenty of technologies involving a continuous evolution of techniques, there is a growing need for a framework capable of allowing scientists to define their own code. Therefore, it is crucial that the development of new data acquisition frameworks allows the creation of scripts. This should be conceived in a simple fashion, not only for custom routines but also for complex scans. Nevertheless, the framework must not be tied to a particular solution or data format.

Nowadays, for almost every technique in use, custom software is developed using its own standards, scripting language and data output. Among all frameworks for data acquisition and devices manipulation, we evaluated four solutions used or tested by Brazilian Synchrotron Light Laboratory (LNLS) beamlines.

SPEC (Certified Scientific Software, 2015), developed by Certified Scientific Software, is widely used by the X-ray scientific community and offers support to a large amount of equipment and also EPICS (Argonne National Laboratory, 1994). *SPEC*'s code already covers most of the needs regarding scans and device manipulation but on the other hand this tool is not open-source, charging an annual fee for every license in use if support and updates are needed. By having its environment based on a single thread, a large set of possibilities, like parallel processing, would depend on external code or tools. As *SPEC* is not an open-source solution, changes in its core code, for corrections and improvements, are more difficult. As in any commercial solution, bugs discovered and modifications are subject to the company's discretion. The development of custom code, *e.g.* different scan routines, requires *SPEC*'s own programming language, quite similar to C, and in some occasions its error messages are not clear.

OpenGDA (Diamond Light Source, 2003), by Diamond Light Source, offers an open-source framework for data acquisition using Java and based on the Eclipse RCP graphical interface and counts on an embedded Jython interpreter for scripting and control over the command line. For custom developments of scripts and the GUI, users have a steep learning curve, mainly in terms of the requirement of Java programming language and Jython.

Another evaluated solution was *APSpy* (Toby *et al.*, 2013), maintained by the Advanced Photon Source (APS). This is an open-source Python-based beamline control scripting tool with EPICS support. As an open-source solution and developed in Python, the learning curve from users was smooth, but, as mentioned on the tool website, this code was developed for operations at station 1-ID at APS, with configuration details specific to this beamline inside the code. Also, a large effort would be required to remove the specific configurations and behaviors.

In addition to the described tools, there are custom scripts developed in-house, another very common solution. Usually, those scripts are made to solve a specific problem in a very short time, in most of the cases at the cost of quality, resulting in a non-standardized code, difficult to maintain and of low reliability. This diversity of control software creates an environment which limits the collaborations and exchange of codes among facilities and perhaps even among different experimental stations in a given facility.

In this manuscript we report on *Py4Syn* (http://py4syn.readthedocs.org/), an open-source Python-based library which provides, from its basic principles, high-level abstraction for device manipulation, scan routines, interactive data plots, fitting of functions and customized data output. This is an advance over frameworks currently in use throughout some experimental stations.

2. Architecture

With the aim of producing a modular, readable and easy-to-use library, *Py4Syn* was implemented using Python programming language. Python, combined with some of its standard packages, *e.g.* Numpy (NumPy Developers, 2013), Matplotlib (Hunter, 2007), SciPy (Jones *et al.*, 2007), provides a strong and easy-to-learn platform. Also, it offers integration with code written in other languages like C and Fortran as it adopts shared libraries for critical performance tasks (Behnel *et al.*, 2011). Therefore, it is possible to reach a nearly real-time feedback of control instrumentation and data as *Py4Syn* offers a perfect integration with EPICS, the control software in use at LNLS beamlines, through a high-level Python library, PyEpics (Newville, 2014*a*). Mainly, our software is organized in a modular converged triad: 'epics', 'utils' and 'writing'; these are carefully described below.

3. Epics module

Briefly, this module holds helper classes to manipulate most of the common devices present at beamlines like motors, counters and CCDs, among others.

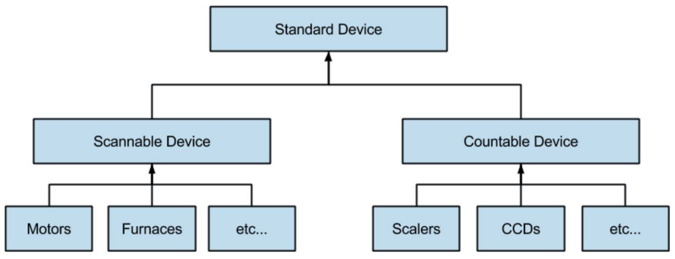


Figure 1
All devices extend from the 'Standard device' interface and also from the 'Scannable' or 'Countable' interfaces according to the device type and desired functionality.

