

---

# STATE-DEPENDENT FORCES IN COLD QUANTUM GASES

---

Christopher Billington

Submitted in total fulfilment of the requirements  
of the degree of Doctor of Philosophy

**Supervisory committee:**

Prof Kristian Helmerston

Dr Lincoln Turner

Dr Russell Anderson



School of Physics and Astronomy  
Monash University

April, 2018

rev: 145 (c13b5cabdafa)  
author: chrisjbillington  
date: Sun May 20 18:41:56 2018 +1000  
summary: Working on software

This page intentionally left blank

---

# Contents

<b>Contents</b>	<b>i</b>
<b>4 Software for experiment control and analysis</b>	<b>3</b>
4.1 The programs . . . . .	4
4.1.1 labscript . . . . .	4
4.1.2 runmananger . . . . .	5
4.1.3 runviewer . . . . .	5
4.1.4 BLACS . . . . .	6
4.1.5 lyse . . . . .	7
4.2 Design philosophy and advantages of approach . . . . .	8
4.2.1 It's code . . . . .	8
4.2.2 Coupled components and the Unix philosophy . . . . .	10
4.2.3 Off-the-shelf hardware . . . . .	10
4.2.4 Open source Python . . . . .	11
4.3 Recent developments . . . . .	11
4.4 Future developments . . . . .	12
4.5 Regrets, Antiregrets, and friction . . . . .	12
4.6 Reproduced publication: A scripted control system for autonomous hardware-timed experiments . . . . .	12
<b>References</b>	<b>23</b>

rev: 145 (c13b5cabdafa)  
author: chrisjbillington  
date: Sun May 20 18:41:56 2018 +1000  
summary: Working on software

This page intentionally left blank

rev: 145 (c13b5cabdafa)  
author: chrisjbillington  
date: Sun May 20 18:41:56 2018 +1000  
summary: Working on software

## Todo list

This page intentionally left blank

## Software for experiment control and analysis

[TODO: better screenshots: example.py? Monkeypatch to hide errors in BLACS?]

Software underlies a huge part of physicists work, whether experimental or theoretical. On the experimental side in our field, increasingly complex and precise experiments in atomic physics require increasingly sophisticated control of the lasers, magnetic coils, frequency synthesisers, cameras, etc that interact with the quantum systems we study. These devices necessitate an interface, between themselves and the experimentalist, and whilst the interfaces of the past were more likely to be knobs and dials on the front of the device, they are increasingly taking the form of software. Software is needed to convert from a smooth ramp of voltages designed to ramp up a magnetic field slowly into a finite list of voltages and times that a device can output with precise timing to make it happen. Software is needed to transmit this data to the device in question, in the format it requires. Software is required to extract the images and voltage traces from the cameras and voltage acquisition devices and store them in computer memory or on disk. And finally software is required to compute meaningful results from this raw data.

A large fraction of my time during my PhD was spent developing, maintaining and improving the laboratory control system software suite that has emerged from our group at Monash—the *labscript suite*. Originally envisioned as a Python [CITE] library for generating arrays of hardware instructions to be programmed into devices via a LabVIEW VI [CITE? Registered trademark?] in a manner similar to most existing control systems in our field, the software suite grew to encompass most aspects of day-to-day control and analysis in our labs. At present it comprises about five separate programs/libraries, depending on how one chooses to draw the borders between them, that control every aspect of a cold atom physics experiment from setting parameters to analysing results.

An overview of the process is shown in Figure 4.1.

In this (short) chapter, I'll first give a quick overview of each program and what it does. Then I'll outline the design and development approaches we have taken with the *labscript suite* and comment on the effects these choices have had on the course the project has taken over the last few years. Then I'll summarise recent developments since the publication of our paper on the software; *A scripted control system for autonomous hardware-timed experiments* [1], which is reproduced at the end of this chapter. Further details on the role of each program in the suite and the design underlying it are available in the paper, and a more detailed presentation of the software, its design principles, and recent and future developments is available in Philip Starkey's thesis [CITE].

rev: 145 (c13b5cabdafa)  
author: chrisjbillington  
date: Sun May 20 18:41:56 2018 +1000  
summary: Working on software

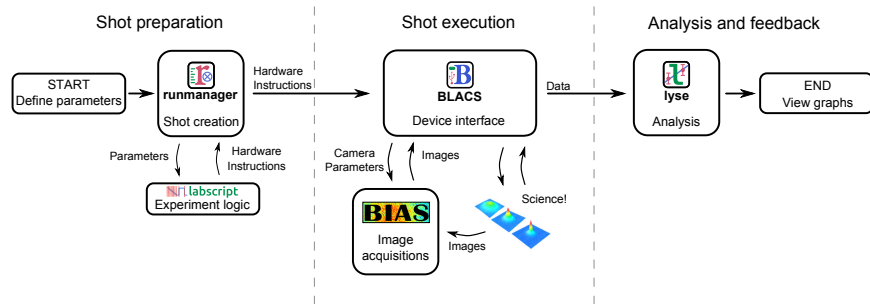


Figure 4.1: The labscript suite comprises a number of libraries and programs allowing one to perform precisely timed experiments, each run of which we call a ‘shot’, using commodity hardware such as devices from SpinCore, NovaTech, National Instruments and others. Experiment logic is described by the user in the form Python code using the **labscript** Python module, which produces from the user’s code a set of low-level instructions appropriate for being programmed into the hardware. The program **runmanager** provides a graphical interface for inputting parameters into this experiment logic, and allows this process to be repeated to produce multiple sets of instructions for repeated execution of the experiment with different input parameters. Not shown in this flowchart is **runviewer**, which is a graphical program displaying plots of the instructions that have been generated by **labscript**. One the instructions have been generated, the program **BLACS** (Better Lab Apparatus Control System) is responsible for communicating with the hardware: programming in the generated instructions, beginning the experiment, and saving any acquired data to file. An auxiliary LabVIEW program called **BIAS** (BEC Image Acquisition System) is used for communication with cameras in Monash quantum fluids group laboratories, though other groups use a number of alternate programs in its place, including a stripped-down derivative of BIAS called **unBIASed**, as well as several Python-based ‘camera servers’. After BLACS is finished with a shot, the data is passed to **lyse**, which executes user-written analysis scripts on each shot to analyse the results, and also executes scripts that operate on sets of data over multiple shots, producing interactive plots. Runviewer provides a Python library for producing shots with programmatically provided parameters, allowing the flowchart to close into a loop and produce shots based on analysis results. This has been used to optimise experiment outcomes with respect to a give figure of merit.

## 4.1 The programs

### 4.1.1 labscript

Labscript isn’t a program but rather a library: that is, it is a set of classes, functions and methods that can be called from user-written code. We call labscript a *compiler*, because what the functions, classes and methods within it do is generate tables of low-level instructions appropriate for programming into devices to execute the experiment described by the user. Thus the user writes a line of code like `MOT_beams.constant(t=3, 1, 'V')` and this will add an entry to the table of instructions for whichever DAC is controlling the MOT beams to go to three volts at  $t = 3$  s after the beginning of the experiment. This is a simple example, but has advantages over having a human write the table directly.<sup>1</sup> After one has told the labscript compiler with this line that the MOT beams should have their control voltage set to three volts, it knows that at all later times the same state should remain, until the user says otherwise. Thus the user doesn’t need to also change all future rows of the table: it is enough to declare a change.

<sup>1</sup>Most existing control systems in our field are more-or-less in the form of a large table with each row corresponding to a particular time in the experiment, with the user editing values in the table directly, or typing mathematical expressions in the table describing values as a function of time between one row of the table and the next.

rev: 145 (c13b5cabdafa)  
author: chrisjbillington  
date: Sun May 20 18:41:56 2018 +1000  
summary: Working on software



Labscript automates much of the tedious, repetitive work that is required in generating those lists of voltages, frequencies, and digital values required to control an experiment. This tedium mostly comes from the fact that devices are sharing timing (see the paper for a description of *pseudoclocking*, the method by which the timing of devices is controlled). When one device changes state, several devices may receive a timing pulse at that time, and so they must have an entry in their corresponding tables in order to output the correct value (possibly the same as the previous value they were outputting) at that time, lest they get too far ahead and output a value that was meant for a later time. Labscript takes high level descriptions of what voltages etc. are required at different times, puts them on a common timing base and generates the correct tables of values. It also collects any other instructions such as camera exposure durations, or the position a translation stage should move to at the start of the experiment even though it is not capable of moving quickly during the experiment. These instructions are processed by labscript and saved to a file in HDF5 format [CITE]. For more information, see the paper at the end of this chapter, and Philip Starkey's thesis [CITE CURRENTLY IN PREPARATION TO BE SUBMITTED].

#### 4.1.2 runmanager

A thirty second or so experiment is not the only timescale on which experimentalists require automation. Ordinarily the same experiment is repeated over a range of input parameters, with several input parameters varying to span some parameter space. In addition there are many parameters involved in an experiment that do not vary often, but nonetheless need to be managed. Runmanager is a graphical program for entering and managing these parameters and describing the parameter spaces over which they should vary. Users can enter simple numbers into the interface (or other non-numerical variables), or lists of numbers that will be considered a description of a dimension of a parameter space. These dimensions may be combined in an outer product resulting in a larger space, or equal length dimensions may be looped over in tandem, if the two variables are intended to vary together rather than separately.

The graphical interface of runmanager is shown in Figure 4.2. Runmanager produces the initial HDF5 files that are passed to the user's labscript code, which the user writes in the form of a Python script that makes use of the labscript library. The user specifies in runmanager's interface which python file contains their 'experiment script'. When the user clicks 'engage', runmanager produces one HDF5 file—each containing a set of globals—for each point in the parameter space described by the globals currently selected in the runmanager interface. For each HDF5 file runmanager initialises the labscript library such that these values become global variables from the perspective of the users experiment script, which then runs. The labscript library then writes the resulting hardware instructions to that same HDF5 file.

We refer to the process of passing the HDF5 file to the user's code and running it as 'compilation', and the resulting HDF5 file containing both globals and hardware instructions a 'compiled shot file'. Compilation occurs in a separate process from the runmanager graphical interface, allowing a clean separation between user code and runmanager, so that even the most low-level crashes of the user's code cannot crash runmanager and only require a restart of its subprocess. This type of separation is a repeated occurrence in the labscript suite and has been invaluable for making robust programs that can keep operating in the case of inevitable crashes of user code, or of bugs within the labscript suite itself.

