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 Helmerson
Dr Lincoln Turner
Dr Russell Anderson

School of Physics and Astronomy
Monash University

April, 2018

```
rev:     156 (128ef198499e)
author:  chrisjbillington
date:    Sat Jun 02 17:54:23 2018 +1000
summary: Added history and attribution section
```

This page intentionally left blank

Contents


```
rev:     156 (128ef198499e)
author:  chrisjbillington
date:    Sat Jun 02 17:54:23 2018 +1000
summary: Added history and attribution section
```

This page intentionally left blank

```
rev:     156 (128ef198499e)
author:  chrisjbillington
date:    Sat Jun 02 17:54:23 2018 +1000
summary: Added history and attribution section
```

Chapter 4

Software for experiment control and analysis

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 being studied. Use of these devices necessitates some kind of interface between the experimentalist and each device, and whilst 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, using a communications protocol and data format compatible with the device. Software is required to extract the images and voltage traces from 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.

Part of my time during my PhD was spent developing, maintaining and improving the laboratory control system software suite that has emerged from the Quantum Fluids group at Monash: the *labscript suite*. Originally envisioned as a Python [1] library for generating arrays of hardware instructions to be programmed into a specific, fixed set of hardware devices via a LabVIEW [2] Virtual Instrument (VI), 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.

The types of experiments the labscript suite addresses are 'shot-based' experiments—ones in which precise timing is required over hardware during some interval of time while the experiment is performed (a 'shot'), after which the hardware is inactive until the next shot. Many repetitions of similar shots are often performed to build up measurement statistics or investigate the response of a system to a change in some parameter. This general method of performing experiments is common to many experiments in cold quantum gases and trapped ions [3, 4], quantum computation [5, 6] and quantum simulation [7, 8].

In this 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

```
rev:     156 (128ef198499e)
author:  chrisjbillington
date:    Sat Jun 02 17:54:23 2018 +1000
summary: Added history and attribution section
```

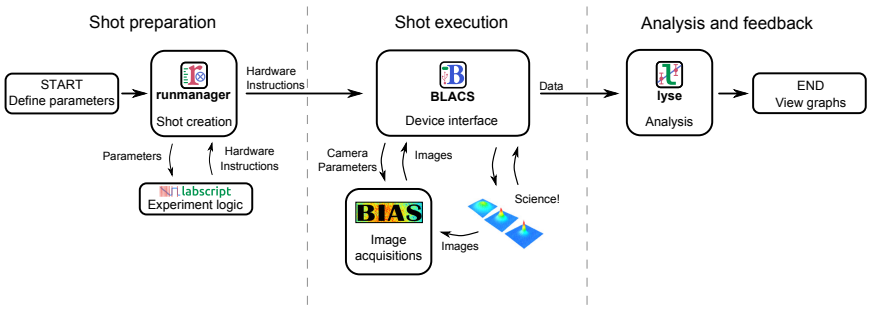


Figure 4.1: The labscript suite comprises a number of libraries and programs allowing one to perform precisely timed experiments, each run of which we call a 'shot', using commodity hardware such as devices from SpinCore, NovaTech, National Instruments and others. Experiment logic is described by the 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`. Once 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 other 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. In addition to providing a graphical interface for setting parameters, `runmanager` provides a Python library for producing shots with programatically provided parameters, allowing the flowchart to close into a loop and produce shots based on analysis results. This can be used to optimise experiment outcomes with respect to a give figure of merit. Figure reused with permission from Starkey et al. [9], © American Institute of Physics 2013.

of our paper on the software; *A scripted control system for autonomous hardware-timed experiments* [9], which is reproduced at the end of this chapter. I will also discuss planned and in-progress developments. Further details on the role of each program in the suite and the design underlying it are available in the paper, and a more thorough presentation of the software, its design principles, comparisons with other laboratory control software, and recent and future developments are available in Philip Starkey's thesis [10].

The labscript suite has been adopted by world-leading research groups at the National Institute of Standards and Technology, the University of Maryland, National Research Laboratories, US Army Research Laboratory, Stanford University, JILA, the University of Rochester, Dartmouth College, Universität Tübingen, Bates College, Universität Basel, and Technische Universität Darmstadt. It continues to grow as a collaborative open-source software project benefiting the experimental physics community.

```
rev:     156 (128ef198499e)
author:  chrisjbillington
date:    Sat Jun 02 17:54:23 2018 +1000
summary: Added history and attribution section
```

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 digital-to-analogue Converter (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.[1] 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 it enough to declare a change.

[1] 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.

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 the Hierarchical Data Format, version 5 [11] (HDF5). This format is a convenient, standardised, cross-platform, and self-documenting format with widespread adoption across many disciplines, and compatibility with a wide range of programming and analysis environments, providing a high degree of interoperability between the data files produced by the labscript suite and other software tools.

4.1.2 runmanager

A thirty second or so experiment (a typical duration for a BEC experiment, though ion trapping experiments are often much shorter) is not the only timescale on which experimentalists require automation. Commonly, the same brief experiment is repeated over a range of input parameters, with several input parameters varying to span some parameter space. In addition to these varying parameters, there are many parameters involved in an experiment that do not vary often, but nonetheless need to be managed. runmanager is a program providing a graphical user interface (GUI) for entering and managing these parameters and describing the parameter spaces over which they vary. Users can enter simple numbers or expressions (including expressions for non-numerical variables) into the interface, or lists of numbers that can optionally 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 GUI of runmanager is shown in Figure 4.2. runmanager produces the initial HDF5 files that are passed to the user's 'experiment script', i.e. their Python script describing the experiment logic using the labscript library. The user specifies in runmanager's in-

```
rev:     156 (128ef198499e)
author:  chrisjbillington
date:    Sat Jun 02 17:54:23 2018 +1000
summary: Added history and attribution section
```

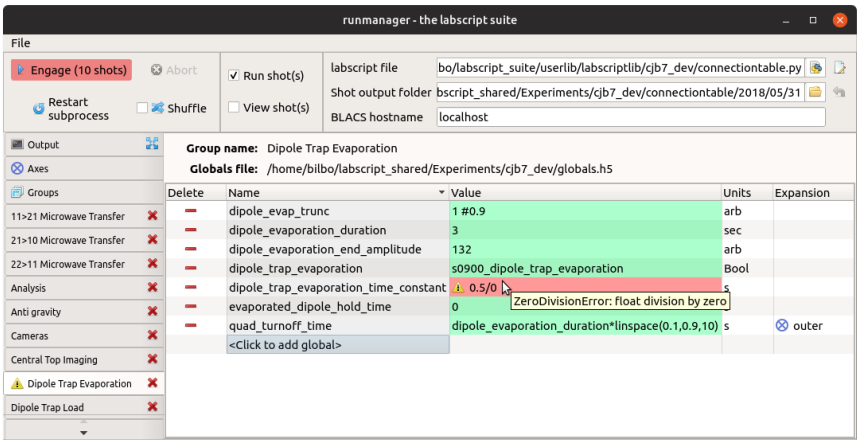


Figure 4.2: `runmanager` as of 2018, showing the interface for entering 'globals', so called because they appear to the user's code as global variables. Boolean globals can be turned on and off with a checkbox, and expressions resulting in an error are highlighted in red. The 'expansion' column is where the user specifies 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. Globals can be entered as Python expressions, including expressions containing the names of other globals.

terface which Python file contains this experiment script, and when they click the 'engage' button, `runmanager` produces one HDF5 file—each containing a set of parameters—for each point in the parameter space described by those parameters currently active 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 user's experiment script, which then runs. For this reason we call the parameters 'globals'. After the user's instructions are processed, the `labscript` library writes the resulting low-level 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 theme in the labscript suite and has been invaluable for making robust programs that can continue to operate in the case of inevitable crashes of user code, or of bugs within labscript suite or third-party code.

4.1.3 `runviewer`

`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'

```
rev:     156 (128ef198499e)
author:  chrisjbillington
date:    Sat Jun 02 17:54:23 2018 +1000
summary: Added history and attribution section
```

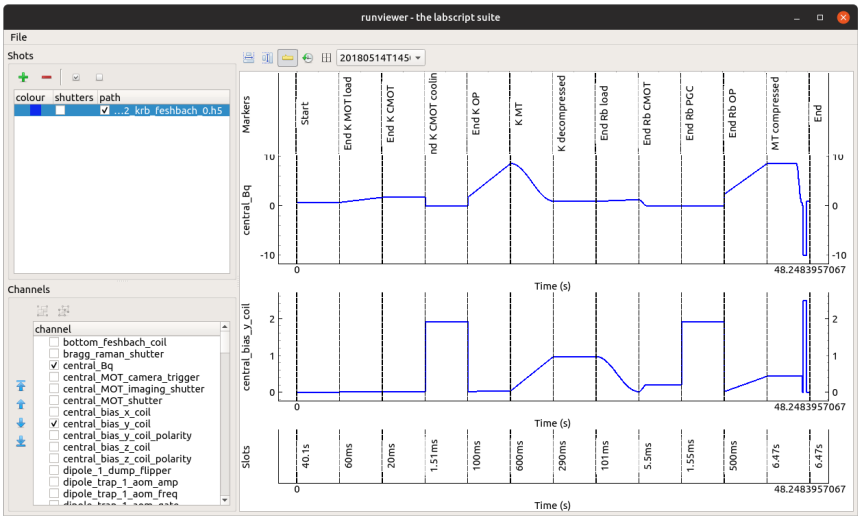


Figure 4.3: The interface of `runviewer` as of 2018, showing the recently-added 'nonlinear time' feature, in which different stages of the experiment—as declared with `labscript` code—can be meaningfully shown on the same time axis despite occurring on very different timescales.

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. The GUI of `runviewer` is shown in Figure 4.3.

4.1.4 BLACS

BLACS (Better Lab Apparatus Control System) is a graphical program responsible for queueing experiment shots as compiled by `labscript` in tandem with `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, re-initialising communication with the offending device. This is useful both for 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 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.[2] 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

[2] Particularly since many delays in programming the devices are communication delays, during which the process is simply idle.

```
rev:     156 (128ef198499e)
author:  chrisjbillington
date:    Sat Jun 02 17:54:23 2018 +1000
summary: Added history and attribution section
```

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 ultraviolet light-induced atom desorption [12], as these processes must run on an approximately fixed duty cycle to maintain the desired atomic vapour pressure, otherwise experiments need to 'warm up' after being idle to reach adequate pressure.

When processing of the shot queue is paused by the user or there are no shots for `BLACS` to run, `BLACS` remains in 'manual mode', in which the outputs of the devices can be controlled in real-time by the user. The graphical interface of `BLACS` (shown in Figure 4.4) presents controls for the outputs of all devices, with each device and output channel labelled with its name as specified in a 'connection table' file containing `labscript` code describing the connection hierarchy of all devices.