3.1. Devices structure

In *Py4Syn* the devices are divided between two main groups, the 'Scannables' and the 'Countables', as described in the scheme in Fig. 1.

The 'Standard device' interface in *Py4Syn* holds the basic information of every device, while only a mnemonic field is present so far. The Scannable interface represents the common operations of all devices that can be used in a scan routine, *e.g.* define a setpoint, read the current value, check limit values, check if the device is idle or busy, among others. On the other hand, the Countable interface represents all equipments that can monitor or acquire signals and, as with the Scannable interface, this implements common methods like defining the acquisition time, starting/stopping an acquisition, *etc*. Altogether, these interfaces provide a standard to create and manipulate devices inside the library and also allow new custom device development and transparent use of the existing functionalities.

4. Utils module

The utils module is composed of helper functions for motor movement, counters, scans, plots and fit. Below, we will describe each component of this module.

4.1. Motor functions

One of the most common tasks in a beamline is the setup of experiments, which requires the precise moving of motors and devices. Hence, to speed up the processes, *Py4Syn* comes with a large set of built-in functions for the most common operations, *e.g.* check the limits of all motors, absolute/relative motors movement and more. Most of these commands have a syntax similar to the corresponding ones normally used in a software known to many beamlines, *SPEC*. We chose to keep this syntax to reduce the learning curve for beamline scientists and users (see Fig. 2).

4.1.1. Pseudo motors. In *Py4Syn*, in addition to the common motor implementation using the EPICS motor record, we also provide the possibility to create a pseudo motor, or virtual motor, by defining relationships between a virtual axis and the real ones. In order to represent this relationship, the user can provide a custom formula using plain Python language and for this purpose functions defined in maths and NumPy libraries are also available. Especially for

```
from py4syn.utils.motor import *
mtop = 'mtop'
mbot = 'mbot'

# creates a motor using the mnemonic and PV.
createMotor(mtop, 'SOL:DMC1:m3')
createMotor(mbot, 'SOL:DMC1:m4')

# move the motor mtop to the position 10.
umv(mtop, 10)

# print the position of all motors registered.
wa()

# store the motor mtop position in the variable mtopPosition.
mtopPosition = wmr(mtop)
```

Figure 2
Example script for motor setup movement and position check using *Py4Syn*.

NumPy functions, users must explicitly name them, for example, `numpy.linalg.solve()`. As many of the axis relation formulas depend on the real and target motors and virtual motors positions, *Py4Syn* offers six special functions to reach this values as shown in Table 1.

A real-world example would be implementing the control of a double-crystal monochromator with fixed exit, where the second crystal must be translated in two directions t and g (Fig. 3) to keep the beam height at the exit constant along the rotation. Assuming the rotation center of the monochromator is in the middle of the surface of the first crystal, the incoming beam at an angle θ_0 and the reflected beam at an angle θ_h define (Fig. 3) the translations t and g of the center of the second crystal with respect to the center of the first crystal.

Using the geometric relationships of Fig. 3, the equations defined below are for the translations, t (1) and g (2), and also the Bragg angle (3), which is defined by the crystal and energy according to Bragg's law,

$$t = \frac{h}{\sin\theta_0 + \sin\theta_0 \tan\theta_h}, \tag{1}$$

$$g = \frac{h\tan\theta_h}{\sin\theta_0 + \sin\theta_0 \tan\theta_h} \tag{2}$$

and

$$\theta_{\rm B} = \arcsin\left(\frac{0.62}{dE}\right), \tag{3}$$

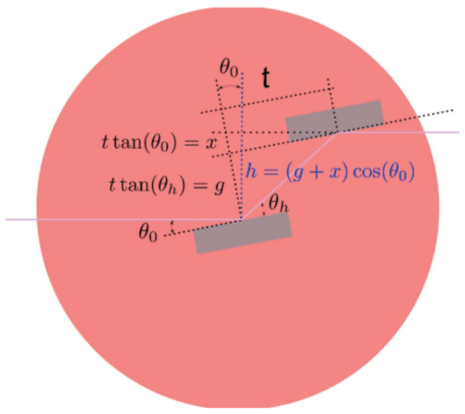


Figure 3
Angles and distances defined in the monochromator circle of rotation.