#### 4.1.3 runviewer

[TODO SCREENSHOT]

```
rev:      145 (c13b5cabdafa)
author:   chrisjbillington
date:     Sun May 20 18:41:56 2018 +1000
summary:  Working on software
```

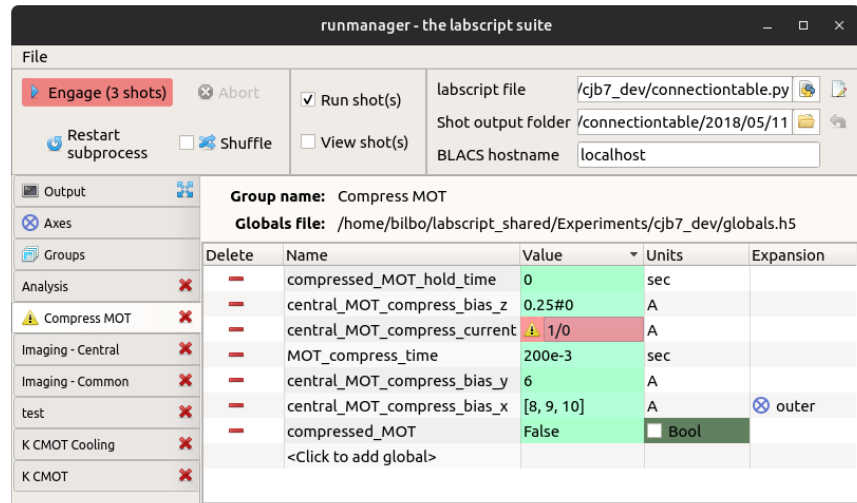


Figure 4.2: Runmanager as of 2018, showing the interface for entering ‘globals’, so called because they appear to the user’s code as global variables. Boolean globals can be turned on and off with a checkbox, and expressions resulting in an error are highlighted in red. The ‘expansion’ column is how one chooses whether a global should be considered a list of values to loop over, re-running the experiment each time, and if so if that loop should be combined with other such globals to loop over the resulting product space (‘outer’) or whether the globals should be looped over together (‘zip’). Zipped globals can be grouped together by typing a name in the expansion column to identify which ‘zip group’ the global belongs to. Globals in the same zip group will loop together, and multiple zip groups will form separate axes of a product space.

Runviewer is a program for viewing the results of **labscript** compilation in the form of graphical plots of the voltages, digital values, frequencies etc that comprise the hardware instructions produced. This is useful for debugging experiment design and timing of instructions, as well as verifying that a newly made **labscript** device class (the ‘driver’ code for each device that converts **labscript**’s intermediate description of hardware instructions into the actual format required for a given device) is functioning as intended.

#### 4.1.4 BLACS

BLACS (Better Lab Apparatus Control System) is a graphical program responsible for queueing experiment shots as compiled by **labscript** and **runmanager**, and executing them one after the other on the hardware. As such, BLACS interacts with a number of python classes that function as *drivers* for each devices, containing code that uses the required software libraries, hardware drivers or communication protocols to communicate with the devices in the manner required by each one individually. BLACS executes the code for communicating with each device in a separate process in order to isolate them from each other, so that communication failures, software bugs or other failures that may occur in the interactions with one device will not stop BLACS from continuing to function in other respects. Errors are presented graphically and each device process may be restarted with the click of a button if something goes wrong. This is useful for both responding to unexpected failure, as well as for debugging during the development of the driver for a new device (or of new features for an existing device) being integrated with the **labscript**

rev: 145 (c13b5cabdafa)  
author: chrisjbillington  
date: Sun May 20 18:41:56 2018 +1000  
summary: Working on software

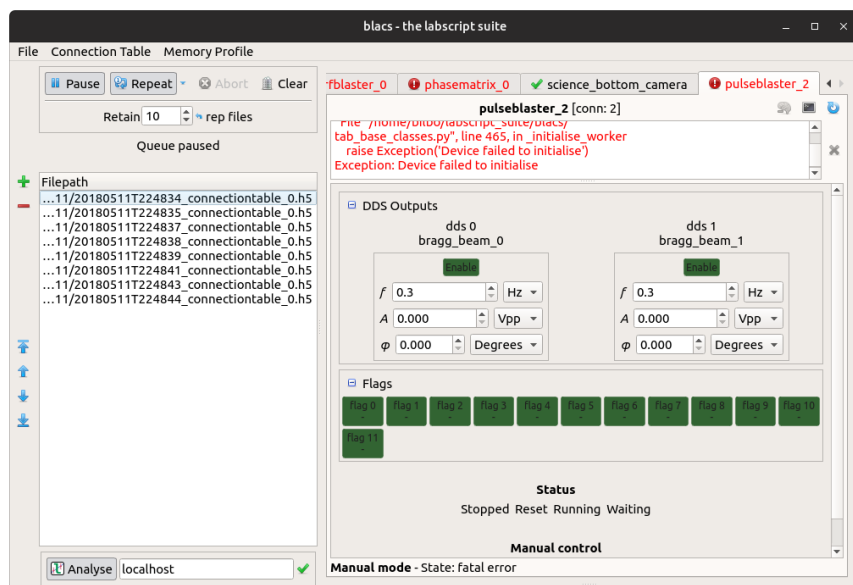


Figure 4.3: BLACS as of 2018

suite.

Upon receiving an HDF5 file from `runmanager`, BLACS adds it to the queue of shots to be executed on the hardware. It then executes these shots in order, by programming the instructions stored in the HDF5 file into each device, and then giving the top level device the command to begin the experiment. Devices are programmed in parallel in their separate processes, saving time.<sup>2</sup> Once the shot is complete, each device process is given the command to write any data acquired to the HDF5 file.

BLACS can also repeat shots, by copying and then ‘cleaning’ an HDF5 file after it has already run to produce a new shot file ready to be run on the hardware. It can repeat either all the shots, or just the last one in the queue. This ability to always keep running by repeating the last shot in the queue is crucial for experiments using alkali metal dispensers (‘getters’) or desorbing metal vapour using UV desorption [CITE, check correct name], as these must run on an approximately fixed duty cycle to maintain the desired atomic vapour pressure. If the experiment stops running, it will have to ‘warm up’ again.

<sup>2</sup>Particularly since many delays in programming the devices are communication delays, during which the process is simply idle.

#### 4.1.5 lyse

Once BLACS is done with the experiment, the shot files are optionally passed on to the analysis program `lyse`. `lyse` is essentially a scheduler for user-provided analysis routines. A list of analysis routines (in the form of Python scripts), called *single-shot routines* are executed in order whenever a new shot is received by `lyse`, with the shot file as input. The analysis routines may read raw data from the HDF5 file, or read analysis results saved by previously-run analysis routines, and save their own results. Analysis routines may also produce plots using the `matplotlib` library—`lyse` detects these and re-uses the same window for subsequent plots so that repeated runs of the analysis routines result in the plot updating in-place, rather than a proliferation of plot windows. Any other plotting library can be used (for example `pyqtgraph` [CITE]), though the user needs to write their code for the updating behaviour in this case.

The shot globals and analysis results for all shots received by `lyse` are maintained in

```
rev:      145 (c13b5cabdafa)
author:   chrisjbillington
date:     Sun May 20 18:41:56 2018 +1000
summary:  Working on software
```

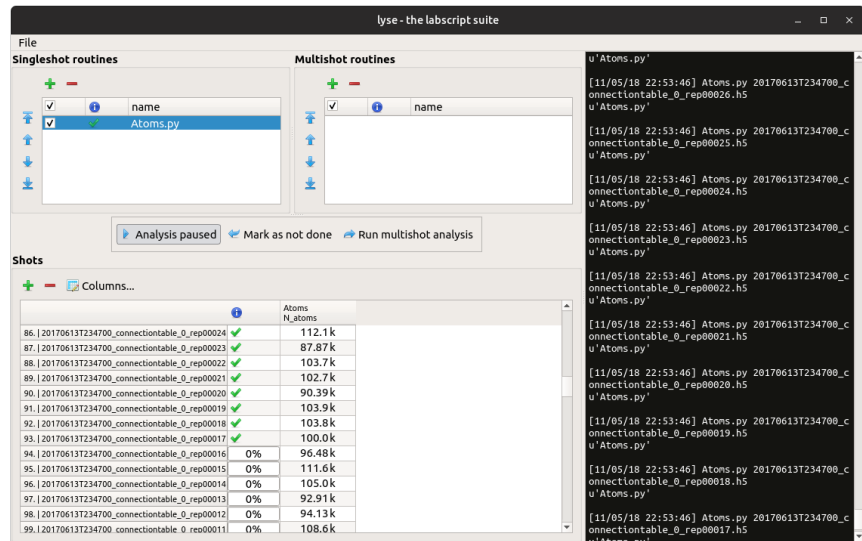


Figure 4.4: Lyse as of 2018

a tabular data structure: a ‘dataframe’ provided by the `pandas` [CITE] Python package, which is browsable in the `lyse` interface. This table of data is available to a further list of analysis routines, called `multi-shot routines`. This list of analysis routines is also run in sequence, but only once single-shot analysis has completed on all shots presently loaded into `lyse`. These routines can be analyses of relations between input parameters and analysis results of the shots, in order to say, measure a trend of the number of atoms in a MOT as the magnetic field gradient was varied. Both single-shot and multi-shot routines can be run from within `lyse`, or externally by running Python manually. In this latter case, the shot file on which to run single shot analysis can be provided as a command line argument, and the dataframe analysed by multi-shot routines will be obtained from a running instance of `lyse` over the network. This is no different to what happens when multi-shot routines are run from within `lyse`: they are simply run by `lyse` with no input, and are expected to call a function `lyse.data()` to obtain the dataframe. Because of the way this is implemented, one can also open an interactive Python interpreter on any computer on the same network, type `import lyse; df = lyse.data(hostname)` (where `hostname` is the hostname of the computer running `lyse`) and begin interactively exploring the data using the `pandas` library, one of its great strengths.

## 4.2 Design philosophy and advantages of approach

### 4.2.1 It’s code

The design of our software brings the process of experimental physics closer to that of software development, making it amenable to many of the tools and processes in use in software development such as version control, bug tracking, textual diffs of what has changed in experiments. Both the experiment logic and the analysis routines comprise Python code that can be examined for changes using standard ‘diff’ tools, stored in version control, or forked into multiple versions using distributed version control and merged back together again. This can allow risky changes to experiment logic or analysis code with the reassurance of being able to roll back to a known working state should the changes not prove fruitful. One exception to this ‘everything is code’ philosophy is the

```

rev:      145 (c13b5cabdafa)
author:   chrisjbillington
date:     Sun May 20 18:41:56 2018 +1000
summary:  Working on software

```



Figure 4.5: Helpful advice from a PhD student of the Spinor BEC lab

globals themselves, they are stored in HDF5 files, though we do have a tool for ‘diffing’ them too to highlight what has changed between two globals files, or between the current set of values configured in `runmanager` as compared to a particular shot file (Figure 4.5).