4.1.5 `lyse`

Once `BLACS` has finished with a shot, the shot file is 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 provided as input to each script. 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. Analyis routines may also produce plots using the `matplotlib` library [13]—`lyse` detects these and reuses 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` [14]), though in this case the auto-updating behaviour is not provided automatically by `lyse`.

The shot globals and analysis results for all shots received by `lyse` are maintained in a tabular data structure—a 'dataframe' provided by the `pandas` [15] 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 can 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 containing multi-shot data. Because of the way this is implemented, one can also open an interactive Python interpreter such as IPython [16] on any computer on the same network, type `import lyse; df = lyse.data(hostname)` (where `hostname` is the network name 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, and textual diffs highlighting changes between versions of experiments. Both the experiment logic and the analysis

```
rev:     156 (128ef198499e)
author:  chrisjbillington
date:    Sat Jun 02 17:54:23 2018 +1000
summary: Added history and attribution section
```

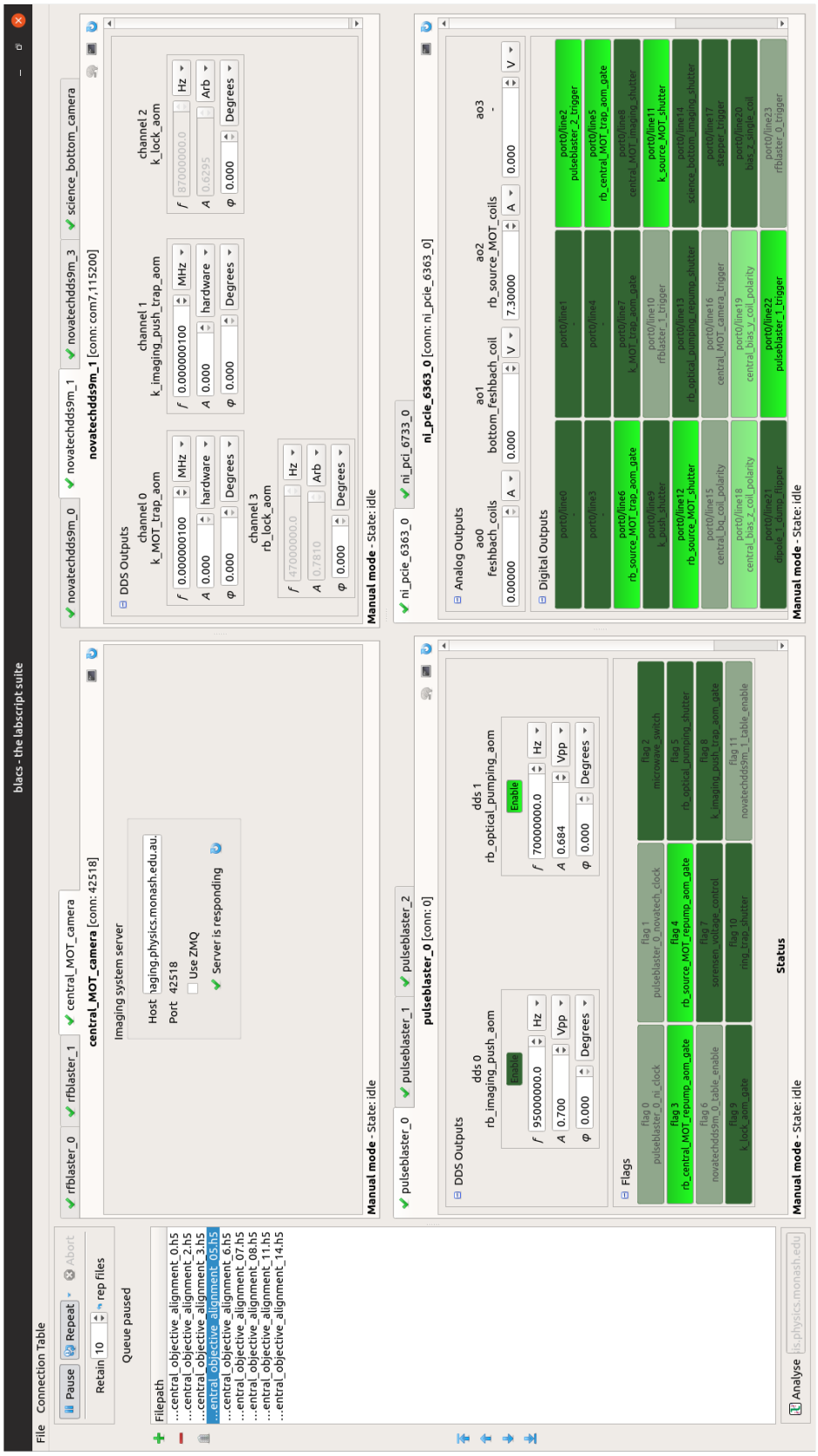
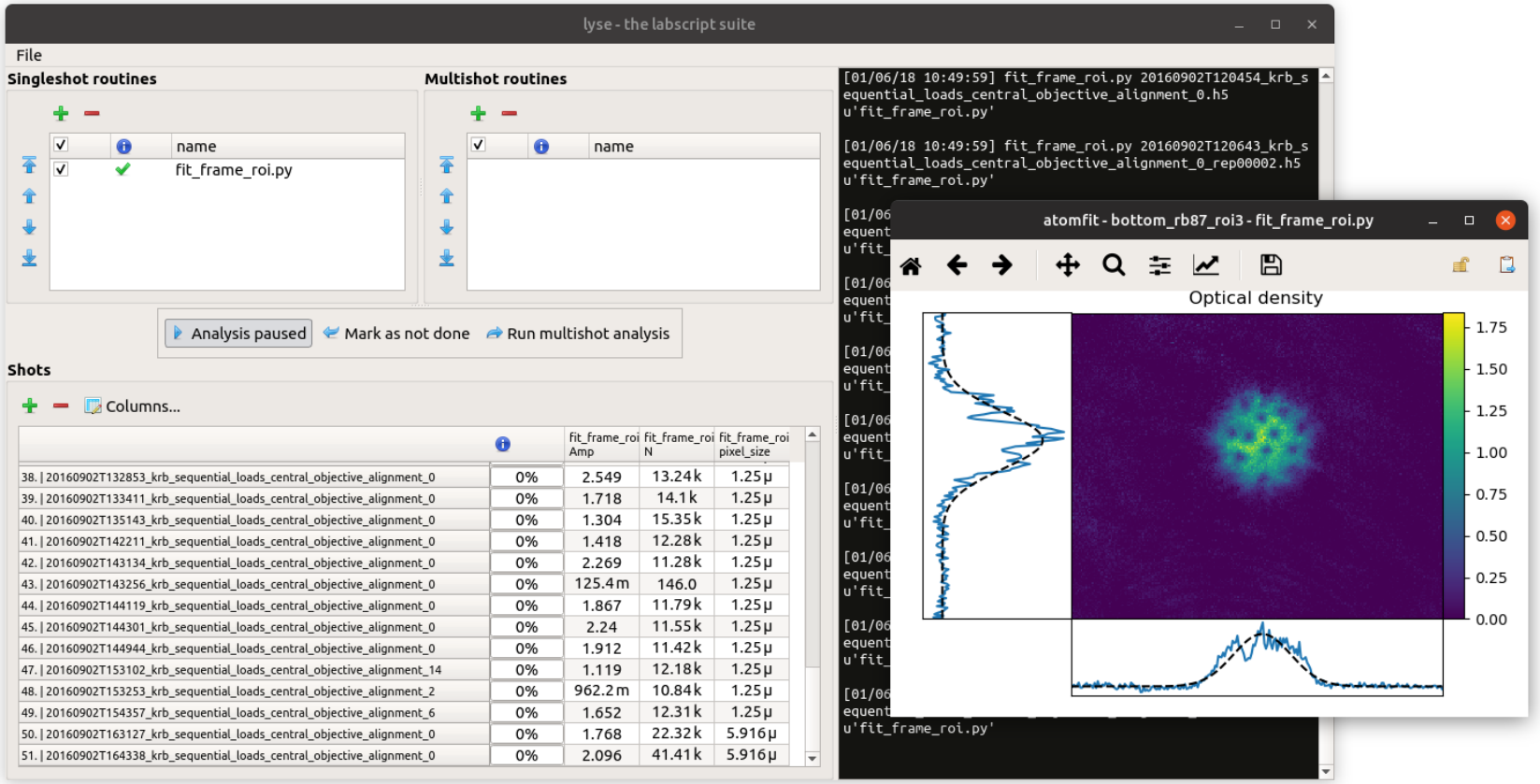


Figure 4.4: BLACS as of 2018, showing the experiment queue to the left and manual controls of devices within a tabbed interface to the right. When a hardware-timed experiment is not in progress, outputs may be controlled manually using this interface. Presently the GUI for four devices can be displayed simultaneously, though we plan to allow BLACS tabs to occupy their own windows or even reside on different computers to allow a better use of screen space. Output widgets are labelled with the names of the outputs they control and can be 'locked' to temporarily disallow modifications to their state. Most of the interface for each device is automatically generated from the description of the device and its outputs given in `labscript` code, though the author of a device driver for the labscript suite is free to include any graphical elements in a device tab. This screenshot was taken using the current device configuration of the Monash KRb lab.

```
rev:     156 (128ef198499e)
author:  chrisjbillington
date:    Sat Jun 02 17:54:23 2018 +1000
summary: Added history and attribution section
```



		fit_frame_roi Amp	fit_frame_roi N	fit_frame_roi pixel_size
38. | 20160902T132853_krb_sequential_loads_central_objective_alignment_0	0%	2.549	13.24k	1.25µ
39. | 20160902T133411_krb_sequential_loads_central_objective_alignment_0	0%	1.718	14.1k	1.25µ
40. | 20160902T135143_krb_sequential_loads_central_objective_alignment_0	0%	1.304	15.35k	1.25µ
41. | 20160902T142211_krb_sequential_loads_central_objective_alignment_0	0%	1.418	12.28k	1.25µ
42. | 20160902T143134_krb_sequential_loads_central_objective_alignment_0	0%	2.269	11.28k	1.25µ
43. | 20160902T143256_krb_sequential_loads_central_objective_alignment_0	0%	125.4m	146.0	1.25µ
44. | 20160902T144119_krb_sequential_loads_central_objective_alignment_0	0%	1.867	11.79k	1.25µ
45. | 20160902T144301_krb_sequential_loads_central_objective_alignment_0	0%	2.24	11.55k	1.25µ
46. | 20160902T144944_krb_sequential_loads_central_objective_alignment_0	0%	1.912	11.42k	1.25µ
47. | 20160902T153102_krb_sequential_loads_central_objective_alignment_14	0%	1.119	12.18k	1.25µ
48. | 20160902T153253_krb_sequential_loads_central_objective_alignment_2	0%	962.2m	10.84k	1.25µ
49. | 20160902T154357_krb_sequential_loads_central_objective_alignment_6	0%	1.652	12.31k	1.25µ
50. | 20160902T163127_krb_sequential_loads_central_objective_alignment_0	0%	1.768	22.32k	5.916µ
51. | 20160902T164338_krb_sequential_loads_central_objective_alignment_0	0%	2.096	41.41k	5.916µ

Figure 4.5: `lyse` as of 2018. To the top left is the list of single-shot routines, and to the right of it the multi-shot routines presently loaded (empty in this example). The list of shots is below them, showing some of the globals and analysis results of these shots. To the right is the output box where any textual output produced by analysis routines is displayed. Overlaid is a plot produced by a single-shot analysis routine. This screenshot was taken using shot files and an analysis routine script from the Monash KRb lab's two-dimensional turbulence experiment [17]

```
rev:     156 (128ef198499e)
author:  chrisjbillington
date:    Sat Jun 02 17:54:23 2018 +1000
summary: Added history and attribution section
```

Figure 4.6: Helpful advice from a PhD student of the Monash Spinor BEC lab.