Table 1
Special functions provided by *Py4Syn* in order to recover real and target positions of motors and virtual motors.

Function	Description	Usage
A	Actual position of a motor	A[mne]
T	Target position of a motor	T[mne]
AD	Actual dial (encoder) position of a motor	AD[mne]
TD	Target dial (encoder) position of a motor	TD[mne]
AR	Actual raw (steps) position of a motor	AR[mne]
TR	Target raw (steps) position of a motor	TR[mne]

where d is the spacing between the planes in the atomic lattice in nm and E is the energy of the incident beam in keV.

Assuming a Si(111) crystal, it is possible to map this monochromator and its degrees of freedom using pseudo motors inside *Py4Syn* as shown in Fig. 4.

4.1.2. Validations and error handling. One of the main concerns regarding motor movements, especially in virtual motors, is the validation of this action prior to the real movement. In *Py4Syn*, the motor and pseudo motor classes implements a method called *canPerformMovement* which provides limit checking and calculates whether the desired movement can be executed regardless of hardware limits. In a negative response, an explanatory message is returned and the execution is stopped; otherwise, the movement occurs directly. However, if any of the motors involved in a movement stops for limit switch or failure, then an error message is raised as an output.

4.2. Counter functions

The counter functions follow almost the same syntax familiar to *SPEC* users. So far only a few functions are needed, as counting represents a very simple process.

Using *Py4Syn*, one can create a new counter, enable or disable a specific channel, start and stop a counting process and more (see Fig. 5). The full list of functions and also usage examples can be found in the library manual.

4.2.1. Pseudo counters. As in the pseudo motors implementation, *Py4Syn* offers the creation of pseudo counters, or virtual counters, by defining a custom formula to obtain the value of a virtual channel using real device channel values. Similarly to the pseudo motor, the user can supply a custom formula using plain Python language and functions available in maths and NumPy libraries. To access the real counter values, a special code is needed. An example is presented in Fig. 6, where two counters, `det` and `mon`, exist and the spectroscopy is calculated in a virtual channel. For instance, one would be the incoming intensity (I_0) and the other the transmited intensity (I_1). In this case the formula for the pseudo counter relationship would be `C[det]/C[mon]` where `C[mnemonic]` corresponds to the value in the channel defined by the given mnemonic.

4.3. Plot functions

In order to reach the best performance and achieve a near real-time live plot, a plotter class based on the Matplotlib multiprocess example was implemented. When created, this

```
from py4syn.utils.motor import *
# Define the formula to calculate the Bragg Value
braggBackFormula = "asin(1.977/A[energy])"

# Define the relation between thetaCrystal and bragg target value
braggDict = {"thetaCrystal": "T[bragg]"}

# Define the formula to calculate the Energy Value
energyBackFormula = "1.977/sin(A[thetaCrystal])"

# Define the relation between the needed motors and the energy target value
energyDict = {"bragg": "asin(1.977/T[energy])",
    "gap": "A[beamOffset]*tan(T[bragg])/ (sin(T[bragg])+(cos(T[bragg])*tan(T[bragg])))",
    "trans":"A[beamOffset]/(sin(T[bragg])+(cos(T[bragg])*tan(T[bragg])))"}

# Define the formula to calculate the Beam Offset
beamOffsetBackFormula = "2*A[trans]*tan(A[bragg])*cos(A[bragg])"

# Define the relation between the needed motors and the beam offset
beamOffsetDict = {"trans": "T[beamOffset]/(sin(A[bragg])+cos(A[bragg])*tan(A[bragg]))",
    "gap": "(T[beamOffset]*tan(A[bragg]))/(sin(A[bragg])+(cos(A[bragg])*tan(A[bragg])))"}

# Crete a simple motor for the 2nd Crystal Translation
createMotor("trans", "SOL:DMC1:m1")

# Crete a simple motor for the 2nd Crystal Gap
createMotor("gap", "SOL:DMC1:m3")

# Crete a simple motor for the 1st Crystal Rotation
createMotor("thetaCrystal", "SOL:DMC1:m4")


# Create the Pseudo-motor Bragg, Energy and BeamOffset with the respective formulas
# and relations
createPseudoMotor("bragg", "Bragg Angle",
                  backFormula=braggBackFormula, forwardDict=braggForwardDict)

createPseudoMotor("energy", "Energy in kEv",
                  backFormula=energyBackFormula, forwardDict=energyForwardDict)

createPseudoMotor("beamOffset", "Beam offset (mm)",
                  backFormula=beamOffsetBackFormula, forwardDict=beamOffsetForwardDict)

# Print all motors positions
wa()

# Move the motor Energy to the value 2.
mv("energy", 4, wait=True)

# Print all motors positions
wa()
```