As in software development, experimental physics re-uses the same procedures time and time again, and needs a way to manage the complexity of turning large parts of functionality on and off. In a traditional atomic physics control system, disabling a part of the experiment logic might involve tedious clicking to remove or disable each instruction involved. In Python, this can be a single `if` statement wrapping a function call containing all the complexity of the part of the experiment being disabled. The ability of high level programming languages to manage complexity via encapsulation and code re-use with functions, classes and modules carries well over to experimental physics. Using an existing programming language saves us, as the developers of a control system from having to re-invent (likely badly) the features of a programming language within our control system. For example, the tedious clicking required to ‘comment out’ part of an experiment in the aforementioned hypothetical ‘traditional’ control system could be avoided if the system implemented a feature for this in particular. But then what about *nested* if statements? To support all the use cases that might come up, one would have to essentially invent a programming language within such a control system. Instead of this, why not use an existing programming language?

The use of an existing programming language only aids the labscrip suite in the case of ‘compile-time’ conditionals and other control statements— that can be evaluated when the hardware instructions are produced by labscrip, as opposed to ‘run-time’ when the experiment is run on the hardware (such as conditionally turning a laser on if a photon is detected on a photo-detector). These ‘run time’ control statements do require special treatment in labscrip, and labscrip currently only has one type of functionality like this built-in. This is the ability to pause the experiment until a pulse is produced that resumes the master pseudoclock. This allows the common use case of synchronisation with the background 50 Hz magnetic noise from mains electricity by pausing the experiment until a fixed point in the 50 Hz cycle to ensure the magnetic field is close to identical from one shot to the next. It also allows servoing of the MOT load, by loading for a variable amount of time based on a threshold fluorescence such that variations in MOT loading efficiency from one shot to the next do not result in variations in actual atom numbers.

Further ‘run-time’ control flow tools such as conditional branches would not be too difficult to incorporate into labscrip in the future, but would require hardware capable of holding multiple alternative sets of instructions in memory and able to switch between them on receipt of a digital pulse on some input. The SpinCore PulseBlaster [R?]<sup>2</sup>—the device used by most users of the labscrip suite to produce clocking pulses—supports this [VERIFY], but most output devices labs use with the labscrip suite do not.

rev: 145 (c13b5cabdafa)  
author: chrisjbillington  
date: Sun May 20 18:41:56 2018 +1000  
summary: Working on software

### 4.2.2 Coupled components and the Unix philosophy

An aspect of the Unix philosophy [CITE] is that tools should ‘do one thing and do it well’. The components of the labscrip suite are not as minimal as they could be, but are nonetheless separate components that talk to each other over network connections.<sup>3</sup> Thus in principle one can remove a component and replace it with another that plays a different role. For example, `runmanager` has been re-purposed by Philip Starkey [thesis in preparation] to generate parameter space scans for numerical simulations instead of experiments, by calling code other than labscrip code. `lyse` is in use by one group [CITE THE GROUP? FRED] for on-line analysis of experiment results in the form of HDF5 files that were produced by Matlab code.

The separate programs of the labscrip suite are also implemented in many cases as separate processes exchanging instructions over network sockets. Whilst these processes are so far running on the same computer, it is possible to extend them to run on separate computers merely by changing the hostname that the network sockets connect to from `'localhost'` to the hostname of some other computer (actually starting the other process on the other computer automatically whilst retaining security is another matter). This makes it not too large a modification to be able to have lyse analysis routines run on a remote computer (perhaps a server with a powerful GPU or many CPU cores for computationally intensive analyses), or to have devices controlled by `BLACS` be connected to a different computer than the one running `BLACS`. This latter configuration can aid in reducing the size of ground loops due to long cables, or allow the simultaneous use of devices that require different operating systems or other conflicting software or hardware configurations preventing them from being used on the same computer.<sup>4</sup>

`runmanager` is a graphical program, but also a software library for setting globals and compiling shots by running labscrip code, and so is in a sense two separate components. As a result, it is possible to write code that produces shots based on something other than the simple parameter space scans `runmanager` is capable of producing. In the past, the labscrip suite contained a program called `mise`, used for performing optimisation of the experiment. `mise` would use the `runmanager` library (but not the graphical interface) to produce shots based on a genetic algorithm and the results of analysis communicated to it by `lyse` analysis routines. This type of optimisation was powerful, but has been superseded, and we no longer maintain `mise`. However a similar workflow is used in the spinor BEC laboratory at Monash to integrate the MLOOP machine-learning optimisation library with the labscrip suite for experiment optimisation.

The separation of components also aids in development of the labscrip suite. Programs can be ported to use updated versions of libraries one at a time, and tested separately, enabling a more flexible process of development.

### 4.2.3 Off-the-shelf hardware

The labscrip suite is the software part of an experiment control system, and does not mandate any particular hardware. Whilst certain hardware drivers are developed and maintained by myself and the other labscrip suite developers, if one wants to use different hardware, one can write drivers for it and use it with the labscrip suite. This does mean that the software cannot make hard assumptions about the hardware and has to deal with a wider range of possibilities, which is a complicating factor in maintaining the code, but I believe this approach is the better one compared to having a limited range of hardware designed specifically for the labscrip suite. It is hard to predict what hardware people will need for atomic physics experiments, and our efforts are better spent making software than hardware. Labscrip developers and users have made some hardware of their own as well however, and there exists a low-cost pseudoclock and a digital out device both based on a sub-\$100 microcontroller board [CITE SOMETHING? Phil,

<sup>3</sup>This is in violation of another part of the Unix philosophy that says programs should exchange text streams: our programs mostly exchange messages over network sockets, containing filenames pointing to HDF5 files on a network drive.

<sup>4</sup>A pertinent example is the bandwidth of a computer’s USB2 bus being the limiting factor in the speed at which one lab can run experiments: using two computers to program two USB devices could halve the programming time between shots



me, Martijn?]. However these have proved difficult to maintain in the face of changing software development kits for the microcontrollers, and I suspect FPGAs are a better solution. Rory Spiers [CITE thesis? Or say recently graduated Melbourne and now at JQI?] has developed a low-cost FPGA-based pseudoclock [not yet published] that may be an attractive alternative to the PulseBlaster.

#### 4.2.4 Open source Python

open source and using open source libraries: advantage: patching upstream is a viable option. I have submitted and had patches accepted by numpy, pandas, h5py and pyzmq that improved their usability for specific use cases we have had.

crowdsource bugfixes! Lots of competent programmers in physics groups. They implement long-standing features that we have not had time to implement ourselves.

Popular, dynamic programming language Allows us to hack on it to introduce custom behaviour whilst keeping the benefits of using a general purpose programming language. Allows new students to get up to speed quickly. Allows interoperability with a wide range of tools and protocols for talking to other code and to hardware.

Standard format: HDF5

### 4.3 Recent developments

Firstly, far more people using the software.

Since the paper:

General: A port to the Qt GUI toolkit, which occurred while I was in the US visiting Ian. This toolkit is more widely used and cross-platform. A proper installer such that installation is now running one script and works reliably across platforms. An (almost complete) port to Python 3. Miso is no longer used, instead people are plugging lyse and the runamanager API into things like MLOOP. This demonstrates the pluggability of the system, but could be improved. General polishing all round. More devices. Output capturing of subprocesses now captures output of subsequent subprocesses and extension code. More flexible camera interfaces, there are several Python camera servers. We no longer treat cameras as special, and if they need to be on a different computer for reasons of hardware or screen real estate, the remote devices feature will get that. A port to the Qt GUI toolkit.

Delete shots functionality - lyse can deal with it and BLACS has a plugin to do it.

Labscript: Arbitrary function ramps. Markers. Inverted DO. Gated clocks: allow more devices to run off the same clock given very different memory capacities.

lkyse: copy to clipboard.

BLACS: Repeat functionality. Repeat last. Delete repeated shots. Plugin mechanism. Optional. Terminals visible for all devices for debugging purposes. Can be started without a Connection table. Bugfixes leading to faster times in between shots.

runmanager: Saving and loading of configuration settings. Finer control over parameter spaces: per-axis shuffle, control over axis loop order. Agnostic compilation: can be used to make parameter space scans for arbitrary things.

runviewer: Saving and loading of configuration settings. Nonlinear time (using markers) don't reverse engineer data? Save extra data? Maybe, maybe not.

lyse: Copy figures to clipboard. Saving and loading of configuration settings. Can work with deleted shots. Performance enhancements.

devices: Supporting more devices and more modes of operation. Generic NI DAQmx. More PulseBlaster models. More modes of operation for the Novatech.

rev: 145 (c13b5cabdafa)  
author: chrisjbillington  
date: Sun May 20 18:41:56 2018 +1000  
summary: Working on software

## 4.4 Future developments

Analog input widgets for BLACS

- Plugin tabs

- Remote control of runmanager - makes it easier to implement optimisers and other feedback

- Compile queue: just-in-time compilation so that feedback can be on a per-shot basis, so that repeated shots can take into account updated variables immediately

- Integrate remaining features from another fork (spielman fork): postprocessing scripts for per-shot feedback, fixed-duration shots.