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 to be explored 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 globals themselves, which are stored in non-textual 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, which is crucial in the experiment debugging process (Figure 4.6).

As in software development, experimental physics re-uses the same procedures time and time again, and benefits from a way to manage the complexity of turning large parts of functionality on and off, repeating them, or otherwise conditionally modifying their behaviour. In a traditional atomic physics control system, disabling or repeating a part of the experiment logic might involve tedious clicking to remove or duplicate each instruction involved. In Python, this can be a single `if` statement or `for` loop wrapping a function call containing the complexity of the part of the experiment being disabled or repeated. This 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 the 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 disable part of an experiment in the aforementioned hypothetical 'traditional' control system could be avoided if the system implemented a feature for conditionals—akin to `if` statements. But then what about *nested* `if` statements? To support all the use cases that may arise, one would ultimately be inventing and embedding a complete programming language within such a control system. Use of an existing complete (and well-designed) programming language obviates this need.

The use of an existing programming language for experiment control floe only aids the labscript suite in the case of 'compile-time' conditionals and other control statements—those that can be evaluated when the hardware instructions are produced by `labscript`—as opposed to those at '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). Such run-time control statements do require special treatment in `labscript`, and `labscript` 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 or 60 Hz magnetic noise from mains electricity by pausing the experiment until a fixed point in the

```
rev:     156 (128ef198499e)
author:  chrisjbillington
date:    Sat Jun 02 17:54:23 2018 +1000
summary: Added history and attribution section
```

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 `labscript` in the future, but would require hardware capable of holding multiple alternative sets of instructions in memory and able to switch between them on a digital edge or the state of a digital signal on some input. The SpinCore PulseBlaster—the device used by most users of the labscript suite to produce clocking pulses—supports this, but most output devices labs use with the labscript suite do not. Custom field-programmable gate arrays (FPGAs) implementing this functionality however for digital or analogue output could be integrated with the labscript suite for this purpose, as they have been for other purposes [18].

4.2.2 Modularity and the Unix philosophy

An aspect of the Unix philosophy [19] is that tools should 'do one thing and do it well'. The components of the labscript suite are not as minimal as they could be, but are nonetheless discrete components that encapsulate different concerns and communicate with to each other over network connections.[3] Thus in principle once can remove a component and replace it with another that plays a different role. For example, `runmanager` has been re-purposed by Philip Starkey [10] to generate parameter space scans for numerical simulations instead of experiments, by calling code other than `labscript` code. `lyse` is in use by Fred Jendrzejewski's group at Universität Heidelberg for on-line analysis of experiment results in the form of HDF5 files produced by a different data acquisition system.

[3] 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.

The separate programs of the labscript suite are also implemented in many cases as a number of sub-components exchanging instructions over network sockets, even though the collection of processes semantically comprise a single application. This architecture facilitates a fairly direct extension (currently in development) in which different parts of the same program can be run on different computers. One can then imagine having `lyse` analysis routines run on a remote computer (perhaps a server with a powerful GPU or many CPU cores for computationally intensive analyses), or having devices controlled by `BLACS` be connected to a different computer than the one running `BLACS`. This latter configuration can aid in reducing the effect 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.[4].

[4] A pertinent example is the bandwidth of a computer's USB 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. This is a limiting factor in the speed of experiments in the Spielman group's RbLi lab at the Joint Quantum Institute.

As mentioned earliers, `runmanager` is a graphical program, but also a software library for setting globals and compiling shots by running `labscript` 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 labscript 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 inflexible, and we no longer maintain `mise`. However a similar method is used in the Spinor BEC lab at Monash to integrate the MLOOP machine-learning optimisation library [20] with the labscript suite for experiment optimisation.

The separation of components also aids in development of the labscript suite. Programs can be ported one-at-a-time to use updated versions of libraries, and tested separately, enabling a more flexible model of development which has proved invaluable to the open-source development process.

```
rev:     156 (128ef198499e)
author:  chrisjbillington
date:    Sat Jun 02 17:54:23 2018 +1000
summary: Added history and attribution section
```

4.2.3 Off-the-shelf hardware

The labscript suite is the software part of an experiment control system, and does not mandate any particular hardware. Whilst drivers for a range of off-the-shelf and in-house devices are developed and maintained by myself and the other labscript suite developers, if one wants to use different hardware, whether off-the-shelf or custom, one can write drivers for it and use it with the labscript 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 designing a limited range of hardware specifically for the labscript suite or vice versa. It is difficult to anticipate what hardware capabilities experimentalists will require for atomic physics experiments in the future, and science is by its nature pushing the envelope such that converging on 'standard' hardware can be at odds with making scientific progress. Therefore it is better to leave the software as agnostic as reasonably possible when it comes to hardware.

Labscript suite developers and users have also made some hardware of their own, and there exists a low-cost pseudoclock and a digital out device both based on a sub-$100 microcontroller board (Called the PineBlaster and BitBlaster respectively). However these have proved difficult to maintain in the face of changing software development kits for the microcontrollers, and I suspect FPGAs implementing similar functionality would be a solution with greater longevity. Rory Speirs [21] has developed a low-cost FPGA-based pseudoclock (planned to be released as an open-source project) that may be an attractive alternative to the SpinCore PulseBlaster for use with the labscript suite.

4.2.4 Open-source, popular programming language and data format

Using open-source technologies in the labscript suite, as well as developing the labscript suite itself as an open-source project have benefited the project considerably.

The labscript suite itself being open source allows others to modify it to their needs, expanding the range of experiments that it can be used with. If modifications made are applicable to a wide enough range of users, they can be contributed back to the main project. Not least importantly, bugs in the code that have been worked around or fixed by a single end-user can be contributed back to the main project for the benefit of all. This has led to a steady improvement in the usability and stability of the software as usability issues and bugs have been noticed and fixed by people inside and outside the core development team. Sometimes there is disagreement about what features belong in the labscript suite, or more often, over how they should be implemented. The open-source nature of the project allows end users to continue to use an implementation of functionality that there may not be agreement about including in the main project, either permanently or until another implementation is available. This is preferable to simply being at the mercy of the core developers as to what features they will or will not implement (whether due to differing opinions or to time constraints).[5]

[5] Although a commercial model has the potential to to work here too, in which users pay for features to be implemented. There is also the 'best of both worlds' approach of bug and feature 'bounties', where users add a cash reward for implementing certain features or resolving bugs, adopted by some open-source software projects.

The use of third-party open-source technologies in the labscript suite has also been advantageous, as when the projects we rely on do not fully satisfy our needs, there is often the possibility of modifying them to meet those needs. Furthermore, these modifications, if agreeable to the developers of said libraries, can be included in the 'upstream' project to remove the requirement that we maintain the patches ourselves. To this end I have had changes accepted into the `numpy` [22], `pandas` [15], `pyzmq` [23] and `h5py` [24] projects which improve their behaviour for the labscript suite's purposes.

The use of a popular programming language with a gentle learning curve such as Python allows new students writing experiment code using `labscript` to get up to speed with what is necessary to work in their laboratory quickly compared to other programming languages. Many students and other physicists are already using Python

```
rev:     156 (128ef198499e)
author:  chrisjbillington
date:    Sat Jun 02 17:54:23 2018 +1000
summary: Added history and attribution section
```

for data analysis and other tasks, and so knowledge of it is more widespread than other languages, decreasing the barrier to entry for modifying the labscript suite or contributing to its development.

The use of the standardised HDF5 format, and of the popular `zeromq` [25] messaging protocol for communication between components allows interoperability with a wide range of other programming languages and technologies, such that software written in other programming languages can interact with running labscript suite programs and read the data files produced by them. The camera interface program `BIAS` mentioned earlier is an example of this—it is written in LabVIEW and yet reads and writes data to the same shot files as the rest of the labscript suite using the HDF5 library, and communicates with components of the labscript suite over the network using the `zeromq` library.

4.2.5 Collateral benefits

Developing and maintaining the labscript suite has involved achieving several intermediate, instrumental goals in order to achieve the ultimate goals of the software. Some of these solutions have been packaged as separate software projects, or are available from within the labscript suite, and may be used for other purposes.

For example, as mentioned, the labscript suite comprises several programs, each of which contains multiple threads and/or processes, communicating with each other by message passing (largely in line with the 'actor model' of concurrent programming[6]). As such the code contains reusable pieces for launching processes and initiating message passing with them, of redirecting output of subprocesses to some visible location rather than a terminal, or of starting and stopping 'servers' and creating 'clients' to either pipe data continuously or make discrete requests that necessitate a response. Furthermore, these multiple threads and processes often require access to the same files (the HDF5 files containing the data for each experiment run), and access to these files needs to be serialised to prevent data corruption, including the case of multiple computers attempting to access the file over the network. The solutions to these problems that we have arrived at are encapsulated into the `zprocess` project [27], an open-source software project officially separate from the labscript suite but very much developed to meet its needs for multiprocessing. This project is used not just for the labscript suite, but for other laboratory automation tasks. It is also used in the software implementation of an undergraduate experiment in the School of Physics and Astronomy at Monash University to allow students to remotely control the experiment and collect data over the internet [28].

[6] Though we had no formal education in this model before writing code in this style, and I attribute my fondness of the model to the computer game *SpaceChem* [26], in which the player constructs chemical 'reactors' comprising multiple threads of execution exchanging atoms and molecules.