Figure 4
Example script for double-crystal monochromator using *Py4Syn* pseudo motors.

plotter spawns a new process responsible for data plotting and graph screen updating without overhead to the main thread, responsible for the scanning and other activities. All data and configuration parameters are transferred to the spawned process using a FIFO queue. So far it supports only simple plots with subplots and axis overlays, as shown in Fig. 7. We are looking forward to implementing more visual resources and tools such as axis formatting in the near future.

4.4. Scan functions

Scans are the main operation used at beamlines, either for data acquisition as for setup, *e.g.* finding the beam, or alignment of samples, among others. In *Py4Syn* all scans are step scans. Thus, for every point specified, the devices will be properly set to a new value and when all of them reach the respective setpoints a count process will start. At the end of this counting the process will repeat until the end of the points list. *Py4Syn* can handle three scan types: *scan*, *mesh* and *timescan*, which we describe now.

4.4.1. Scan. Different from the other types, when using the scan all devices in use must have the same number of steps since they will step together. One can specify as many devices as wanted, in view of the fact that the library has no limit regarding the number of devices in a scan. Therefore, only devices that implement the Scannable interface can be used.

The simplest syntax of the scan command inside *Py4Syn* is represented in the formula below, where `device1...deviceN` can be either the mnemonic of a motor previously registered

```
# Create an EPICS Scaler
scaler = Scaler("SOL:SCALER", 10, "scaler1")

# Create a simulated counter
scalerSIM = SimCountable("ANY:PV:HERE", "simMnemonic")

# Create a channel using the real scaler,
# channel number 1 and a factor.
createCounter("seconds", scaler, 1, factor=1e+7)

# Create a channel that will act as a monitor
createCounter("mon", scaler, 10, monitor=True)
createCounter("cyberSim", scalerSIM, 2)

# Start a count process until 10000 counts in the monitor
# and show the results for each channel at the end.
ct(10000)

# Start a count process with 2 seconds of integration
# time. Returns the counts as a map.
counts = ctr(2, use_monitor=False)

print("Counts on cyber: ", counts['cyberSim'])
```

Figure 5
Example script for counter and channels setup, counting process over a monitor and counting over time using *Py4Syn*.

```
# Create a Simulated Counter
scaler = SimCountable("SOL:SCALER01", 2)

# Create two channels: det and mon
createCounter("mon", scaler, 1)
createCounter("det", scaler, 2)

# Create a Pseudo counter with the formula
pseudoCounter = PseudoCounter("relation", "C[det]/C[mon]")

# Create a new channel based on this pseudo-counter
createCounter("relation", pseudoCounter)

# Start a count process
ct(1)
```

Figure 6
Example script for a pseudo counter using *Py4Syn*.