- labscript rewrite for more precise and nondestructive control of timing

- remote processes so devices can be connected to different computers

- Progress bar in BLACS (can be coloured by markers)

- Analysis globals

## 4.5 Regrets, Antiregrets, and friction

GTK. Bad call Labscript processing should be integer based and nondestructive. Bad call, yet to fix

- Friction: pandas

- Antiregrets ZMQ and the zprocess module Qt Python 2 threading like crazy

## 4.6 Reproduced publication: A scripted control system for autonomous hardware-timed experiments

```
rev:      145 (c13b5cabdafa)
author:   chrisjbillington
date:     Sun May 20 18:41:56 2018 +1000
summary:  Working on software
```



# A scripted control system for autonomous hardware-timed experiments

P. T. Starkey,<sup>a),b)</sup> C. J. Billington,<sup>a)</sup> S. P. Johnstone, M. Jasperse, K. Helmerson,  
 L. D. Turner, and R. P. Anderson

*School of Physics, Monash University, Victoria 3800, Australia*

(Received 11 April 2013; accepted 17 July 2013; published online 8 August 2013)

We present the *labscript suite*, an open-source experiment control system for automating shot-based experiments and their analysis. Experiments are composed as Python code, which is used to produce low-level hardware instructions. They are queued up and executed on the hardware in real time, synchronized by a pseudoclock. Experiment parameters are manipulated graphically, and analysis routines are run as new data are acquired. With this system, we can easily automate exploration of parameter spaces, including closed-loop optimization. © 2013 AIP Publishing LLC. [<http://dx.doi.org/10.1063/1.4817213>]

## I. INTRODUCTION

Modern experiments in quantum science demand flexible, autonomous control of heterogeneous hardware. Many such experiments are *shot*-based: a single experiment shot comprises analog, digital, and radiofrequency (rf) outputs operating under precise timing, as well as synchronized camera exposures and voltage measurements. Bose–Einstein condensation (BEC) experiments,<sup>1</sup> for example, require a timing resolution down to a few hundred nanoseconds, and may last for up to a minute. Output must, therefore, be hardware timed, requiring devices be programmed with instructions in advance of an experiment shot. Most measurements of interest require numerous shots, to build up statistics, or to observe the response of the system to varying parameters. Such repetition is common to experiments employing cold quantum gases or trapped ions for precision metrology,<sup>2</sup> quantum computation,<sup>3</sup> and quantum simulation.<sup>4</sup>

Individual shots are typically complex, requiring the coordination of many devices. This coordination is the role of a *control system*. A good control system should automate the programming of devices based on a high-level description of the experiment logic.<sup>5</sup> It should handle the repetition of shots and automated variation of experiment parameters, the increasingly complex demands of which cannot be rapidly, robustly, and continuously met by human operators. It should automate analysis, leading to the prospect of closed-loop control: the results of analysis influencing subsequent experiment shots. Applications of such closed-loop control include autonomous algorithmic optimization of parameters, and automatic recalibration in response to environmental drifts.

Most existing control systems take one of the two approaches for providing a human interface to programming hardware. One is text-based, in which experiments are written using a general purpose programming language.<sup>6</sup> In the other, experiments are instead described graphically using a custom user interface.<sup>7–11</sup> The text-based approach natively offers the

advantages of a programming language, particularly control-flow tools such as conditional statements, loops, and functions. Its disadvantage is that frequently varied settings and parameters may be hidden in hundreds of lines of code. Conversely, the graphical-user-interface (GUI) approach makes experiment parameters more accessible to the user, but features providing for complex experiment logic must be anticipated and implemented specifically.<sup>10,11</sup>

The two approaches need not be mutually exclusive: by separating experiment parameters from experiment logic, parameters can be manipulated graphically and logic textually.<sup>12,13</sup> We contend that by using a high-level programming language with appropriate hardware abstraction, text-based control can be more comprehensible to a newcomer than an equivalent graphical representation of hardware instructions.

We present the *labscript suite* which utilizes a hybrid text-and-GUI approach for control and builds on previous work by addressing the need for autonomous control, analysis, and optimization. Hardware control is abstracted, providing an identical software interface to devices of a common type. Graphical interfaces are dynamically generated based on the current hardware set in use. Analysis is an integral part of the control system, with user-written analysis routines run automatically on new data. Finally, analysis results can modify subsequent experiment shots, closing the feedback loop on analysis and control.

## II. AN OVERVIEW OF THE LABSCRIPT SUITE

The *labscript suite* comprises several programs, each performing one main function; the flow of data between programs is shown in Fig. 1. Each experiment shot is associated with a single file: each program writes to and reads from this file as required before passing it on to the next program. Programs may be run on separate computers, communicating over the network using the ZeroMQ messaging library,<sup>14</sup> exchanging references to the experiment file.

We use the Hierarchical Data Format (HDF version 5)<sup>15</sup> which provides cross-platform storage of large scientific datasets. Exploiting the extensibility of HDF, each file

<sup>a)</sup>P. T. Starkey and C. J. Billington contributed equally to this work.

<sup>b)</sup>Author to whom correspondence should be addressed. Electronic mail: [philip.starkey@monash.edu](mailto:philip.starkey@monash.edu)

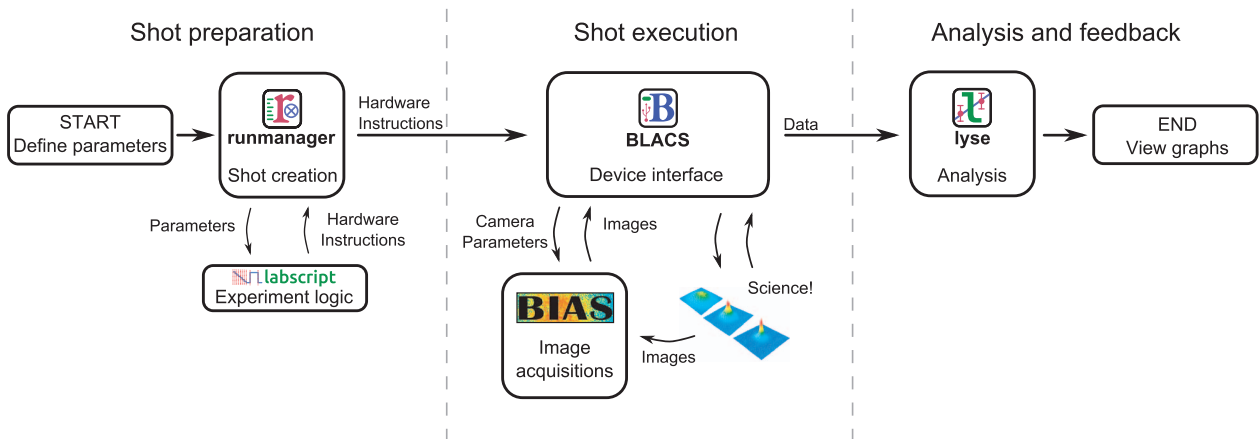


FIG. 1. Each experiment shot comprises three stages: preparation, execution, and analysis. Arrows indicate how the HDF file for an experiment shot passes between software components of the labscript suite. Only the shot execution stage is coupled to hardware timing, allowing new shots to be created and queued while others are running. Similarly, analysis can be performed on executed shots at any time.

is a complete description of the experiment shot. The HDF file begins its life containing only experiment parameters. As it is passed between components of the labscript suite, the file grows to contain the hardware instructions, acquired data, and analysis results. Metadata is also stored including user-written scripts and version control information. This maintains a comprehensive record of the experiment shot for post-hoc analysis, reproducibility, and publication preparation.

Attempts to standardize laboratory device programming have largely failed, with only a minority of devices conforming to standards such as SCPI (Standard Commands for Programmable Instruments).<sup>16</sup> This calls for abstraction to shield the user from low-level interaction. We have created a software library for Python,<sup>17,18</sup> `labscript` (Sec. IV), which provides a common interface for commanding output and measurements from devices. The user writes the experiment logic in Python, and `labscript` generates the required hardware instructions, including a clocking signal for timing (Sec. III).

The `labscript` suite separates experiment logic (written in Python) from experiment parameters, which are manipulated in a GUI. The GUI, `runmanager` (Sec. V), creates the HDF file for the experiment shot and stores the parameters within. If a parameter is a list of values, rather than a single value, `runmanager` creates an HDF file (a prospective shot) for each value. If lists are entered for more than one parameter, `runmanager` creates a file for each point in the resulting parameter space.

For each shot, `labscript` inserts the parameters from the HDF file into the experiment logic, compiles hardware instructions for each device, and writes them to the same file. `runmanager` sends the compiled HDF files to `BLACS` (Sec. VI) which places them in a queue. `BLACS` interfaces with hardware devices either directly, or via secondary control programs such as `BIAS` (Sec. VII). `BLACS` programs the hardware and triggers the experiment shot to begin. The experiment then proceeds under hardware-timed control.

Once the experiment shot has finished, acquired data such as voltage time-series and images are added to the HDF file.

`BLACS` then passes the file to a dedicated analysis system, `lyse` (Sec. VIII). `lyse` coordinates the execution of analysis routines, which are Python scripts written by the user. These scripts may analyze individual shots or a sequence of shots as a whole. This facilitates autonomous analysis of results from parameter space scans, as experiment shots are completed.

The `labscript` software library can be applied to automatically generate shots based on the results of analysis. We have used this to implement a closed-loop optimization system, `mise` (Sec. IX).

### III. PSEUDOCLOCK

A typical BEC experiment requires precise timing over a large range of time scales.<sup>1</sup> There are periods during which magnetic fields or laser intensities, for example, may change with sub-microsecond resolution. Conversely, there are periods during which no devices change their output for several seconds, e.g., loading a magneto-optical trap (MOT). To ensure accurate output during the rapid changes, hardware devices must be preloaded with a set of instructions that can be stepped through by a clock once the experiment begins. Stepping through instructions at a constant rate requires repetitive instructions during the more inactive periods. As many devices only support a limited number of instructions, a constant-rate clock limits the maximum sample rate. A common solution<sup>8,9,11,12</sup> is a variable frequency master clock, or *pseudoclock*, which steps through instructions only when a clocked device needs to update an output (see Fig. 2). This removes the need for redundant instructions.

All devices sharing a pseudoclock must have an instruction when any one of their outputs changes value. This can lead to redundant instructions if only some of the devices are changing at a given time. The instruction limitations of one device may then limit another, e.g., some devices hold only a few thousand instructions in their internal memory, whereas others are limited only by the RAM of the host computer refilling their buffers. To solve this problem, we employ

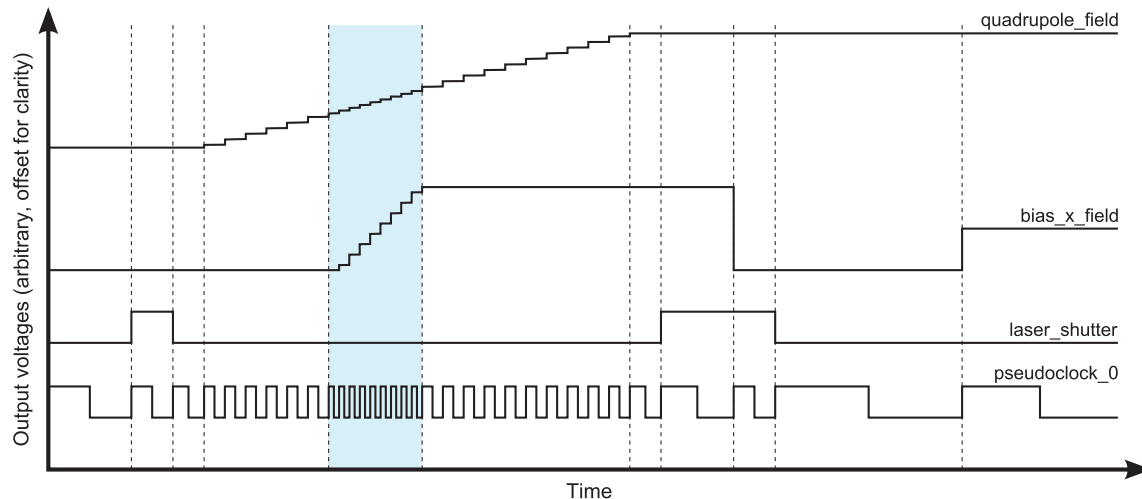


FIG. 2. An example of digital and analog voltage outputs generated by the labscript code in Fig. 3. The pseudoclock (lower trace) ticks when a digital output must change, or at the requested sample rate for time-varying analog outputs (upper two traces). Dashed vertical lines indicate a change in the pseudoclock frequency. When multiple analog outputs are varying at the same time (shaded region), the pseudoclock ticks at the highest of their sampling rates.

multiple pseudoclocks, assigning devices of similar memory limitations to the same clock. At the beginning of a shot, the software starts one pseudoclock (the *master clock*), which then triggers other clocks.