Multi-threaded graphical programs can present development problems as the software library for the graphical interface generally must be accessed only from the one thread. This is generally the case for the Qt toolkit [29] as used in the labscript suite, and so interacting with the graphical interface from multiple threads generally involves message passing to request that an operation be performed in the 'main' thread. We have similarly encapsulated our solutions to this—as well as several other problems repeatedly encountered in using the Qt toolkit, such as an icon set, automatic loading of graphical layouts from external files, and others—into the `qtutils` project [30], which is used for other small graphical utilities in our group, separately from the labscript suite. This library has also been used in an undergraduate teaching context in the School of Physics and Astronomy at Monash University to improve IT infrastructure for digital logbooks used by undergraduate students.

We also have a number of debugging/profiling tools that I often find myself reaching for to debug numerical simulations, or other code unrelated to the labscript suite.

```
rev:     156 (128ef198499e)
author:  chrisjbillington
date:    Sat Jun 02 17:54:23 2018 +1000
summary: Added history and attribution section
```

4.3 Recent and future developments

Since the publication of our paper in 2013 [9], the number of groups using our software has increased considerably, from being limited to just the two groups at Monash, to a modest number of groups around the world.

In 2014 during my PhD I was invited to visit the Joint Quantum Institute in the group of Ian Spielman, with the aim of improving the labscript suite to make it more generally usable and easier for others to install. The primary goal was to port the graphical programs from the GTK toolkit [31] to the Qt toolkit [29], and to write an installation program that would automate the previously somewhat tedious installation process. These goals were achieved, and the installation process for the labscript suite is now a matter of installing an appropriate Python environment, and then downloading and running our installation script.

The following subsections detail this and a number of other developments since the publication of our paper, as well as planned and in-progress improvements to the labscript suite.

4.3.1 Port to Qt

The port to Qt [29] was a crucial development. As discussed in the publication reproduced at the end of this chapter, we initially chose the GTK toolkit for its cross-platform compatibility and good Python bindings [32]. This proved to be the wrong decision, as the GTK project has rapidly developed and dropped support for older versions, with the newer versions being difficult to install or deploy on operating systems other than Linux, and with changes being significant enough to impose a considerable ongoing development cost to keeping code up to date with them. Although slated as a cross-platform toolkit, the GTK project is developed by and primarily serves the needs of the GNOME project [33], and thus its development is driven by those needs. There is little incentive for Windows- or Mac-specific bugs to be fixed, or for the installation process to be improved. Although it's an open-source project such that people other than the core developers could contribute fixes to these issues, the Linux-centric nature of the project impedes these fixes from being accepted for inclusion. Switching to the (also open-source) Qt toolkit is the chosen solution for most for cross-platform graphical software, and so this is what we have now done as well. The experience of both maintaining and installing the labscript suite is much improved as a result. Being the standard cross-platform GUI toolkit, Qt is already available in the standard Anaconda Python distribution [34], which is the preferred Python distribution among scientists at the present time. With our `qtutils` package, an icon set is available, obviating the need to install a separate icon pack as was previously the case. Most importantly, the Qt software project exists to serve the needs of graphical programs generally, and is used by many major cross-platform projects. Combined with the fact that Qt is open source, this is insurance against future breaking changes in Qt or against the Qt Company going out of business—open source projects can be forked and maintained by the community, and the more popular they are the more likely this is to occur if they take an unpopular turn or are abandoned by their present maintainers. We therefore have confidence that the Qt project's direction is aligned with the needs of the labscript suite, and that we can continue to rely on it in a way we could not with the GTK project for our project's longevity without introducing unnecessary technical debt.

4.3.2 Python 3

The labscript suite was initially written in version 2.7 of the Python programming language, even though version 3 of Python had been released several years prior. The Python

```
rev:     156 (128ef198499e)
author:  chrisjbillington
date:    Sat Jun 02 17:54:23 2018 +1000
summary: Added history and attribution section
```

community has in this time been in a decade-long transition from one version to the next due to some non-trivial differences between the two versions of the language. I believe we made the right decision to initially use Python version 2.7, as many of the technologies the labscript suite relies on did not have Python 3 compatibility for some time. However, the point of inflection in the adoption curve for Python 3 has occurred in the last two years or so, and now is the right time for Python 3 adoption. With the help of third party contributors (primarily Jan Werkmann), the entirety of the labscript suite has now been ported to run on both Python 2 and Python 3, though we do not consider the support 'official' until more testing has been performed, particularly using hardware not in use by the groups that are running with Python 3 daily.

We do not expect such an extended issue such as this to occur again: the Python core developers consider this transition to be a one-off, and future porting efforts of labscript-suite components will likely be no more work than the usual required to keep up with minor changes between language versions.

4.3.3 More devices, more features, general polish

There are more devices with compatibility with the labscript suite, and more models and features of existing devices are now supported. The programs are easier to use, and many cases where obscure errors were thrown have been replaced with friendlier error messages explaining the situation in human-readable terms. Following is an incomplete list of minor to modest usability improvements:

- `BLACS` now has a plugin to delete shots that are repeated versions of previous shots. This prevents unnecessary consumption of disk space when these shots are running only to keep an experiment 'warm' (initially implemented by Ian Spielman, and re-implemented by me as a plugin for `BLACS`).

- `lyse` now more gracefully handles shot files that have been deleted off disk: declining to run single-shot analysis on them, but keeping their data in the dataframe available for multi-shot analysis until they are deleted from the `lyse` interface. This aids in, for example, diagnosing a day-long drift in some performance characteristic of the experiment even though most shot files are deleted due to the aforementioned desire to save disk space (implemented by me).

- `lyse` is now more performant, only updating those values of the dataframe that have changed, and minimising the number of times it opens a HDF5 file. This improves performance for very large numbers of very short duration shots, as is common in ion trapping (implemented by Jan Werkmann with changes by me).

- `BLACS` has had some bugs resolved that unnecessarily introduced delays on the order of 0.5 s in between running shots. These delays were not very noticeable for the cold atom experiments of order 15–30 s, but again are important for the ion trappers (implemented by me and Philip Starkey).

- `BLACS` tabs now have a separate optional terminal output for each device subprocess, allowing simpler debugging and development of devices (implemented by me).

- More flexible camera interface. There are now a number of camera 'servers' in use by various groups playing the role that `BIAS` plays at Monash, including a fork of `BIAS` named `unBIASed` by Ian Spielman (changes made to facilitate communication with Python camera servers implemented by me)

```
rev:     156 (128ef198499e)
author:  chrisjbillington
date:    Sat Jun 02 17:54:23 2018 +1000
summary: Added history and attribution section
```

- labscript can now accept arbitrary function ramps using user-supplied functions, not limited to the built-in list of functional forms. This has always been possible somewhat manually, but now has a more friendly interface (implemented by me).

- There is now a unified interface for saving and retrieving configuration settings of devices in labscript to the HDF5, including Javascript Object Notation serialisation for complex data types that do not coincide with a HDF5 datatype. This replaces a number of ad-hoc serialisation methods previously in use to store configuration settings of devices (specification designed by Ian Spielman and core developers, implementation by Ian Spielman and me).

- There is now a unified labscript device driver for National Instruments DAQmx devices, removing the code duplication and complexity of maintaining multiple device classes for this range of devices. This class exists in the fork of the labscript_devices repository in use by the Spielman group at the Joint Quantum Institute, but will be merged into the mainline codebase soon—it is already in use by groups who are not otherwise using the Spielman fork of the code, and so has undergone some testing and bugfixes outside of the hardware in the Spielman group. In time the model-specific code will likely be removed in favour of the unified interface (implemented by Ian Spielman).

- The ability to mark certain points in time of the experiment with a named marker, visible in runviewer to visibly delineate different stages of the experiment (implemented by Jan Werkmann).

- A 'nonlinear time' mode for runviewer, where the time axis is not linear but instead uses a different timestep for the different stages of the experiment as described by the markers. This allows short timescales and longer timescales to be visible on the same plots in runviewer. (implemented by Jan Werkmann with changes by Shaun Johnstone).

- The ability to mark digital outputs as 'inverted', such that digital low represents a device being on, semantically speaking. The buttons for these outputs are represented with different colours in the BLACS interface to avoid confusion (implemented by Jan Werkmann).

- Gated clocks: devices with vastly different memory capabilities can receive clocking signals from the same clocking device such as a PulseBlaster, but on different outputs such that the clock ticks intended for one device are not received by other devices. There is still a single pseudoclock, but its outputs are 'gated'—whilst the clock is ticking for one device it is not ticking for another. Two devices configured in this way are still however sharing a clock in a sense: they cannot both receive rapid clock ticks simultaneously but at different rates, multiple pseudoclock devices are still required for this (implemented by Philip Starkey).

- lyse plots can be copied to the clipboard with a button click, reducing the number of steps to include plots in a digital log book (implemented by me).

- runmanager, lyse, and runviewer now have the ability to save and load configuration settings, such that the same sets of globals files in the case of runmanager or the same sets of analysis routines in the case of lyse and the same view settings in the case of runviewer can be loaded at start-up of each application (implemented by me for runmanager and Jan Werkmann for lyse and runviewer).

```
rev:     156 (128ef198499e)
author:  chrisjbillington
date:    Sat Jun 02 17:54:23 2018 +1000
summary: Added history and attribution section
```

- runmanager now allows finer control over parameter spaces, including randomising the order of a parameter space scan on a per-axis basis, and control over the nesting order in which the axes are looped over (implemented by Philip Starkey).

4.3.4 Optimisation

The program mise, mentioned in the paper, has been deprecated. Nothing has replaced it, though due to the modularity of the labscript suite, optimisation is still possible through use of the runmanager library directly. This is a testament to the pluggability of the labscript suite components, but could nonetheless be improved.

I plan to improve this functionality in the future, and one of the near-term development goals is to add a 'remote' application programming interface (API) for runmanager. This will enable a program to control a running instance of runmanager to set the values of globals and initiate shot compilation and submission to BLACS. This will be a much simpler interface as well as being compatible with just-in-time compilation, discussed below.

4.3.5 Just-in-time compilation

One feedback mechanism not previously anticipated—more accurately described as *feed-forward* in this context—is the need to modify one or more parameters for the very next shot to be run on the experiment, but otherwise remain performing some parameter space scan or repetition. For example, in the Spielman group's atom chip lab, an environmental magnetic field drift on a several hour timescale is corrected for by performing an error measurement each shot, to be fed-forward to the next shot as a change in the applied field bias. Other than this, the experiment is not performing any optimisation or feedback. In the chip lab, this functionality is implemented by having BLACS use the runmanager API to re-compile the shot files just before running them, to be sure to include the updated magnetic field bias estimate.

This works, and is an example of functionality being implemented by users to serve an immediate need. However I would like to move this 'just-in-time' compilation into runmanager, so that the shot is compiled only once rather than being recompiled. Compilation by BLACS is unappealing since it may be on a different computer with different versions of the code being compiled, leading to the possibility of subtle errors, and more generally violates the design philosophy of separation of concerns that has served us well so far.

