ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-WEPHA091

GENERALISING THE HIGH-LEVEL GEOMETRY SYSTEM FOR REFLECTOMETRY INSTRUMENTS AT ISIS

T. Löhnert, A. J. Long, ISIS Neutron and Muon Source, Didcot, UK
J. R. Holt, Tessella, Abingdon, UK

Abstract

At the ISIS Neutron and Muon Source [1], we are currently in the process of rolling out the next generation IBEX control system [2] across all beamlines. One class of beamline we have yet to migrate to the new system is reflectometers. These beamlines require equipment to track the path of the neutron beam to high levels of precision over various experimental configurations, which results in a unique set of requirements for the motion control system. We have implemented a higher level geometry layer responsible for linking the positions of beamline components together so that experimental parameters such as the incident beam angle θ (Theta) are preserved across different beamline configurations. This functionality exists in the legacy system, but needed to be re-implemented for IBEX. The new reflectometry system [3] is written as a Python server running as part of the server on the local instrument machine. This paper provides an overview of the architecture of the new system, specifically how it supports the design goal of making the system easy to extend and reconfigure while preserving the functionality of the existing solution, as well as an outlook on future plans for a more sophisticated motion control system enabled by axis synchronization in real-time.

INTRODUCTION

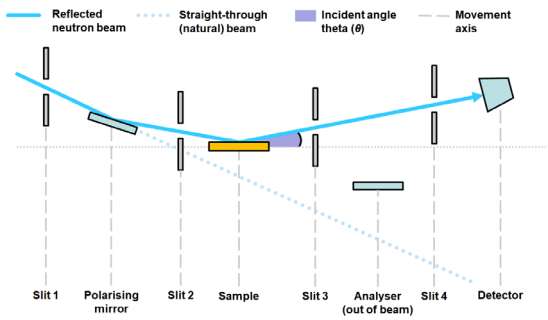

Figure 1: Schematic of a typical reflectometry beamline.

Reflectometers are complex beasts that require the user to keep tight control of a multitude of experimental parameters in order to achieve the intended results. As control system developers, we strive to make this process as easy and intuitive as possible.

In the case of reflectometers, we do this by hiding the complexity of low-level motor control behind a higher level parameter layer. We achieve this by linking the positions of devices on the beamline in physical space so that when one of them (e.g. a polarising mirror) alters the beam path, other devices further along the beam automatically move to track the new path. Currently, the high-level motion control for our 5 reflectometry beamlines is provided by the legacy SECI control system through dedicated LabVIEW [4] VIs. This is currently being replaced by the new IBEX control system based on the open source EPICS toolkit [5]. In order to support reflectometers in the new system, we had to replicate the functionality available in SECI from scratch.

The existing solution under SECI is rather fragmented, having been extended and changed over the course of many years to deal with requirements as and when they arise. Each beamline has its own variation of control interfaces, scripts and workflows with limited documentation. As such, we took this as an opportunity to create a generalised design under IBEX, so that we only have one system that can then be configured for different beamlines in terms of the composition and geometry of the beamline, i.e. which moving parts exist, where on the beamline do they sit, and what is their range of motion. This has a multitude of benefits: We only need to maintain one system, any improvements to the system are available to all beamlines simultaneously, and it helps establish standards and consistency for all ISIS reflectometers.

In the following sections, we will explore the design of the reflectometry server under IBEX, and how it achieves these goals while preserving the workflows the scientists have developed and grown accustomed to over the course of ISIS' many years of operation.

SERVER ARCHITECTURE

Overview

IBEX is implemented in a client-server architecture. Individual devices are controlled by EPICS Input Output Controllers (*IOCs*). These IOCs can be run as part of the server and provide device-specific Process Variables (*PVs*), values which can be read or written to over the network using the EPICS Channel Access protocol.

The Reflectometry Server is implemented in python, using the PCASpy library [6] to expose values and methods in the code via PV addresses. To the outside, the reflectometry server looks and acts like any other EPICS IOC.