To be a useful pseudoclock, a device must be able to deterministically generate arbitrary digital signals, be hardware triggerable, and hold enough instructions for the required experiment. We currently use two pseudoclocks: the Spin-Core PulseBlaster DDS-II-300-AWG, a commercial device based on a field-programmable gate array (FPGA); and the PineBlaster, a device developed in house based on a microcontroller. Both devices are externally referenced to a stable 10 MHz source.

The PineBlaster is a low-cost device using commodity hardware, based on the Arduino-like Digilent ChipKIT Max32 microcontroller prototyping board.<sup>19</sup> The board is flashed with a program that accepts clock instructions over universal serial bus (USB) and executes them with deterministic timing. It is capable of clocking at up to 10 MHz (100 ns between rising edges) with a resolution of 25 ns. The PineBlaster needs one instruction for each change in clock rate (see Fig. 2) and supports up to 15 000 instructions.

labscript provides support for adding new pseudoclocks. It uses an intermediate format for storing timing instructions; implementing a new pseudoclock entails translating them into the required format for the hardware.

Some experiments require the time between instructions to be determined *during* a shot. This can be achieved by pausing the pseudoclocks until some condition is met. A common example<sup>11–13</sup> is waiting for a sufficient level of fluorescence from a loading MOT. Both the PulseBlaster and the PineBlaster support *wait instructions*, which pause output until resumed by a trigger. These instructions, when used in tandem with devices such as voltage comparators, can command the experiment to wait for events of interest.

#### IV. THE LABSCRIPT LIBRARY

We have created a Python software library, *labscript*, for defining experiment logic. *labscript* provides *hardware abstraction*, a common interface to control heterogeneous hardware. For example, the `DigitalOut` class provides `go_high(t)` and `go_low(t)` functions to set the state of a digital output at time  $t$ . The user calls these functions without regard to the underlying device, its method of programming, or the state of other digital outputs connected to the same device. Based on an experiment script containing such function calls, *labscript* automatically generates instructions for output and measurement devices as well as pseudoclocks. The automatic generation of pseudoclock instructions saves the user from dividing overlapping function ramps into segments (Fig. 2), or manually interpolating output values when a new time point is created on another channel.

An experiment script consists of two parts: a *connection table* (Fig. 3(a)), and code defining the logic of the experiment (Fig. 3(b)). The connection table provides a complete description of devices that are required for the experiment and how they are connected. *labscript* creates a set of Python objects based on the connection table, each with associated functions for commanding output or measurement from devices. The logic of the experiment is then defined by calling these functions with parameters such as time and output value.

As the experiment script is executable Python code, the user has full access to standard Python control flow tools, as well as standard and third party Python libraries. Using a high level language such as Python spares the user from low-level tasks such as memory management.<sup>5</sup> User-created functions can be stored in modules and imported into other experiment scripts. This allows complex experiments to be constructed from simple components, while maintaining comprehensibility, resulting in a gentler learning curve for new students. For example, one might define a `make_BE_C()` function which

```

(a) # Device definitions
PulseBlaster(name='pseudoclock_0', board_number=0)
NI_PCIE_6363(name='ni_card_0', parent_device=pseudoclock_0, clock_type='fast clock',
             MAX_name='ni_pcie_6363_0', clock_terminal='/ni_pcie_6363_0/PFI0')

# Channel definitions
Shutter (name='laser_shutter', parent_device=ni_card_0, connection='port0/line13')
AnalogOut(name='quadrupole_field', parent_device=ni_card_0, connection='ao0')
AnalogOut(name='bias_x_field', parent_device=ni_card_0, connection='ao1')

(b) # Experiment logic
start()
t = 0

# first laser pulse at t = 1 second
t += 1; laser_shutter.open(t)
t += 0.5; laser_shutter.close(t)
t += 0.4;

t += quadrupole_field.ramp(t, duration=5, initial=0, final=3, samplerate=4) # samplerate in Hz
# start ramping the bias field 3 seconds before the quadrupole ramp ends
bias_x_field.ramp(t-3, duration=1, initial=0, final=2.731, samplerate=8)
# t is now 6.9s, the end of the quadrupole field ramp

# second laser pulse
t += 0.4; laser_shutter.open(t)
t += 1; bias_x_field.constant(t, value=0.0)
t += 0.5; laser_shutter.close(t)
t += 2

# hold bias field at bias_x_final_field for 2 seconds before finishing shot
bias_x_field.constant(t, value=bias_x_final_field)
t += 2
stop(t)

```

FIG. 3. An example labscript file. The connection table (a) defines a pseudoclock and a multifunction DAC object and configures three output channels. This is followed by the experiment logic (b) which commands output from these channels by name at times specified by the variable  $t$ . The experiment logic refers to the parameter  $\text{bias\_x\_final\_field}$  which is set in runmanager (Sec. V).

contains the logic to form a Bose–Einstein condensate. While students might not fully understand the experiment logic to create a BEC, they can focus on subsequent experiment logic after a BEC is made. We have found that text based experiment scripts benefit not just from code re-use but also version control, bug tracking, and comparison of incremental changes (diffs).

When the experiment script is run and a timing sequence created, the labscript functions take into account hardware limitations and provide error messages if these are exceeded. If no errors are found, the hardware instruction set for all devices in the connection table is written to the HDF file.

While a text-based definition of experiment logic gives a broad overview of the timing sequence, it is not ideal for visualizing the device outputs to ensure the experiment logic is as intended. The hardware instructions generated by running experiment scripts are difficult to interpret (indeed, labscript was created to mitigate this very problem). Our program (runviewer) produces plots (similar to Fig. 2) of the hardware instructions generated by labscript, allowing quick diagnosis of the timing sequence before reaching for the oscilloscope.

## V. SETTING PARAMETERS—RUNMANAGER

Repeating experiments while varying parameters is a fundamental part of the scientific method. Anyone who has performed a quantum science experiment will be familiar with

tweaking parameters to find a resonance, calibrating a measurement, or acquiring a large amount of scientific data prior to publication. The logic of the experiment does not change every time a parameter is adjusted, and it is cumbersome to edit numbers in a text file for each modification.

To ameliorate this, labscript experiments can take a series of parameters as input. The names and values of these parameters are defined in the graphical interface of runmanager (Fig. 4). The values can be any valid Python expression (such as  $0.74$ ,  $1\text{E}-3$ ,  $\sin(\pi/2)$ , or  $\text{True}$ ) and can refer to each other. We call these parameters *globals* because they are available as global variables in experiment scripts, where they are simply referred to by name. For example, these globals might be used to specify the duration of a  $\pi$ -pulse, the delay between releasing atoms from a trap and imaging them, or the field strengths of bias magnetic coils. This provides a clean separation between code, which defines the nature of the experiment (such as creating a BEC with a vortex or performing a matter-wave mixing experiment), and parameters that modify individual shots.

The user may enter a list of values for a global, such as  $[1, 2, 3]$ , or  $\text{linspace}(0, 10, 100)$ . In this case runmanager produces a corresponding list of experiment shots: one for each value. If multiple globals are entered as lists, runmanager performs a Cartesian product, creating one shot for each point in the resulting parameter space. Two or more lists can be *zipped*, in which case runmanager iterates over these lists in lock-step when producing shots.

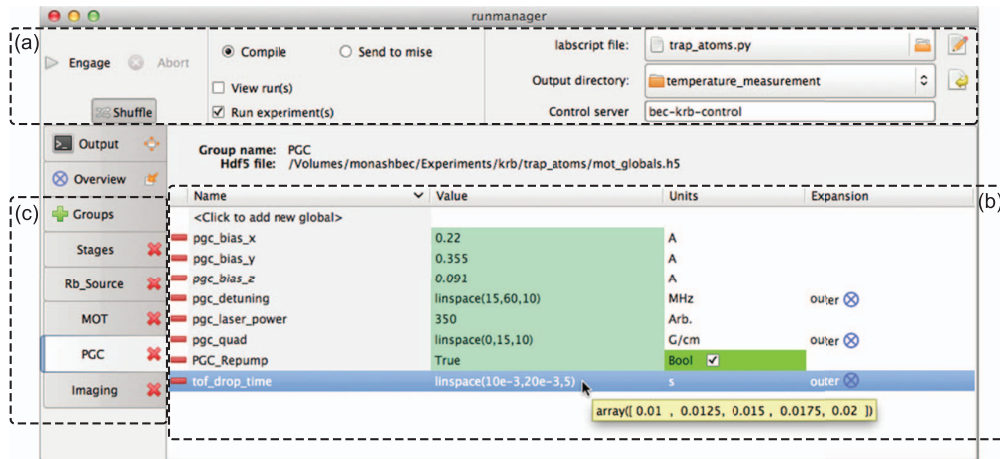


FIG. 4. The runmanager interface for configuring experiment parameters. (a) The experiment logic is specified by the labscript file (here `trap_atoms.py`). HDF files for experiment shots created by runmanager are saved in the output directory. (b) The value of experiment parameters (“globals”) are specified by Python expressions and may have units. These can be single values (i.e., 350 or `True`), lists, or expressions creating lists (as shown for the globals `pgc_detuning`, `pgc_quad`, and `tof_drop_time`). A tooltip shows the evaluation of the global. The “Expansion” column specifies how lists of values are combined to construct a parameter space. (c) Globals can be separated into groups for convenience.