For this reason as well the applicability to optimisation mentioned above, I plan to implement both a remote API for runmanager as well as a 'compilation queue' containing shots yet to be compiled, whose variables can still be changed (possibly remotely) up until the moment BLACS requests a new shot,[7] triggering compilation to occur.

[7] Either because its queue is empty, or nearly empty—by requesting a shot before the queue is fully empty one can prevent the experiment being idle during compilation, at the expense of the feed-forward changes taking effect only some number of shots in the future.

4.3.6 Fixed duration shots

Since BLACS takes a usually small—but variable—amount of time to program the hardware in between shots, this contributes to a variation in the average proportion of time an atom dispenser is receiving current or ultraviolet light-induced desorption is active, leading to vapour pressure variations in experiments that operate in this manner. In the Spielman fork of the BLACS repository, there is functionality to set a fixed overall duration for an experiment, such that after programming devices, BLACS waits some additional amount of time such that experiments are run at precisely equal intervals. This can be used not only to smooth out the variations in programming time, but also to smooth our variations in actual shot duration caused by changes of parameters that affect the shot duration, or changes in shot duration due to variable-length waits while the experiment

```
rev:     156 (128ef198499e)
author:  chrisjbillington
date:    Sat Jun 02 17:54:23 2018 +1000
summary: Added history and attribution section
```

is paused, such as servoing a MOT load as mentioned in Section 4.2.1. This feature is yet to be merged into or reimplemented in the mainline labscript suite codebase from the Spielman fork.

4.3.7 Remote devices

A work in progress is to allow devices to be connected to any computer, not just the one that BLACS is running on. As mentioned in Section 4.2.2, this allows one to avoid long signal cables and their associated propagation delays and potential ground loops, as well as to make better use of limited computer resources by spreading processing over more computers. It also would allow one to make better use of computer screen real estate if the graphical interfaces for each device within BLACS could be presented on a different computer as well, since the BLACS interface can become quite cluttered with a large number of devices. One reason for making a separate camera control program in the form of BIAS was to be able to view images immediately at all times without having to ensure the correct tab in the BLACS interface is active. Being able to display these tabs as separate windows on different screens or computers will obviate this need and allow cameras to be once again treated the same as other devices.[8]

In discussion with developers and users, we have designed a specification for how the desired layout of devices on a network will be described by users in labscritp code, and we have a partial implementation. Most of the work toward this has been in the zprocess package, which I have been adapting to these needs, including the use on encryption to ensure that the ability to start processes on remote computers is secure. This feature in zprocess is nearly complete, after which some work in BLACS will need to be done to implement the designed specification and make the appropriate requests to zprocess to launch remote processes for communicating with hardware and displaying the graphical interfaces for them. These two features are planned to be separate, such that BLACS, the graphical interface for a specific device, and the device itself can be on the same computer, or two, or three, for maximum flexibility in where the graphical interfaces and actual hardware is located within the lab.

[8] This was not the only reason BIAS was made as a separate system, another reason was the availability of software libraries for interacting with certain cameras, these were available for LabVIEW at the time, but not Python. Python wrappers, pyvisa [35] and pynivision [36] for the National Instruments VISA and NI-Vision libraries have since become available, removing the need for LabVIEW to be used with cameras or other devices requiring these interfaces.

4.4 labscript version 3

The labscript compiler itself is the oldest part of our codebase, and has changed significantly since its initial incarnation. We have become more skilled programmers in the 7 years since it was first written, and some design decisions have proved to be the poorer choices. For example, most processing performed by labscript during processing is *destructive*—that is, new data replaces old data as processing steps proceed. Specifically, timing delays are incorporated by replacing timing data in instructions, rather than introducing new variables in code specifically for the delayed timing data whilst leaving the original timing information intact. This makes it difficult to debug where timing problems have occurred, and makes the code fragile to the introduction of bugs in which timing offsets are accidentally taken into account twice or not at all, as opposed to exactly once as required. Furthermore, these timing calculations are performed mostly using floating point arithmetic, and so all comparisons need to be performed with some tolerance or rounding. This is error-prone and unnatural given that the hardware devices generally have quantised timing in their instructions. Finally, by the time an exception occurs in labscript indicative of the user requesting something not possible (such as two instructions closer together in time than the hardware is capable of), labscript no longer has much information about where in the user's code the instruction originated, making it difficult to give the user information that helps them resolve the issue.

To address these three concerns, I have been working on an experimental restructuring of the core instruction and timing processing of labscript, in which all processing is

```
rev:     156 (128ef198499e)
author:  chrisjbillington
date:    Sat Jun 02 17:54:23 2018 +1000
summary: Added history and attribution section
```

non-destructive, the points in the user's code where instructions are created are noted for later use in error messages, and all timing calculations are performed using integer arithmetic after quantising timing details as early as possible in processing according to the time resolution of the pseudoclock controlling the timing of each device. This project, called `labscript_core`, if successful will eventually replace the part of `labscript` responsible for timing calculations and instruction handling, making the code better from both the perspectives of users and developers. If possible I will keep the timing computations separate from the other higher level parts of `labscript` (globals, device properties, HDF5 files) so that it can be independently tested (ideally in the context of an automated test suite) and verified so that we can have a high degree of confidence in the output it produces, and confidence that regressions have not been introduced when changes are made.

4.5 Other future developments

A pressing concern is to unify the mainline labscript suite code with the Spielman group's fork of the code. This will involve merging (or re-implementing in the mainline codebase) the remaining features present in the Spielman fork as discussed in the previous sections, as well as others I have not mentioned such as a progress bar showing the progression of the experiment, and the ability to insert small analysis scripts to be executed by `BLACS` at the end of a shot with the sole purpose of updating parameters in `runmanager` for feed-forward functionality as described in Section 4.3.5.

There are a number of 3rd party contributions (mostly authored by Jan Werkmann) awaiting approval by a core maintainer such as myself or Philip Starkey. Some of these are:

- Analog input widgets for `BLACS`: these show a numerical value for the voltage of each analog input a device has, or optionally can display an interactively updating plot of the voltage trace over time.
- Plugin tabs for `BLACS`: this allows plugins for `BLACS` to insert tabs into its graphical interface, allowing plugins to have rich interfaces of their own without having to interfere with the main interface of `BLACS`

Other changes proposed but not implemented include:

- 'Analysis globals': like the globals set in `runmanager`, but set in `lyse` instead. Presently, users are 'abusing' globals set in `runmanager` in order to configure how analysis will run. For example, there are globals being set by users that tell analysis code which pair of variables to plot against each other. As this may change after a shot has run, an interface where 'analysis globals' can be set and used by analysis routines would be preferable.

Finally, the entire project would benefit from more and better documentation. Documentation exists,[9], though it could be more thorough, and more importantly the development process has not ensured that documentation keep up to date with changes to the code. Instead, new information is often hidden in comments within the code or in commit messages. I see this as a flaw in the development process, rather than one of lack of effort or consideration. The Python community has a solution to this problem, which is to use libraries that turn appropriately formatted code comments into proper documentation. This then creates incentives to write and update these comments to a higher standard, knowing that they will be visible in official documentation—and this standard is more likely to be enforced during code reviews that occur when pull requests are made for changes to be included in the mainline code repository. Such a change of

[9] Primarily written by Philip Starkey

```
rev:     156 (128ef198499e)
author:  chrisjbillington
date:    Sat Jun 02 17:54:23 2018 +1000
summary: Added history and attribution section
```

process would help maintain and improve documentation continuously over time, and is better in my opinion than occasional bursts of effort and re-writes of the documentation.

In light of the existing open-source development model, an effective strategy to implementing a change such as this is to sow the seeds what we would like to see by putting the documentation rendering mechanism in place despite the current inadequacy of the documentation contents, and then require that contributions (from core developers or otherwise) in future amend the documentation comments (called 'docstrings' in Python parlance) appropriately. This way over time the documentation will improve, and being code rather than a PDF or Microsoft Word document, be amenable to pull requests and bug reports in the same manner as the rest of the code, which we have seen leads to inexorable improvement via an open process that accepts external fixes from others.

There is also a substantial quantity of best-practices and lore built up about hardware as well as software, including many tips and tricks that are specific to certain setups. Whilst the labscript suite has a mailing list in which much of this information is exchanged, it would probably benefit the project to have a user-editable wiki, to decrease the barrier-to-entry for users to share this type of domain-specific knowledge outside the context of an email thread even though it may not suit the official project documentation.

4.6 Project history and attribution

This section acknowledges the contributions of different authors to the different components of the labscript suite, and provides a history of major developments. This is not comprehensive, and many improvements have been provided by others; nonetheless the majority of the design and programming effort behind the labscript suite is due to the authors responsible for the initial creation and major developments of each subproject as outlined below. A full history of the codebase is publicly available on our on-line version control repositories [27, 30, 38] and can be used to credit any particular change or piece of code to the appropriate author.[10] Plots of present and historical authorship by number of lines of code are shown in Figure 4.7.

[10] With the exception of generic usernames occasionally being used from lab computers such that authorship was not recorded.

The initial idea of having object-oriented Python code compiled to low-level instructions was inspired by a similar implementation by Scott Owen and David Hall [39] in which experiment logic is described in C++ and programmed into the hardware using a LabVIEW program with a state-machine architecture.

Toward this goal I developed an initial implementation of the `labscript` compiler in Python in February 2011, subsequently improving it over time. Throughout the following year or so, the architecture of the software suite, its components and how they would interact was developed by members of the Monash Quantum Fluids group including Lincoln Turner, Russell Anderson, myself, Philip Starkey, Shaun Johnsone, Martijn Jasperse, and others.

The initial implementation of programming the `labscript`-generated instructions into hardware in LabVIEW was due to Russell Anderson in early 2011, followed by the initial version of `runmanager` written by me in August 2011 and the initial version of `runviewer`, also by me, in September 2011. Early in development I moved `runviewer` to use the Qt toolkit in order to take advantage of the `pyqtgraph` [14] plotting library, which was necessary for acceptable performance of `runviewer`.

Philip Starkey wrote the initial version of `BLACS` in October 2011, superseding the LabVIEW control system. In November 2011 Philip Starkey and I re-architected `BLACS` to move to a multi-process model (it was previously only multi-threaded) and improve the state machine architecture it uses.