The server's main function is transforming the positions of physical motors into higher level parameters that take the geometry of the beamline and the current path of the neutron beam into account. The reflectometry server itself consists of three layers to accomplish this (see Fig. 1). Each contains a list of items, strictly in order from closest to the beam source to furthest from it since any change may affect items further along the beam. The layers are as follows (see Fig. 2 for schematic):

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-WEPHA091

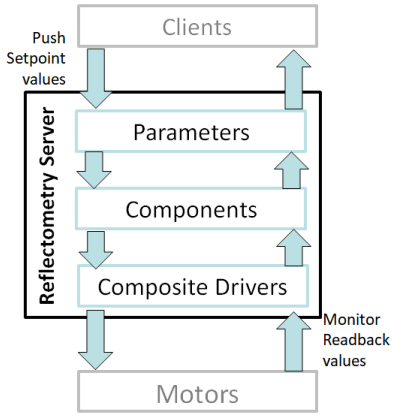

Figure 2: The architecture of the reflectometry server.

Beamline Parameters

Beamline parameters are the values we expose to users as PVs to be accessed via the graphical user interfaces or scripting. Each parameter usually (but not necessarily) relates to one device on the beamline. Parameter values are always relative to the current incoming beam for the given device. Types of parameter include device offset from beam, device angle, or a binary in/out parameter which is derived from the device position and a given threshold value. The incident beam angle θ is a special type of reflection angle, since its value is derived not from a single motor driving an angle, but rather two positions in a shared 2D-coordinate system for the sample and the next enabled component relative to which the sample should be angled.

Geometry Components

The items in this layer, which we call "components", are the building blocks of the beamline geometry model. This layer calculates the beam path, and handles the conversion of positions between parameter values (positions relative to current beam) and motor values (positions relative to straight-through beam). The beamline parameters provide set point values for the motors from above, and the composite driving layers provides the readback positions for the parameters from below.

We use various different types of component depending on the device that is being controlled, which differ in how they track and affect the beam path. For example, it could be tracking the 2D-position only, tracking both position and angle (such that the device stays perpendicular to the beam), or it could be reflecting the beam thus changing the beam path for subsequent components. We delve into more detail on the geometry model in the next section.

Composite Drivers

This layer pushes values into the motor drivers on top of which the reflectometry server sits. It also contains functionality to apply user-defined corrections to the raw motor values for known imprecisions due to engineering limitations, as well as some coordination logic to ensure motors move concurrently such that all moving axes in a given move finish in time with the slowest axis. For now, the purpose of this simple method of synchronization is to avoid crashing individual axes into each other. Future plans for a more sophisticated synchronization approach are described in the last section of this paper.

All of the above layers are unified within a top-level "Beamline" object, which maintains an ordered list of items for each layer and coordinates the interaction between them.

GEOMETRY MODEL

The model of the beamline consists of the list of components that exist in a shared coordinate system. The coordinate system we use is relative to the straight through neutron beam, i.e. the beam as it enters the blockhouse coming from the target without any alterations to its path. For the moment, it is 2 dimensional, with y being the height above the beam, and z being the distance along it (following existing conventions at ISIS).

Each component in the geometry model has a "zero" position, which is the position at which they are centred on the straight-through beam, and a movement strategy (e.g. linear or along an arc) that defines the range of possible positions in (y, z) for this component. Each component forms the relationship between:

- An incoming beam: the beam which will intersect the component's movement described by a position in (y, z) and an angle.
- outgoing beam: the beam after it has interacted with the component. This becomes the incoming beam for the next component in the list
- user set value relative to the beam: where the user would like to position an object relative to the beam (e.g. 1mm above the beam). This is mainly used to move an axis across a range of positions in isolation during beamline alignment
- Motor position: the value of the underlying motor

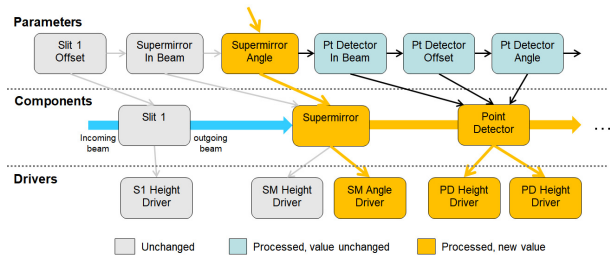

Figure 3: Processing a mirror angle change on move.

The reflectometry server actually maintains two entirely separate beam models, one each for the "should" and "is" state of the beamline. The former is updated every time a user applies a new setpoint value to a top-level parameter (i.e. moves that part of the beamline, see Fig. 3), which then updates the target position in room coordinates at the component level, and recursively triggers the same process on all subsequent components who are subject to the updated beam path. At the end, these new positions are physically actuated by the motor drivers. The latter is updated every time a motor reports a new position, triggered via monitoring processes.