Specifying globals as lists makes it possible to explore complicated parameter spaces containing hundreds or thousands of shots. For example, one might investigate how the temperature of laser cooled atoms varies with laser detuning and magnetic field gradient. Taking the Cartesian product of ten field strengths and ten detunings results in a parameter space of one hundred points. Thermometry at each point in this parameter space commonly requires multiple shots to characterize the expansion rate of the atom cloud. A five-shot temperature measurement brings the number of shots to five hundred. Producing these shots amounts to entering three lists in runmanager and clicking on the “Engage” button, as shown in Fig. 4. runmanager then creates five hundred HDF files containing the globals for each shot. The experiment script is run for each shot, storing hardware instructions in each file.<sup>20</sup> The HDF files are then submitted to BLACS for execution.

## VI. EXPERIMENT EXECUTION—BLACS

BLACS coordinates input and output through hardware devices. These devices can be local, and thus under the direct control of BLACS, or connected to a different computer as part of a secondary control program such as BIAS (Sec. VII). BLACS provides both manual control of devices (through a GUI) and buffered execution of experiment shots.

The GUI for manual control is dynamically generated from a *lab connection table* that describes the current configuration of all connected devices. Each device is allocated a tab in the interface, containing controls for commanding output when in manual control mode (Fig. 5).

Upon submission to BLACS, HDF files containing hardware instructions are checked for validity and placed in a queue. The queue can be reordered, paused, or put on repeat. The validity check compares the connection table of each shot to the lab connection table, rejecting those with incompatible

hardware. This prevents unintended device output that would produce nonsensical results and possibly damage equipment.

BLACS takes the first experiment in the queue, coordinates hardware programming, and sends a start trigger to the master pseudoclock. The experiment then proceeds under hardware timing. At the end of a shot, BLACS coordinates saving data acquired by devices to the HDF file, and returns to manual control mode. Each GUI control is updated to the final values of the shot, maintaining output continuity.

Laboratories are a hostile environment for hardware interface libraries. Power cycling of devices and unplugging of cables are common occurrences. A student tripping over a USB cable (health and safety implications notwithstanding) might be expected to cause an experiment to fail, however the control system ought to recover gracefully when it is plugged back in. Similarly, bugs in closed source drivers and libraries are points of failure outside of a users control.

To make our system robust against such hardware and software failures, BLACS implements a multiprocess architecture similar to the sandboxed tabs of the Google Chrome web browser.<sup>21</sup> For each device in BLACS, a *worker process* is spawned, which communicates with the hardware device. This makes BLACS robust against crashes: if one device has a problem it will not affect others. If a hardware device becomes unresponsive, or the device driver encounters a serious error, its isolation in a separate process prevents the GUI and other devices from suffering the same fate.

Should a worker process crash, the user is presented with the option of restarting the process, which will reload any device libraries it uses. It is worth noting that systems implemented in LabVIEW cannot force libraries to reload, so errors leading to an undefined state would only be remedied by restarting the entire control system.

The initialization of hardware in preparation for a shot is an important part of an experiment, and can significantly contribute to the experiment cycle time. The multiprocess



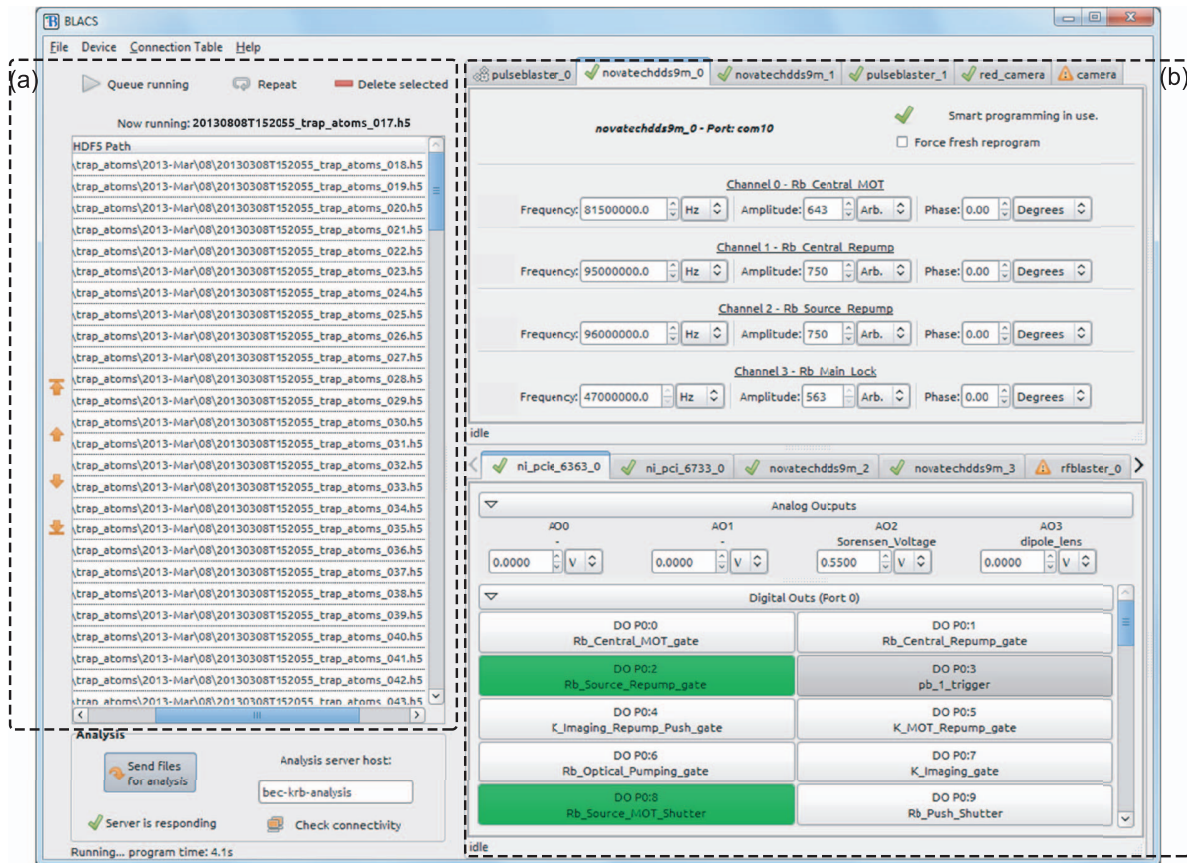


FIG. 5. The BLACS interface for controlling hardware. (a) The queue of shots submitted via runmanager. (b) The manual control interface. Each tab controls one device. Controls for all outputs are automatically generated and are named based on the BLACS connection table.

architecture naturally provides for simultaneous programming of hardware devices, resulting in an increased experiment duty cycle. We have implemented a *smart programming* feature on many of our devices, further decreasing programming time, reprogramming them only if their instructions have changed since the previous shot (on a per-instruction basis when possible). Devices with large buffers and slow communication (such as the Novatech DDS9m rf synthesizer) benefit greatly from this technique.

## VII. IMAGE ACQUISITION—BIAS

Using secondary control programs to communicate with specific devices is desirable when software to do so exists and has been debugged, particularly software written in another programming language. BLACS integrates such programs into the control flow by sending them HDF files containing hardware instructions to program devices for execution upon a hardware trigger. BLACS notifies secondary control programs that the shot has completed, at which point they write any acquired data to the HDF file.

Our camera control and image acquisition system, BIAS, is one such program. BIAS is a LabVIEW application that operates scientific cameras, captures image sequences, and performs image processing tasks such as background subtraction, saturation correction, optical depth calculation, and simple 2D fitting.

Multiple instances of BIAS can be run simultaneously to control multiple cameras in one experiment. BIAS can also run as a stand-alone program for quick visualization of previously captured data or acquire images manually. Hardware communication in BIAS is abstracted through LabVIEW's object hierarchy, allowing a camera class to be written for any vendor library.

LabVIEW provides convenient components for creating graphical interfaces, and BIAS displays raw and computed images as they become available (Fig. 6). Fit results such as atom cloud shape and atom number are prominently displayed to detect and diagnose problems as they occur. The camera acquisition area and regions of interest used to inform fits can be interactively adjusted, without needing to interrupt or recreate a currently running sequence of shots. Multiple regions of interest can be selected and their coordinates saved to the HDF file, enabling further analysis.

## VIII. ANALYSIS—LYSE

Analysis is a critical part of an autonomous control system. Automated analysis—performed immediately after every shot—is often restricted to routines that change infrequently and are applied uniformly once per shot. Ideally analysis should be flexible as well as autonomous; these can be conflicting goals without a unifying analysis framework.

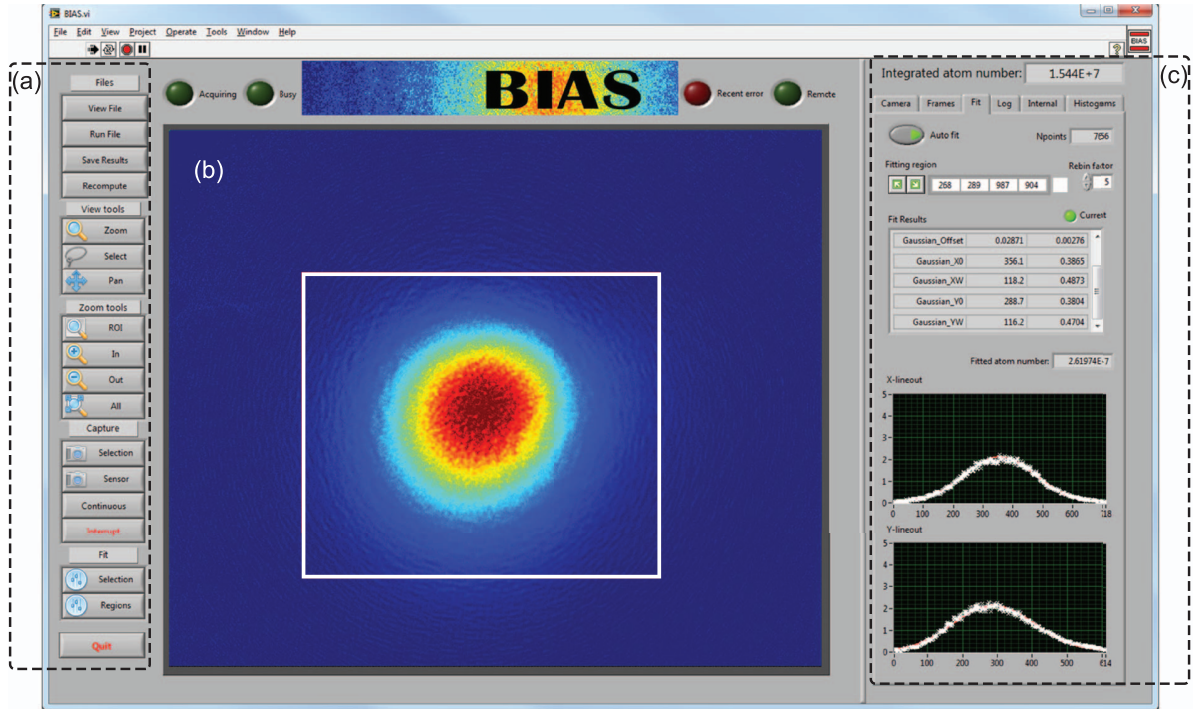


FIG. 6. The BIAS interface displaying a laser cooled atom cloud. (a) Manual controls for loading and capturing images, selecting regions of interest and zooming. (b) Computed optical depth (OD) image of the atoms, with a region of interest (white) selected for fitting. Multiple regions of interest may be selected for multi-component atom clouds. (c) Atom number and cloud size are displayed for immediate feedback.