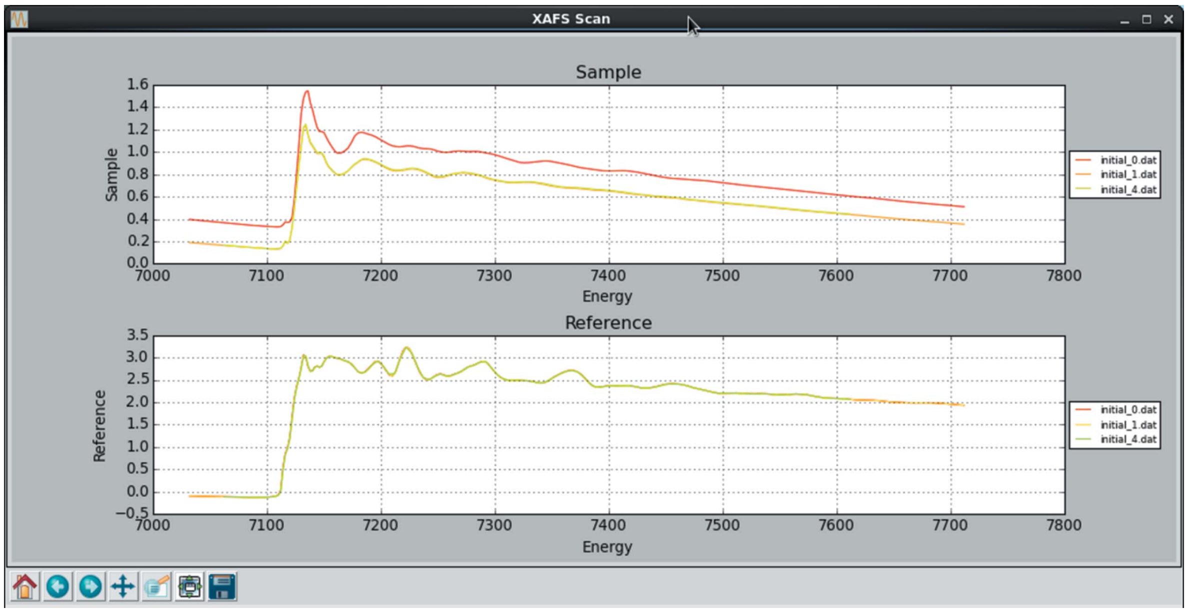


Figure 7
Graph comparing three EXAFS spectra: at the top, comparing the sample value and the natural logarithm of the relationship between ionization chambers I0 and I1; at the bottom, the reference value given by the natural logarithm of the relationship between I1 and I2.

or a reference to any device that implements the Scannable interface, `start1...startN` are scalar values defining the initial setpoint, `end1...endN` are the final setpoints, `steps` represents the number of steps to be executed in this scan and `time` is the value (in seconds) to be used for the count process,

$$\texttt{scan(device1, start1, end1, deviceN, startN, endN, steps, time)}$$

A scan command can also be invoked using an array of points instead of a start/end pair to create a custom list of points. The `points` parameter is an array or list of setpoints and the main restriction is that the number of elements must be equal to the value informed in the steps parameter. One can also inform either an array for one device and a start and end pair for another; there is no restriction regarding this, except the number of steps rule mentioned earlier,

$$\texttt{scan(device1, points1, deviceN, pointsN, steps, time)}$$

Another available option is the use of an array or list of delay values after the `time` parameter in order to generate a delay prior to a next point acquisition. This functionality is very useful when, for example, one wants to carry out a sequence of counts in the same position or wait until the next setpoint is moved to. Also the `time` parameter can be replaced by an array or list of values in order to produce custom count times for every point. In this case the `length` parameter must be the same as the number of points. In general, at least one device must be informed with the corresponding start/end pair or points. The parameters `time` and `delay` are optional. If not explicit, `time` receives the value '1' (corresponding to 1 s of counting time) and `delay` receives 'None', which means that no delay will be added to the process. The scan command syntax can be fully defined by the following expression,

$$\texttt{scan([device, [[start, end], points]]}^{+}\texttt{, steps, [time], [delay])}$$

4.4.2. Mesh. Mesh allows users to define a specific number of steps for every device, producing an *N*-dimensional matrix, where for each step of the current device the next one will go through all of its respective points and so on for all devices in use. Similarly to the scan command, there is no limitation regarding the number of devices involved. The only restriction is that the used devices must implement the Scannable interface. The most simple syntax of a mesh command inside *Py4Syn* is

$$\texttt{mesh(device1, start1, end1, steps1, deviceN, startN, endN, stepsN, time)}$$

This command also supports an array of points instead of a start/end pair to create a custom points list, but in this case the steps parameter must not be used. One can also inform either an array for one device and a start and end pair for another. This way the general syntax of a mesh is presented below, with a restriction: at least one device must be informed with the corresponding start, end and steps group or points array. `Time` and `delay` follow the same rules as Scan where, if not present, `time` receives the value '1' and `delay` receives 'None', resulting in no delay.