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-WEPHA091

Modes

Beamlines usually have a limited number of standard experimental conditions users want to easily switch between, for example experiments using either polarised or unpolarised neutrons. For the former, the polarising supermirror should automatically move to track the beam and apply an offset for its reflection angle to the subsequent beam path. For the latter, the mirror should be parked outside of the beam and the angle parameter should be ignored when computing the path. To make it easy to switch between the two, users can define beamline "modes", which contain a subset of all parameters in the configuration, i.e. all those which should be processed on a move instruction when this mode is active. They can also contain a set of initialisation values to move the beamline into a given default position when the mode is activated.

CONFIGURATION

The reflectometry server is configured through a file written in Python that is read at start-up. In it, we create instances of the parameters, components and motor drivers that exist for this beamline with appropriate geometry information. This is then collated into a top-level beamline object, which is what is returned to the reflectometry server.

The way the architecture is divided into layers allows us to easily extend the server with different kinds of objects for each that can be arbitrarily combined. Since, like the reflectometry server, the configuration file is written in Python, we (or the users) can even define new classes ad-hoc to support special types of devices, without having to modify and deploy the server itself.

Beamline modes are also defined in the configuration file by passing in a list of active parameters and an optional dictionary of parameter initialisation values, and are then passed into the final beamline object.

One of our goals is that scientists should be able to set up and modify their own beamline configurations. While providing a lot of flexibility, our current approach of using a Python file is prone to user error. At the moment, we are mitigating this by providing a set of helper functions that help ensure the configuration created is valid, specifically that items that are created end up being passed into the correct modes and eventually the top-level beamline object. Still, it is possible to write configurations that are syntactically wrong (e.g. typos), or semantically wrong (e.g. nonsensical component ordering). This is mitigated by the fact that an invalid configuration gets flagged with an error on server start-up. Still, configuring the reflectometry server requires a slight learning curve and is mostly in the domain of the developer for the moment. In the future we may wrap this process in a configuration builder abstraction that has stronger validity checks and automation for building the boilerplate configuration around the desired items.

INTERACTION

Graphical User Interface

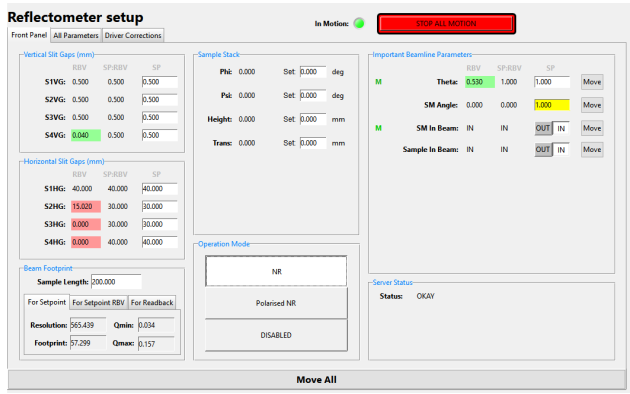

Figure 4: The reflectometry front panel GUI.

The GUI for interaction with the reflectometry server is implemented as a CS Studio [7] Operator Interface (*OPI*), shown in Fig. 4. Since the beamline parameters are accessible as PVs, CS Studio provides all the relevant functionality for interacting with these values.

The interface is loosely modelled after the existing SECI interface. Cornerstone requirements for its design are:

- High information density – ability to see the position and status of all parameters at a glance.
- Recognizing error states – e.g. by highlighting mismatches between target and actual position, or providing a meaningful message for the overall server status
- Input checking – each parameter has an additional setpoint value that is only displayed but not propagated to the motor controller until a "move" instruction has been issued. This allows us to define target values for the entire beamline, visually confirm they are correct, then move all relevant axes simultaneously

Scripting

IBEX provides scripting capabilities via a built-in python library called Genie_Python [8], which contains a command set that replicates that of the OpenGENIE scripting language [9] used in the legacy control system, e.g. for reading/setting control variables, waiting for certain conditions to be met or starting / stopping data collection. In both systems, values are usually manipulated via blocks, which are wrappers around some beamline variable with a user-defined descriptive alias and logging capabilities. In the case of IBEX, blocks are created around EPICS PVs. Since the reflectometry beamline parameters are exposed as PVs via PCASpy, we can create blocks on these values and interact with them via scripting like any other block in IBEX.