Our analysis system *lyse* accommodates collective analysis of a group of shots and trivial re-analysis upon changing or adding routines.

*lyse* is a scheduler for user-written analysis routines, which are ordinary Python scripts. It provides functions for extracting the experiment data and metadata from the HDF files and saving analysis results to these files. Multiple analysis routines added to *lyse* execute one after the other when a new HDF file is received over the network, or on command through the GUI. Plots produced by the user's code are updated following every shot as new data comes in from the experiment.

There are two types of routine that *lyse* can run: single-shot, which are run on every shot, and multi-shot, which analyze a group of shots together. Analysis of the thermometry example in Sec. V is shown in Fig. 7. A single-shot routine computes the size of an atom cloud after a fixed expansion time, and a multi-shot routine uses these results to determine the expansion rate and thus the temperature. The multi-shot routine then plots this temperature as a function of laser detuning and magnetic field strength.

Splitting, sorting, plotting, and exploring large multidimensional datasets are cumbersome when directly accessing a set of files. In addition to direct access to the HDF files, *lyse* provides a tabular data structure—a pandas<sup>22</sup> DataFrame—for multi-shot routines, containing all globals as set by run-manager, and all single-shot analysis results. With pandas and the standard Python scientific stack of numpy,<sup>23</sup> scipy,<sup>24</sup> and matplotlib,<sup>25</sup> *lyse* provides a powerful environment for analysis.<sup>26</sup>

Analysis routines can be run independently of *lyse* if desired. This allows the same framework and analysis code to be used for publication preparation.

## IX. OPTIMIZATION—MISE

Marrying powerful Python tools to shot-based analysis permits extensibility of the control system, such as closed loop optimization of measured quantities. One often performs parameter space scans for optimization, requiring many shots. This may be tuning a parameter of an apparatus to enhance its performance, finding a resonance of some transition, or some other feature of interest. The quantity being optimized is often the result of some analysis, e.g., the temperature of ultracold atoms (mentioned in Secs. V and VIII). We have created *mise*, a program that performs automatic optimization of analysis results using a genetic algorithm.<sup>27</sup> A user specifies one or more parameters to optimize against a predefined figure of merit. Genetic algorithms are resistant to noise, making them particularly useful for optimizing experimental results.

The data flow of the optimization process follows Fig. 8, modifying that shown in Fig. 1. The user specifies in run-manager one or more parameters to optimize, with upper and lower limits for each. An analysis routine in *lyse* reports optimality to *mise*, which creates shots with modified parameters and submits them to BLACS.

For each parameter being optimized the user also specifies a *mutation rate*. This determines how much the parameter is varied per generation of the genetic algorithm:

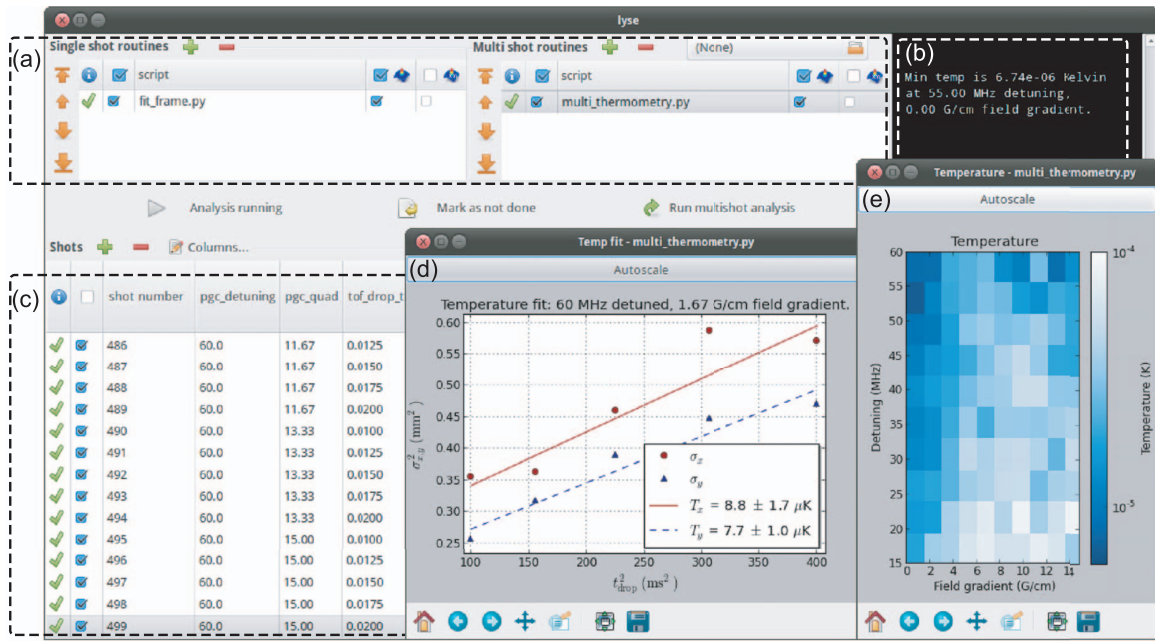


FIG. 7. The *lyse* interface. (a) Routines can be selected to analyze single or multiple shots. (b) Terminal output from the analysis routines in (a). (c) Table of shots; columns show globals and analysis results. A small subset of columns is displayed here. (d) A fit yielding the temperature of laser cooled atoms prepared at a particular field gradient and detuning. (e) The results of the analysis in (d) repeated at each point in the parameter space.

the larger the mutation rate, the faster *mise* will move towards the optimum. However, a large mutation rate limits the precision to which the optimal parameters can be determined.

With this specification of parameters, *mise* creates a population of *individuals*. Each individual comprises values from one point in the optimization parameter space, initially chosen at random. An individual may be a single experiment shot, or—when optimizing the result of a multi-shot analysis—a sequence of shots. Once the shots comprising an individual have executed, the user's analysis routine computes a *fitness*, which may be derived from any measured quantity. *mise* uses the reported fitness in the genetic algorithm to optimize the specified parameters. The genetic algorithm used by *mise*<sup>28</sup> is a variation on pointed directed mutation,<sup>29</sup> in

which mutations are biased in directions previously shown to be successful.

The user can specify when to stop the optimization, either by manual intervention or by a convergence condition written into their analysis script. They may also “guide” the evolution by adding and deleting individuals from the gene pool at any time.

An example of automated optimization using *mise* is shown in Fig. 9. By preferentially exploring the more interesting regions of parameter space, autonomous optimization allows optima to be found in fewer shots.

*mise* uses the *labscript* software library to create HDF shot files and submit them to BLACS. Additional user-written components could similarly submit shots to BLACS if more complex programmatic generation of shots is required.

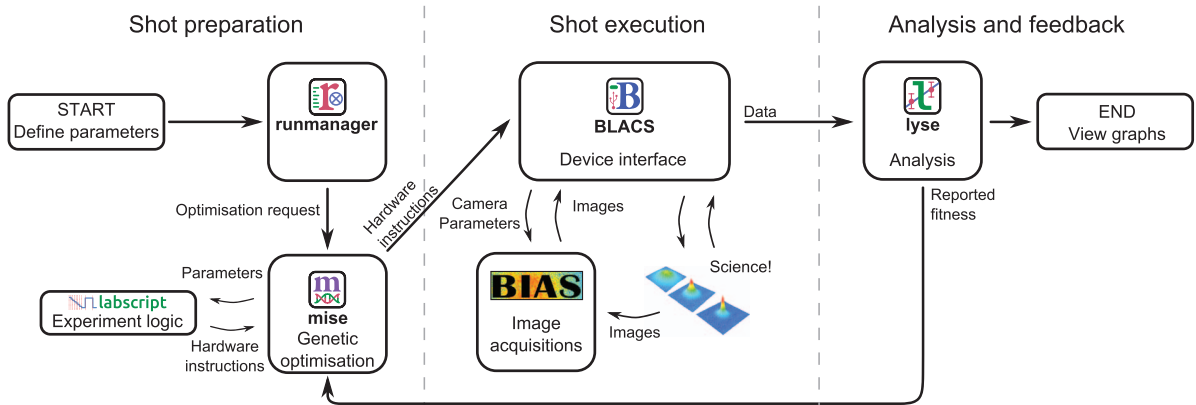


FIG. 8. The data flow for closed loop optimization. In contrast to Fig. 1, analysis results are used to determine future shots automatically. The optimizer *mise* varies parameters, directly calling *labscript* to compile new experiment shots. Parameters to be optimized are selected by the user in *runmanager*. *lyse* reports fitness to *mise* which is used to create the next generation of shots.