In February 2012 I wrote `lyse`. In April 2012, Philip Starkey re-wrote a substantial portion of `runmanager`, making its user interface suitable for larger number of globals and improving other functionality.

```
rev:     156 (128ef198499e)
author:  chrisjbillington
date:    Sat Jun 02 17:54:23 2018 +1000
summary: Added history and attribution section
```

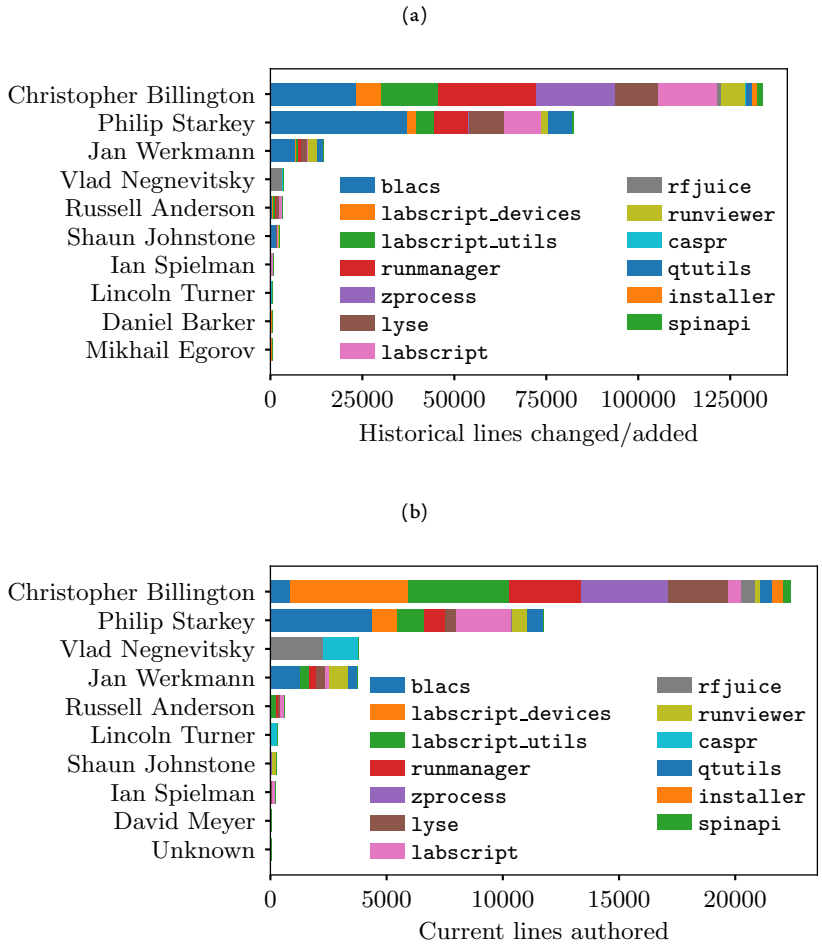


Figure 4.7: Authorship by lines of code of each component of the labscript suite, for the top ten authors. (a) Authorship of line changes over the entire project history. (b) Authorship of lines currently present in the most recent revision of each component. The line counts for these plots were obtained using the mercurial 'churn' extension and 'blame' command to analyse all Python, Cython and C source files in our code repositories. This excludes documentation (most of which is in Microsoft Word format) as well as layout files for the graphical interfaces of the programs. Although these results are broadly representative of authors' contributions to the components of the labscript suite, they should be taken with a grain of salt. Authorship is difficult to ascertain programatically as refactoring counts the same as original authorship even though usually less credit (though not zero) should be assigned to someone refactoring code than the author writing it originally. As such my apparent contribution to `labscript_devices`—the repository containing device drivers—is exaggerated due to me being the one to migrate this code from another repository. Similarly Jan Werkmann's contributions to `BLACS` are exaggerated due to recent renaming of one particular large source file. We both have made original contributions to both repositories as well, but these are not distinguishable from refactoring in these plots. The three repositories `caspr`, `rfjuice` and `spinapi`—not previously mentioned in this chapter—are libraries for interacting with devices, the former two for the 'RFBlaster', an RF synthesiser designed and built by Vlad Negnevitsky and used at Monash University and the Australian National University [37], and the latter for the SpinCore PulseBlaster, in use by most groups using the labscript suite.

```
rev:     156 (128ef198499e)
author:  chrisjbillington
date:    Sat Jun 02 17:54:23 2018 +1000
summary: Added history and attribution section
```

In January 2013, Philip Starkey ported `BLACS` to use the Qt toolkit.

In May 2014, Philip Starkey expanded the capabilities of the `labscript` compiler by implementing the 'gated clocks' feature, allowing more devices to be controlled by a single pseudoclock device such as a SpinCore PulseBlaster. Between August and November 2014 while visiting Ian Spielman's group, I ported `runmanager` and `lyse` to the Qt toolkit, wrote the labscript suite installer script and adopted the practice of tracking dependencies of the different components on each other using semantic versioning.

Throughout this time device driver code was contributed, various features implemented and bugs fixed by many, including Russell Anderson, Shaun Johnstone, Martijn Jasperse and Ian Spielman. Documentation was written primarily by Philip Starkey. `BIAS` was written and maintained by Martijn Jasperse from late 2011 onward.

When we started writing applications in Qt, I wrote the initial code for multithreading in Qt that became the `qtutils` [30] package, with most of the later functionality in the package added by Philip Starkey.

The `zprocess` package, encapsulating the network communication and multiprocessing code common to several components within the labscript suite, was written and is maintained by me (as always with some bugfixes and features contributed by others).

4.7 Conclusion

The labscript suite is an increasingly mature software project for control of hardware-timed experiments. It has an increasing number of users and contributors, and has evolved to keep up with changing software environments in the sense of switching to use more dependable software libraries such as the Qt GUI toolkit, and of keeping up with updates in language and library changes, such as the shift to Qt version 5 and to Python 3. It is a living project accepting changes from non-core developers, and is free for anybody to use under a permissive license. Due to the modular design and open development process, the labscript suite has thus far avoided some of the pitfalls that befall many laboratory control systems and software, such as relegation to legacy software or hardware environments due to a lack of development process capable of keeping them up to date, or long-standing bugs not being resolved because a fix applied in one place has no mechanism of making it into other users' installs of the software, or because the source code is only understood by or available to a small few. The labscript suite is hardware-agnostic, ensuring its use is not restricted to officially sanctioned or in-house hardware. Finally, it is written in a very popular programming language with an abundance of on-line resources and a vibrant community behind it, within both scientific and software engineering circles. Rather than the implementation details of the code itself, it is these decisions that I think are causing the project to thrive and be as beneficial to experiment physics research as it has so far.

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

See over page for a reproduction of our 2013 paper, *A scripted control system for autonomous hardware-timed experiments*, © American Institute of Physics 2013, Reproduced with permission.

```
rev:     156 (128ef198499e)
author:  chrisjbillington
date:    Sat Jun 02 17:54:23 2018 +1000
summary: Added history and attribution section
```

A scripted control system for autonomous hardware-timed experiments

P. T. Starkey,[a),b)] C. J. Billington,[a)] 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,[1] 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,[2] quantum computation,[3] and quantum simulation.[4]

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.[5] 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.[6] In the other, experiments are instead described graphically using a custom user interface.[7–11] 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.[10,11]

The two approaches need not be mutually exclusive: by separating experiment parameters from experiment logic, parameters can be manipulated graphically and logic textually.[12,13] 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,[14] exchanging references to the experiment file.

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

a)P. T. Starkey and C. J. Billington contributed equally to this work.
b)Author to whom correspondence should be addressed. Electronic mail: philip.starkey@monash.edu

 © 2013 AIP Publishing LLC

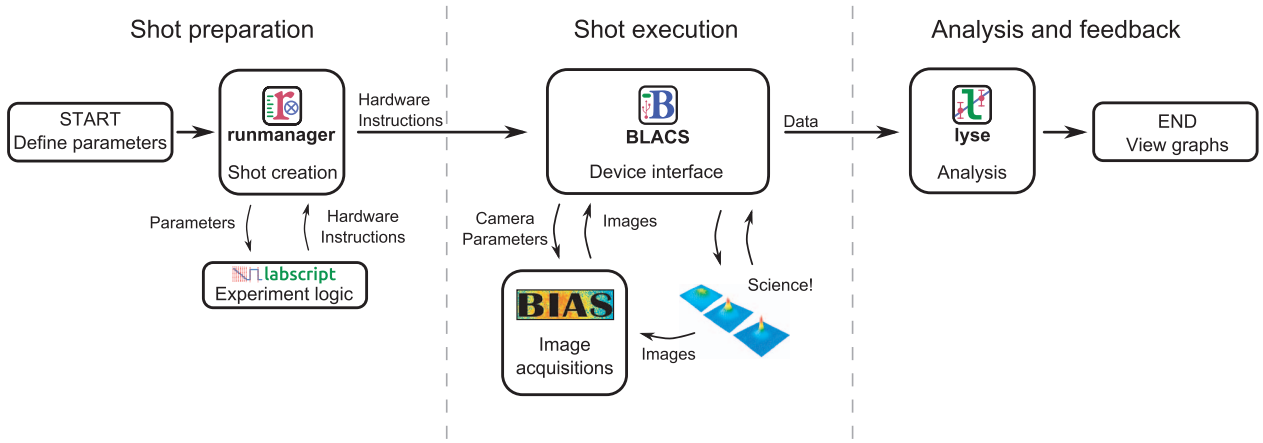


FIG. 1. Each experiment shot comprises three stages: preparation, execution, and analysis. Arrows indicate how the HDF file for an experiment shot passes between software components of the labscript suite. Only the shot execution stage is coupled to hardware timing, allowing new shots to be created and 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 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).[16] This calls for abstraction to shield the user from low-level interaction. We have created a software library for Python,[17,18] `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.[1] 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[8,9,11,12] 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

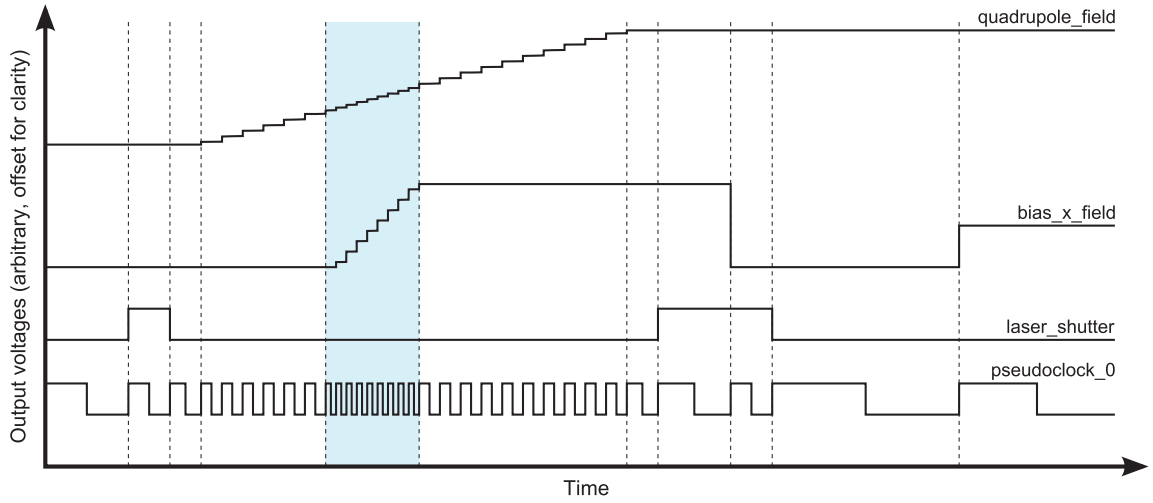


FIG. 2. An example of digital and analog voltage outputs generated by the labscript code in Fig. 3. The pseudoclock (lower trace) ticks when a digital output must change, or at the requested sample rate for time-varying analog outputs (upper two traces). Dashed vertical lines indicate a change in the 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.[19] 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[11–13] 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.[5] 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_BEC()` 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 `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`, `1E-3`, `sin(pi/2)`, or `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 π-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 `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.