In addition, reflectometers at ISIS make use of an OpenGENIE scripting library written by the reflectometry scientists specifically for this type of beamline, which includes commands e.g. to set up the entire beamline for a run at a given angle for θ, or to change the composition of a liquid cell containing the sample. While the scripting commands used by the individual beamlines have a lot of

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-WEPHA091

commonalities, each have their own version which have diverged over time as scientists have made tweaks and additions to their local command set to better accommodate the unique requirements or workflows of their beamline. In IBEX, we have a single script repository that is shared between all beamlines. We have recreated the most common commands and refactored them to maximize shared code between different beamlines. However, discovering the full breadth of individual requirements is an ongoing process and we have yet to take an in-depth look at some of the more complex ISIS reflectometers. As such, the reflectometry scripting library is most likely still subject to change. However, the goal of supporting all reflectometry beamlines through a common scripting library remains the same.

PROGRESS

The new reflectometry server has to undergo rigorous testing to ensure it performs at least as well as the existing system before being greenlit for use in production. One of the challenges we face is that testing time with real neutron data is limited, since all ISIS reflectometers are in regular use. So far, we have configured two different beamlines for the new reflectometry system and have confirmed its behaviour against the legacy system in two rounds of testing. The tests performed include scanning over the abstracted higher level motion axes, and performing a test experiment using a sample with known reflectivity properties. We have confirmed that the beam path tracking capabilities work, and that the new system provides the necessary workflows for aligning a beamline and running a simple neutron reflection experiment, both through its UI and its scripting library. The outstanding work on the server is predominantly in the domain of improving server performance, robustness and usability rather than functionality.

FUTURE PLANS

The way we collect reflectivity data for a given sample is to perform runs at different θ, each shifting the range for the momentum transfer Q for which we record neutron counts, then stitching the datasets together at the end. Since the resolution degrades at the edge of the Q range, observing and stitching the data for more points results in higher quality results. However, the overhead of stopping / starting data collection in between moving the reflectometry beamline to its target positions for a given θ limits for how many different points we can reasonably collect data. What we ideally would like to be able to do is collect position-annotated data while the beamline is performing a steady sweep over a range of θ, which would create “infinite” data points (in practice, the useful maximum is one per accelerator pulse).

θ being a composite axis, we require truly synchronized motion axes in order to do this. Currently, coordinated moves in our system are concurrent rather than synchronous, i.e. there is no guarantee where axes are in relation to each other beyond the initial move instruction. Instead, we want motion axes to continually monitor each other and correct motion in real time if necessary.

The constraint that prevents us from doing this is the motor controllers currently found on most ISIS beamlines do not provide the capabilities for such fast synchronization. However, we expect this to change with a planned roll-out of a new Beckhoff-based [10] motion control system at the ISIS facility. These controllers would allow us to run much of the composite driver level synchronization code on their embedded real-time operating system instead, opening up possibilities to perform the continuous scans described above.

This much more sophisticated method of coordinated motion control is one of the key motivations for the migration to the new IBEX control system in the first place. Although our first target milestone for the reflectometry server is to simply replicate the functionality of the existing SECI system, our work is laying the foundation for such developments.

REFERENCES

[1] ISIS Pulsed Neutron and Muon Source, http://www.isis.stfc.ac.uk

[2] K. V. L. Baker *et al.*, “IBEX: Beamline Control at ISIS Pulsed Neutron and Muon Source”, presented at the 17th Int. Conf. on Accelerator and Large Experimental Control Systems (ICALEPCS'19), New York, NY, USA, Oct. 2019, paper MOCPL01, this conference.

[3] IBEX Reflectometry Server, http://www.github.com/ISISComputingGroup/EP-ICS-inst_servers

[4] National Instruments LabVIEW, http://www.ni.com/labview

[5] EPICS Control System Framework, http://www.epics-controls.org

[6] PCASpy: Portable Channel Access Server in Python http://pypi.python.org/pypi/pcaspy

[7] Control System Studio, http://www.controlsystem-studio.org

[8] Genie_Python, http://www.github.com/ISISComputingGroup/genie_python

[9] S. I. Campbell, F. A. Akeroyd, C. M. Moreton-Smith "Open GENIE – Analysis and Control" http://arXiv:cond-mat/0210442

[10] Beckhoff, http://www.beckhoff.co.uk