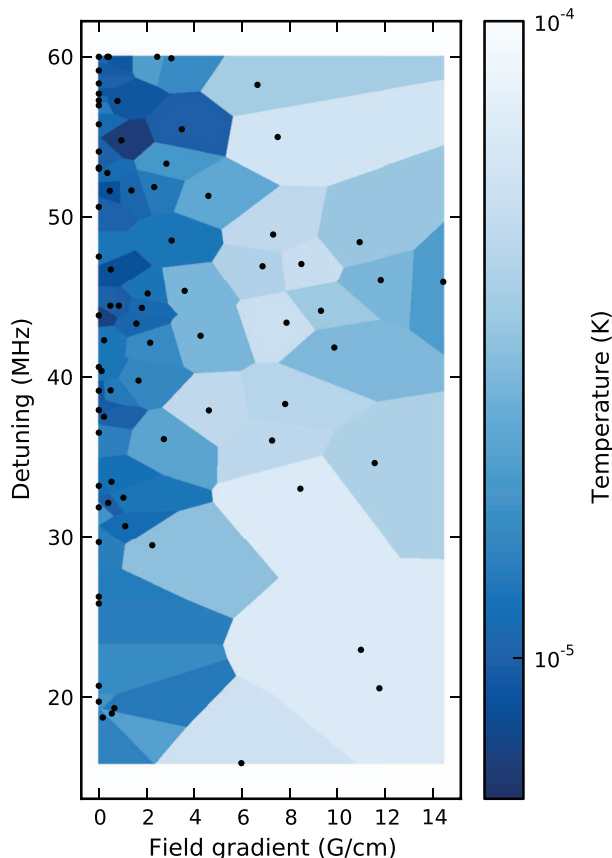


FIG. 9. A proof-of-principle optimization using *mise*. *mise* scanned the parameter space described in Sec. V, searching for the coldest point. Each black point represents a temperature measurement at a specific field gradient and detuning, with the surrounding shading indicating the temperature. Eighty points were taken, corresponding to 400 shots. The colder region of parameter space is sampled more densely than the uniformly-sampled scan, shown in Fig. 7(e), with 500 shots.

## X. PORTABILITY AND EXTENSIBILITY

Our software runs on Windows, Linux, and OS X, although BLACS and BIAS compatibility is subject to the availability of appropriate hardware drivers. If particular devices must be interfaced with a specific computer, operating system, or programming language, a secondary control program (such as BIAS, Sec. VII) can be used. The components of the *labscript* suite communicate with each other via data in HDF files, and over the network with ZeroMQ sockets. The widespread support of these technologies across many platforms<sup>30</sup> ensures users are not bound to any one operating system or programming language. The modular nature of our system allows users to replace or supplement any of our programs in their choice of language.

The programs themselves are also written with extensibility in mind. Adding new hardware support to the *labscript* suite entails writing a new device class for *labscript*, and a GUI tab for BLACS,<sup>31</sup> or a camera class for BIAS. Adding analysis routines to *lyse* amounts to writing a Python script to process experiment data. Existing library functions and base classes assist such development. The suite has already proved useful in a setting distinct from quantum science ex-

periments, automating the prototyping of an objective lens, in which the image of a pinhole was acquired and analyzed at 3600 points in a plane to determine the field of view.<sup>32</sup>

The *labscript* suite is open-source and freely available online.<sup>33</sup> We encourage readers to contact us if they are interested in implementing the suite in their laboratory.

## ACKNOWLEDGMENTS

The authors would like to thank the current users of our system, who were not part of the development team, L. Bennie, M. Egorov, and A. Wood for their input into making this a better system. This work was supported by Australian Research Council Grant Nos. DP1094399 and DP1096830.

- <sup>1</sup>See, e.g., M. Weidemüller and C. Zimmermann, *Cold Atoms and Molecules* (Wiley, 2009), and references within.
- <sup>2</sup>See, e.g., N. Robins, P. Altin, J. Debs, and J. Close, “Atom lasers: Production, properties and prospects for precision inertial measurement,” *Phys. Rep.* **529**, 265 (2013); A. D. Cronin, J. Schmiedmayer, and D. E. Pritchard, *Rev. Mod. Phys.* **81**, 1051 (2009), and references within.
- <sup>3</sup>See, e.g., A. Negretti, P. Treutlein, and T. Calarco, *Quantum Inf. Process.* **10**, 721 (2011); T. D. Ladd, F. Jelezko, R. Laflamme, Y. Nakamura, C. Monroe, and J. L. O’Brien, *Nature (London)* **464**, 45 (2010), and references within.
- <sup>4</sup>See, e.g., I. Bloch, J. Dalibard, and S. Nascimbène, *Nat. Phys.* **8**, 267 (2012); R. Blatt and C. F. Roos, *ibid.* **8**, 277 (2012), and references within.
- <sup>5</sup>G. Varoquaux, *Comput. Sci. Eng.* **10**, 55 (2008).
- <sup>6</sup>P. E. Gaskell, J. J. Thorn, S. Alba, and D. A. Steck, *Rev. Sci. Instrum.* **80**, 115103 (2009).
- <sup>7</sup>R. P. Anderson, Ph.D. thesis, Swinburne University of Technology, 2010.
- <sup>8</sup>M. Beeler, Ph.D. thesis, University of Maryland, 2011.
- <sup>9</sup>P. A. Altin, Ph.D. thesis, Australian National University, 2012.
- <sup>10</sup>T. Stöferle, Ph.D. thesis, Swiss Federal Institute of Technology, 2005.
- <sup>11</sup>A. Keshet and W. Ketterle, *Rev. Sci. Instrum.* **84**, 015105 (2013).
- <sup>12</sup>S. F. Owen and D. S. Hall, *Rev. Sci. Instrum.* **75**, 259 (2004).
- <sup>13</sup>T. Meyrath and F. Schreck, “A laboratory control system for cold atom experiments,” see <http://www.strontiumbec.com/indexControl.html> (2012).
- <sup>14</sup>P. Hintjens, *Code Connected Volume 1: Learning ZeroMQ* (CreateSpace Independent Publishing Platform, 2013); see also “ØMQ: the intelligent transport layer,” <http://www.zeromq.org/>.
- <sup>15</sup>The HDF Group. Hierarchical data format version 5, 2000-2010. <http://www.hdfgroup.org/HDF5>.
- <sup>16</sup>IEEE Standard Codes, Formats, Protocols, and Common Commands for Use With IEEE Std 488.1-1987, IEEE Standard Digital Interface for Programmable Instrumentation,” IEEE Std 488.2-1992.
- <sup>17</sup>G. van Rossum *et al.*, “Python programming language v2.7,” see <http://docs.python.org/2.7/> (2010).
- <sup>18</sup>J. M. Hughes, *Real World Instrumentation with Python: Automated Data Acquisition and Control Systems* (O’Reilly Media, Inc., 2010).
- <sup>19</sup>The source code for turning a ChipKIT Max32 into a PineBlaster is available at <http://hardware.labscriptsuite.org/>.
- <sup>20</sup>In our lab, a typical hardware set for running this experiment would be a SpinCore PulseBlaster DDS-II-300-AWG as a pseudoclock, along with a Novatech DDS9m, National Instruments PCIe6363 and PCI6733 boards and a Photonfocus MV1-D1312(I) camera.
- <sup>21</sup>A. Barth, C. Jackson, C. Reis, and The Google Chrome Team, “The security architecture of the chromium browser,” Technical Report, Stanford Security Laboratory, 2008, available at <http://seclab.stanford.edu/websec/chromium>. See <http://youtu.be/29e0CtgXZSI> for more information.
- <sup>22</sup>W. McKinney, “pandas: a Python data analysis library,” see <http://pandas.pydata.org/>.
- <sup>23</sup>T. Oliphant, “NumPy: numerical Python,” see <http://www.numpy.org/>.
- <sup>24</sup>E. Jones, T. Oliphant, P. Peterson *et al.*, “SciPy: open source scientific tools for Python,” see <http://www.scipy.org/> (2001).
- <sup>25</sup>J. Hunter, *Comput. Sci. Eng.* **9**, 90 (2007); see also “matplotlib: Python plotting,” <http://matplotlib.org/>.
- <sup>26</sup>W. McKinney, *Python for Data Analysis* (O’Reilly Media, Inc., 2012).
- <sup>27</sup>T. Bäck and H.-P. Schwefel, *Evol. Comput.* **1**, 1–23 (1993).

<sup>28</sup>See supplementary material at <http://dx.doi.org/10.1063/1.4817213> for implementation details of the genetic algorithm used by `mise`.

<sup>29</sup>A. Berry and P. Vamplew, "PoD Can Mutate: A Simple Dynamic Directed Mutation Approach for Genetic Algorithms," in *ALSAT2004: International Conference on Artificial Intelligence in Science and Technology*, 21–25 November 2004, Hobart, Tasmania, Australia.

<sup>30</sup>HDF bindings include C/C++, MATLAB, Python, LabVIEW and Mathematica. ZeroMQ support includes C/C++, Python, LabVIEW, Java and many more. See [http://www.hdfgroup.org/products/hdf5\\_tools/](http://www.hdfgroup.org/products/hdf5_tools/) and [http://www.zeromq.org/bindings:\\_start/](http://www.zeromq.org/bindings:_start/) for more complete lists.

<sup>31</sup>BLACS communicates with hardware devices through user-written interface code. Devices communicating over standard buses (RS232, USB, Ethernet) are easily interfaced using standard Python libraries for these buses. Devices with proprietary interfaces can be programmed by calls to vendor-supplied libraries through Python's sophisticated foreign-function interface.

<sup>32</sup>L. M. Bennie, P. T. Starkey, M. Jasperse, C. J. Billington, R. P. Anderson, and L. D. Turner, *Opt. Express* **21**, 9011 (2013).

<sup>33</sup>"The labscrip suite: an open source experiment control and analysis system," see <http://labscripsuite.org/>.

---

## References

- [1] P. T. Starkey, C. J. Billington, S. P. Johnstone, M. Jasperse, K. Helmerson, L. D. Turner, and R. P. Anderson. *A scripted control system for autonomous hardware-timed experiments*. Review of Scientific Instruments **84**, 085111 (2013). DOI: [10.1063/1.4817213](https://doi.org/10.1063/1.4817213).  
[p 3]

rev: 145 (c13b5cabdafa)  
author: chrisjbillington  
date: Sun May 20 18:41:56 2018 +1000  
summary: Working on software

This page intentionally left blank

---

## Word count

### Total

Words in text: 57764

Words in headers: 474

Words outside text (captions, etc.): 9738

Number of headers: 125

Number of floats/tables/figures: 48

Number of math inlines: 1962

Number of math displayed: 321

Files: 8

### Subcounts:

text+headers+captions (#headers/#floats/#inlines/#displayed)  
7422+77+1300 (23/3/559/73) File(s) total: atomic\_physics.tex  
50+0+0 (0/0/0/0) File(s) total: front\_matter.tex  
16769+117+3154 (32/14/478/107) File(s) total: hidden\_variables.tex  
1607+8+98 (3/0/12/0) File(s) total: introduction.tex  
23088+155+3403 (31/15/824/137) File(s) total: numerics.tex  
4043+50+617 (16/5/3/0) File(s) total: software.tex  
4001+32+781 (11/7/79/4) File(s) total: velocimetry.tex  
784+35+385 (9/4/7/0) File(s) total: wave\_mixing.tex

rev: 145 (c13b5cabdafa)  
author: chrisjbillington  
date: Sun May 20 18:41:56 2018 +1000  
summary: Working on software