$$\texttt{mesh([device, [[start, end, steps], points]]}^{+}\texttt{, [time], [delay])}$$

4.4.3. Timescan. In this special type of scan no single device is used as this scan runs over time. It is very useful to monitor signals, especially during setting up and commissioning of experiments and adjusts. When invoking a timescan, `time`, `delay` and `repeat` are optional parameters. The default values are 1 for `time` (corresponding to 1 s of counting time), 1 for `delay` (representing a delay of 1 s between each point) and −1 for `repeat`, which means that this scan will run until it receives a user interruption. The general syntax of a timescan is

```
timescan([time], [delay], [repeat])
```

4.4.4. Callbacks. Callbacks are user-defined custom functions that are executed at certain moments during the scanning process. For now, there are seven callbacks available at *Py4Syn*:

(i) Pre Scan Callback. This callback is executed at the beginning of a scan, *e.g.* move all motors to the start position.

(ii) Pre Point Callback. This callback is executed before a point (before motors movement, or any other device setup), *e.g.* send specific configuration to detectors.

(iii Pre Operation Callback. This callback is executed before the operation, before launching the counters, *e.g.* open the shutter.

(iv) Operation Callback. This is the main callback and it is executed while the counters are running, *e.g.* prepare robotic arm to next sample.

(v) Post Operation Callback. This callback is executed after the operation, after the counters stop and before the plot and screen update, *e.g.* close the shutter.

(vi) Post Point Callback. This callback is executed right after the operation callback and before a new movement, *e.g.* data management routines.

(vii) Post Scan Callback. This callback is executed at the end of the scan, *e.g.* send an email to the user.

To define a custom callback, users must supply a function that receives a special parameter (Fig. 8), named `**kwargs`. This parameter is a dictionary containing the scan object (`scan`), an array with index value for each device (`idx`) and another array with the value of each device (`pos`).

4.4.5. Scan functions core flowchart. All scans execute the same core operations. Here we describe these operations according to the flowchart presented in Appendix *A*.

```
# Creating my function that will be executed
# in the choosen callback.
# This function needs to receive a parameter **kwargs
def myCallback(**kwargs):
    scanObject = kwargs['scan']
    indexArray = kwargs['idx']
    positionArray = kwargs['pos']

    print('Message from my callback')

# This show how to move back to the default callback
setPreScanCallback(defaultPreScanCallback)

# Set my function to the operation callback
setOperationCallback(myCallback)


# Run the scan with the new configurations
scan('m1', 0, 180, 10, 1)
```

Figure 8
Example script for a custom callback code in a scan using *Py4Syn*.

Once the scan command is invoked, all parameters are validated according to the scan type (scan, mesh and timescan). If any of the parameters is invalid or missing, an error is raised and the scan routine ends. After the verification of parameters, a setup of Plotter, FileWriter and data storage map is executed and a custom callback function, the Pre Scan Callback, is called if defined.

Entering the scan main loop, the Pre Point Callback function is executed prior to all other tasks and after elapsing of the proper delay time, if present. Right after the delay time, all devices are set to the corresponding setpoint and the scan framework will wait until the last of them reaches the target value or stop. Previous to the count process starting, another callback, Pre Operation Callback, is invoked and after the start of all counters, while the count process is still running, Operation Callback is executed and allows users to create custom processes while the framework waits for the count to finish.

As soon as the counting process is finished, and the acquired data are saved on the corresponding key of the data map, the Post Operation Callback is called to provide data processing prior to plot, print on screen and save data on disk (if the partial write option is enabled). After all, another callback, Post Point Callback, is executed prior to advancing to the next point.

After completing the list of points a Gaussian fit is executed, if the FIT_SCAN flag is enabled. Then statistical information such as peak value, peak position, center of mass, among others, are presented to the user and become available after the scan through the proper get method. Finally, if partial write is disabled, all data are flushed to disk. Furthermore, in the case of any error or user interruption during the scan process, all the collected data are saved on disk using the corresponding *FileWriter* to avoid data loss.

4.4.6. Handling the data. When running a scan, the data are saved to disk, partial or not, if the proper output was configured, but, despite all this, data are stored in a dictionary kept in memory in order to provide fast access to the data for processing and visualization. This dictionary, which can be accessed through the global variable SCAN_DATA or the *getScanData()* method, contains valuable information about the data acquired but also a reference to the scan object as shown in Table 2.

4.4.7. Special variables. In the scan module there are a few global variables that can be read and configured. These variables offer the possibility of customizing the scan behavior and also access information generated by the scan routines as we present in Table 3. The complete list including the corresponding methods to get and set the variables values is available in the library documentation.

4.5. Fit Functions

Py4Syn makes use of the LMFIT library (Newville, 2014*b*) which provides a high-level interface for curve fitting for Python using the Levenberg–Marquardt method. This library offers a model approach for the curve-fitting problem and

Table 2
Scan data dictionary indexes and contents.

Index	Content
points	Array of points, [0, 1, 2, 3, . . . , *N*]
scan_object	Reference to the scan object
devices mnemonic	For each device used in the scan an entry to store this device value across the scan is created
counters mnemonic	For each counter registered an entry to store this device value across the scan is created
scan_start	Timestamp of scan start
scan_end	Timestamp of scan end
scan_duration	Scan duration (scan_end − scan_start)
Any other user data	Any value you want, must be created using the *createUserDefinedDataField*

Table 3
Part of the special variables available in the *Py4Syn* scan module and its contents.

Variable	Content
SCAN_DATA	Dictionary that contains all scan-related data
SCAN_CMD	String with the last scan command executed
FWHM	Double that represents the FWHM value
FWHM_AT	Double that represents the FWHM position
COM	Double that represents the COM (center of mass) value
PEAK	Double that represents the peak value
PEAK_AT	Double that represents the peak position
MIN	Double that represents the minimum value
MIN_AT	Double that represents the minimum value position
FITTED_DATA	Array that represents the best fit values
FIT_RESULT	ModelFit with fit result information
FIT_SCAN	Boolean that represents if we should or not fit the scan data at end of scan; default is 'True'
PRINT_SCAN	Boolean that represents if we should or not print to the terminal the scan information; default is 'True'
PLOT_GRAPH	Boolean that represents if we should or not create the real-time plot; default is 'True'

counts using many built-in models, such as step and peak-like models. Also the models can be mixed and transformed in a composite model providing a powerful tool for curve fitting of linear and non-linear models.

The fit module also contains an implementation of a one-dimensional total variation denoising filter (Condat, 2013) for data analysis and to improve curve-fitting results.

5. Writing module

The writing module contains the *FileWriter* interface that is the abstraction layer between the data generated in *Py4Syn* scans and any file format, *e.g.* text, binary, xml, HDF (HDF Group *et al.*, 2014). By implementing two simple methods, *writeHeader* and *writeData*, one is capable of creating a Python class to produce a file in the format and structure that supply its needs, *e.g.* HDF with NeXus (Klosowski *et al.*, 1997), without changes in the scan command and code.

By default, to avoid problems for users regarding data analysis software compatibilities, the *DefaultWriter* produces a *SPEC*/PyMCA-like text file as data output. We chose to keep this format as it is the most common in use on our beamlines and beamlines abroad.

6. Conclusions

Py4Syn is a new solution for data acquisition and device manipulation, not only for synchrotron facilities but for any kind of laboratory. By being a control system agnostic solution and based on Python, its acceptance and use by scientists in our laboratory has shown that *Py4Syn* successfully met the locally desired needs.

So far the library is in use at LNLS at the following beamlines: small-angle X-ray scattering (SAXS1 and SAXS2), X-ray micro-tomography (IMX), X-ray absorption and fluorescence spectroscopy (XAFS2 and DXAS), X-ray diffraction (XRD1) and also ultraviolet and soft X-ray spectroscopy (PGM, SGM, TGM and SXS).

As this library is still an ongoing work, there are functionalities yet to be implemented and integrated, such as support for diffractometers and reciprocal space calculation, improvements in the plotter and increasing the number of default devices and simulations, among other users' requests.

APPENDIX *A*
Scan functions core flowchart

The scans core flowchart as described in §4.4.5 is presented in Fig. 9. The simple boxes represent routines that can be implemented or customized by users, *e.g.* '*Execute Operation Callback*', that can be changed using the proper method *setOperationCallback*, as shown in §4.4.4.

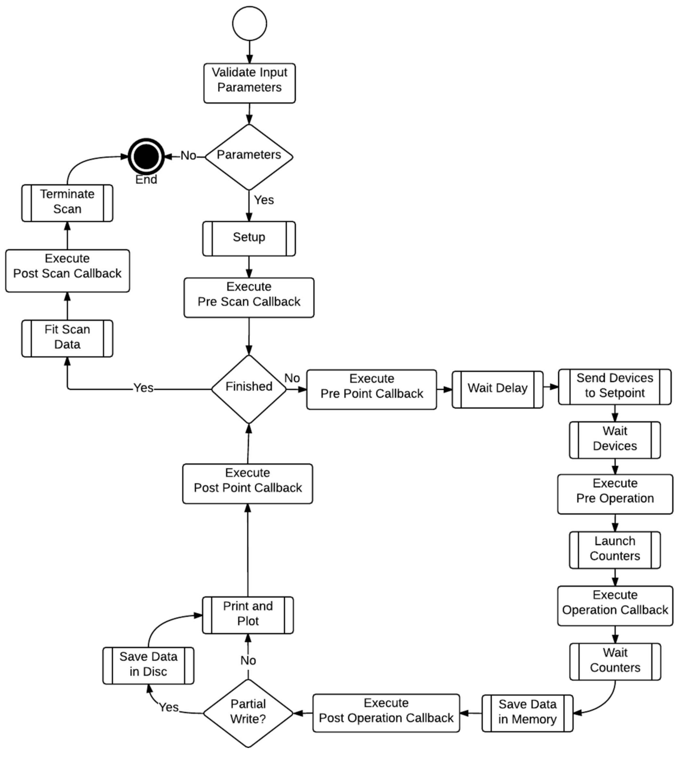


Figure 9
Py4Syn scan functions core operations flowchart.

Acknowledgements

This work was partially funded by Petrobras through the Labweb II project. The authors gratefully acknowledge the SOL group members for their help testing the library. The authors are also grateful for the guidance and valuable suggestions of the two anonymous referees. Many thanks to Dr Daniela Coelho for providing the EXAFS data for the plotter example, Dr Eduardo Miqueles for the revision and help with LaTex formatting, Dr Harry Westfahl Jr for the revision and to the beta users for their valuable suggestions and trust.

References

Argonne National Laboratory (1994). *EPICS; experimental physics and industrial control system*, http://www.aps.anl.gov/epics/.

Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D. S. & Smith, K. (2011). *Comput. Sci. Eng.* **13**, 31–39.

Certified Scientific Software (2015). *SPEC, software for diffraction*, http://www.certif.com/content/spec/.

Condat, L. (2013). *IEEE Signal Process. Lett.* **20**, 1054–1057.

Diamond Light Source (2003). *GDA – generic data acquisition*, http://www.opengda.org.

HDF Group *et al.* (2014). *Hierarchical Data Format*, version 5, http://www.hdfgroup.org/HDF5.

Hunter, J. D. (2007). *Comput. Sci. Eng.* **9**, 90–95.

Jones, E., Oliphant, T., Peterson, P. & others (2001). *SciPy: Open source scientific tools for Python*, http://www.scipy.org.

Klosowski, P., Koennecke, M., Tischler, J. & Osborn, R. (1997). *Physica B*, **241**–**243**, 151–153.

Newville, M. (2014*a*). *PyEpics, Epics Channel Access for Python*, http://cars.uchicago.edu/software/python/pyepics3/

Newville, M. (2014*b*). *LMFIT, Non-Linear Least-Squares Minimization & Curve-Fitting for Python*, http://lmfit.github.io/lmfit-py/.

NumPy Developers (2013). *NumPy*, http://www.numpy.org/index.html.

Toby, B., Sukumar, L., Almer, J. & Jemian, P. (2013). *APSpy: Beam line control scripting with Python and EPICS for APS and others*, http://subversion.xray.aps.anl.gov/admin_bcdaext/APSpy.