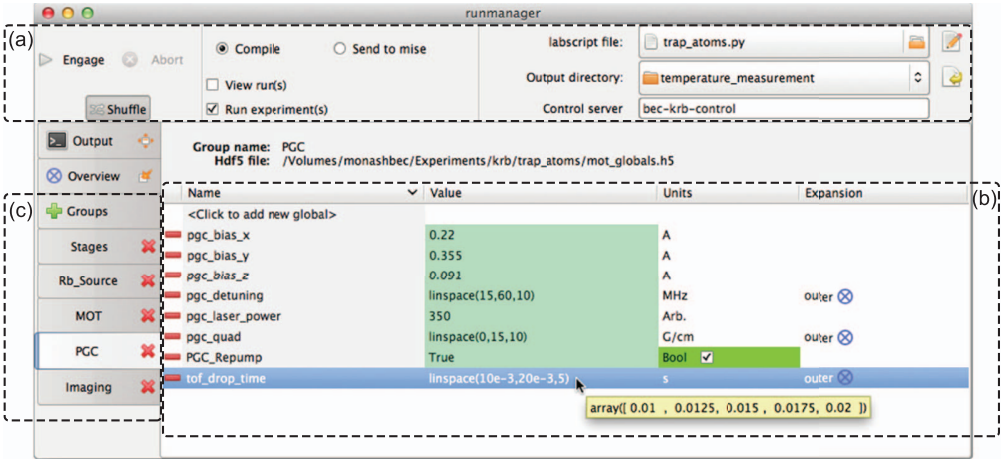


FIG. 4. The `runmanager` interface for configuring experiment parameters. (a) The experiment logic is specified by the labscript file (here trap_atoms.py). HDF files for experiment shots created by `runmanager` are saved in the output directory. (b) The value of experiment parameters ("globals") are 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.[20] 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.[21] 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

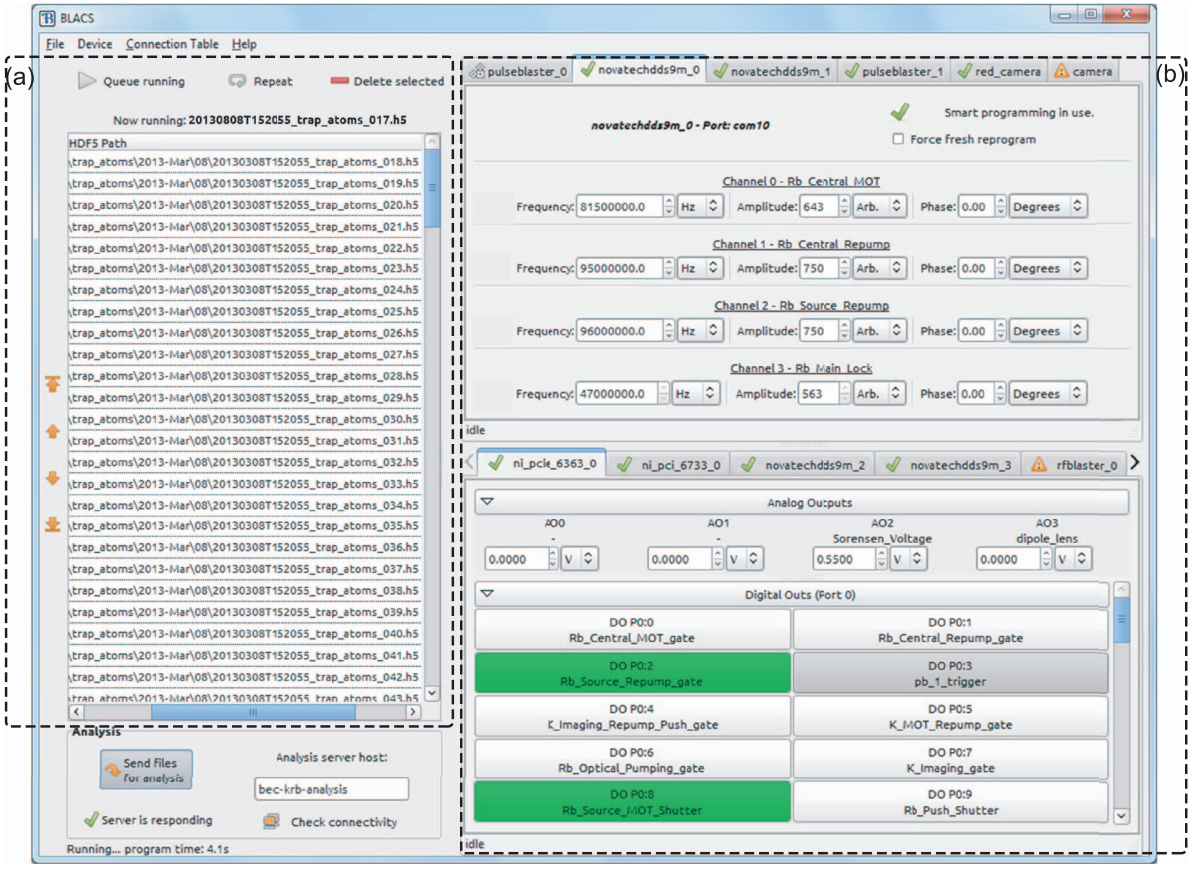


FIG. 5. The BLACS interface for controlling hardware. (a) The queue of shots submitted via runmanager. (b) The manual control interface. Each tab controls one device. Controls for all outputs are automatically generated and are named based on the BLACS connection table.

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.

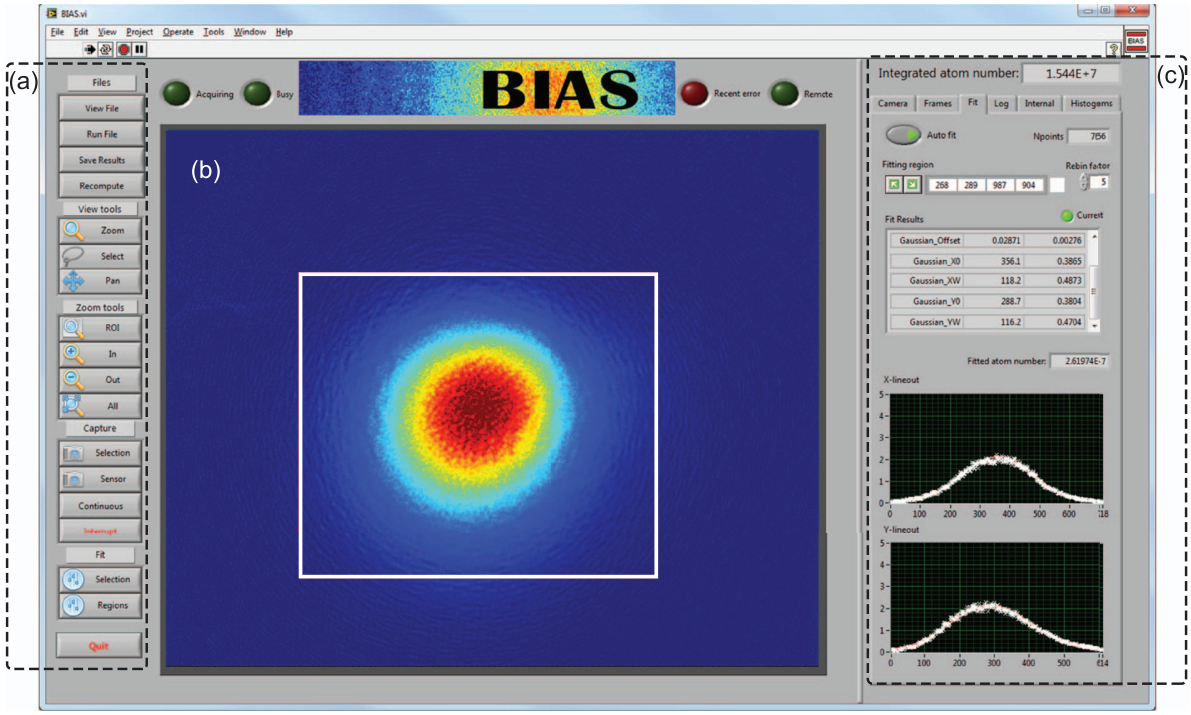


FIG. 6. The `BIAS` interface displaying a laser cooled atom cloud. (a) Manual controls for loading and capturing images, selecting regions of interest and zooming. (b) Computed optical depth (OD) image of the atoms, with a region of interest (white) selected for fitting. Multiple regions of interest 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`[22] DataFrame—for multi-shot routines, containing all globals as set by `runmanager`, and all single-shot analysis results. With `pandas` and the standard Python scientific stack of `numpy`,[23] `scipy`,[24] and `matplotlib`,[25] `lyse` provides a powerful environment for analysis.[26]

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.[27] 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 `runmanager` 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:

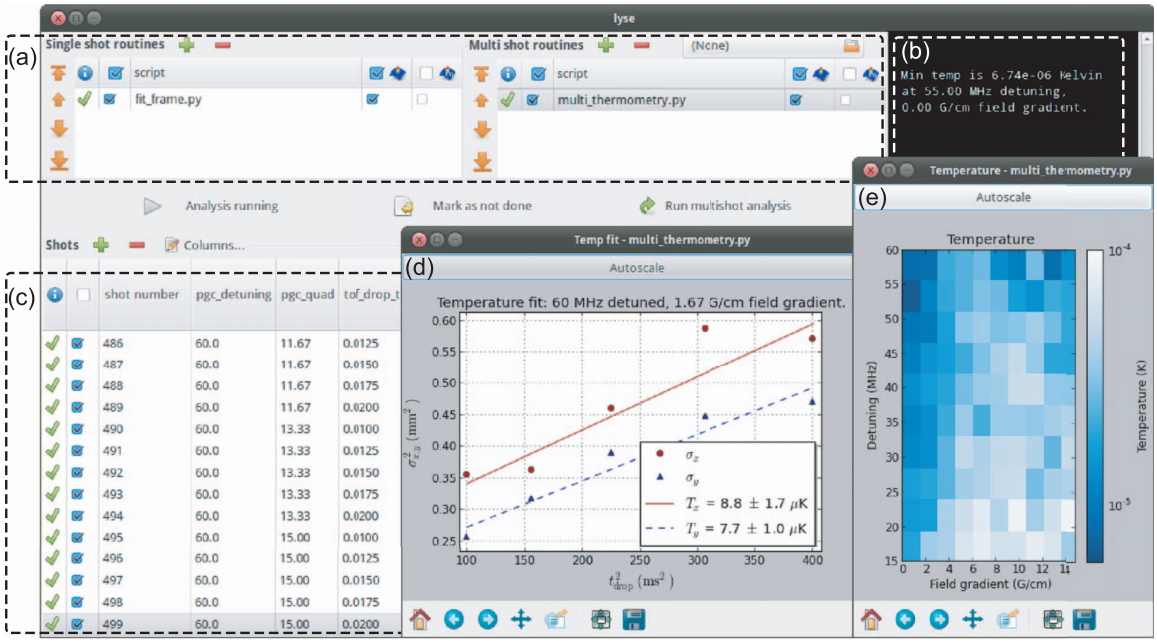


FIG. 7. The `lyse` interface. (a) Routines can be selected to analyze single or multiple shots. (b) Terminal output from the analysis routines in (a). (c) Table of shots; columns show globals and analysis results. A small subset of columns is displayed here. (d) A fit yielding the temperature of laser 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`[28] is a variation on pointed directed mutation,[29] 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.

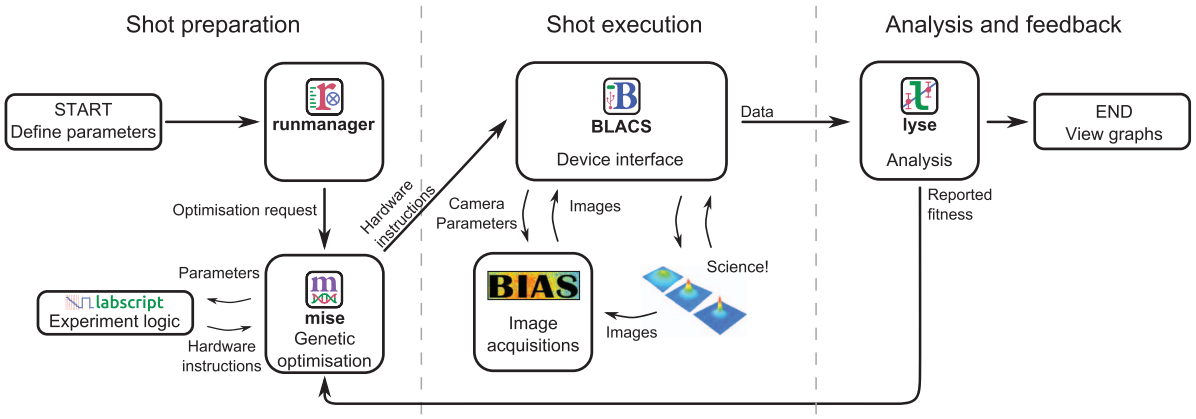


FIG. 8. The data flow for closed loop optimization. In contrast to Fig. 1, analysis results are used to determine future shots automatically. The optimizer `mise` varies parameters, directly calling `labscript` to compile new experiment shots. Parameters to be optimized are selected by the user in `runmanager`.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

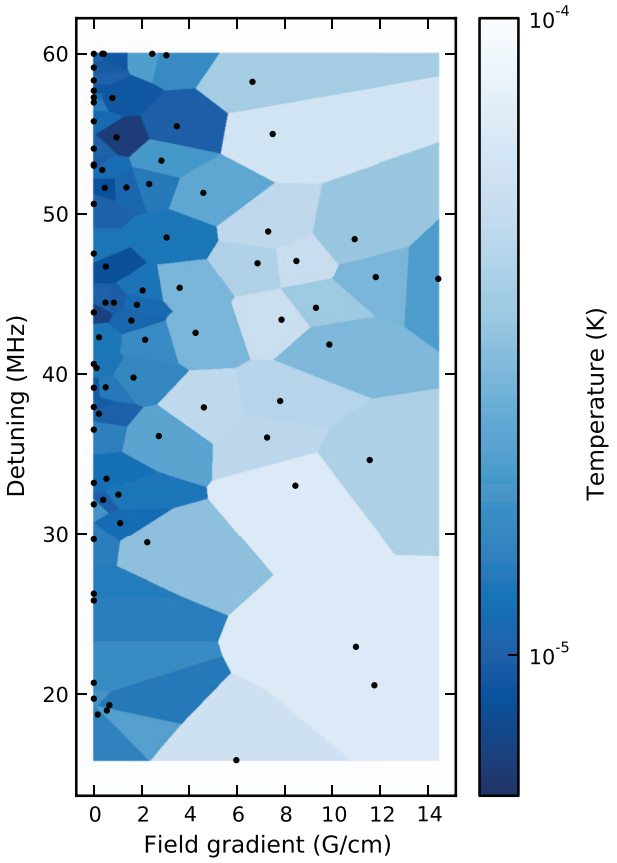


FIG. 9. A proof-of-principle optimization using `mise`. `mise` scanned the parameter space described in Sec. V, searching for the coldest point. Each black point represents a temperature measurement at a specific field gradient and detuning, with the surrounding shading indicating the temperature. Eighty 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[30] 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`,[31] 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 experiments, 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.[32]

The labscript suite is open-source and freely available online.[33] 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.

[1]See, e.g., M. Weidemüller and C. Zimmermann, *Cold Atoms and Molecules* (Wiley, 2009), and references within.
[2]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.
[3]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.
[4]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.
[5]G. Varoquaux, Comput. Sci. Eng. **10**, 55 (2008).
[6]P. E. Gaskell, J. J. Thorn, S. Alba, and D. A. Steck, Rev. Sci. Instrum. **80**, 115103 (2009).
[7]R. P. Anderson, Ph.D. thesis, Swinburne University of Technology, 2010.
[8]M. Beeler, Ph.D. thesis, University of Maryland, 2011.
[9]P. A. Altin, Ph.D. thesis, Australian National University, 2012.
[10]T. Stöferle, Ph.D. thesis, Swiss Federal Institute of Technology, 2005.
[11]A. Keshet and W. Ketterle, Rev. Sci. Instrum. **84**, 015105 (2013).
[12]S. F. Owen and D. S. Hall, Rev. Sci. Instrum. **75**, 259 (2004).
[13]T. Meyrath and F. Schreck, "A laboratory control system for cold atom experiments," see http://www.strontiumbec.com/indexControl.html (2012).
[14]P. Hintjens, *Code Connected Volume 1: Learning ZeroMQ* (CreateSpace Independent Publishing Platform, 2013); see also "ØMQ: the intelligent transport layer," http://www.zeromq.org/.
[15]The HDF Group. Hierarchical data format version 5, 2000-2010. http://www.hdfgroup.org/HDF5.
[16]"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.
[17]G. van Rossum *et al.*, "Python programming language v2.7," see http://docs.python.org/2.7/ (2010).
[18]J. M. Hughes, *Real World Instrumentation with Python: Automated Data Acquisition and Control Systems* (O'Reilly Media, Inc., 2010).
[19]The source code for turning a ChipKIT Max32 into a PineBlaster is available at http://hardware.labscriptsuite.org/.
[20]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.
[21]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.
[22]W. McKinney, "pandas: a Python data analysis library," see http://pandas.pydata.org/.
[23]T. Oliphant, "NumPy: numerical Python," see http://www.numpy.org/.
[24]E. Jones, T. Oliphant, P. Peterson *et al.*, "SciPy: open source scientific tools for Python," see http://www.scipy.org/ (2001).
[25]J. Hunter, Comput. Sci. Eng. **9**, 90 (2007); see also "matplotlib: Python plotting," http://matplotlib.org/.
[26]W. McKinney, *Python for Data Analysis* (O'Reilly Media, Inc., 2012).
[27]T. Bäck and H.-P. Schwefel, Evol. Comput. **1**, 1–23 (1993).

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

[29] A. Berry and P. Vamplew, "PoD Can Mutate: A Simple Dynamic Directed Mutation Approach for Genetic Algorithms," in *AISAT2004: International Conference on Artificial Intelligence in Science and Technology*, 21–25 November 2004, Hobart, Tasmania, Australia.

[30] 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/ and http://www.zeromq.org/bindings:_start/ for more complete lists.

[31] `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.

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

[33] "The labscript suite: an open source experiment control and analysis system," see http://labscriptsuite.org/.

This page intentionally left blank

References

[1] Python Software Foundation. *Python language reference, version 3.7*, (2018). http://www.python.org. [p 1]

[2] National Instruments. *Laboratory Virtual Instrument Engineering Workbench (LabVIEW)*, (2018). http://www.ni.com/labview. [p 1]

[3] N. P. Robins, P. A. Altin, J. E. Debs, and J. D. Close. *Atom lasers: Production, properties and prospects for precision inertial measurement*. Physics Reports **529**, 265 (2013). DOI: 10.1016/j.physrep.2013.03.006. [p 1]

[4] A. D. Cronin, J. Schmiedmayer, and D. E. Pritchard. *Optics and interferometry with atoms and molecules*. Reviews of Modern Physics **81**, 1051 (2009). DOI: 10.1103/RevModPhys.81.1051. [p 1]

[5] A. Negretti, P. Treutlein, and T. Calarco. *Quantum computing implementations with neutral particles*. Quantum Information Processing **10**, 721 (2011). DOI: 10.1007/s11128-011-0291-5. [p 1]

[6] T. D. Ladd, F. Jelezko, R. Laflamme, Y. Nakamura, C. Monroe, and J. L. O'Brien. *Quantum computers*. Nature **464**, 45 (2010). DOI: 10.1038/nature08812. [p 1]

[7] I. Bloch, J. Dalibard, and S. Nascimbène. *Quantum simulations with ultracold quantum gases*. Nature Physics **8**, 267 (2012). DOI: 10.1038/nphys2259. [p 1]

[8] R. Blatt and C. F. Roos. *Quantum simulations with trapped ions*. Nature Physics **8**, 277 (2012). DOI: 10.1038/nphys2252. [p 1]

[9] 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. [pp 2 and 13]

[10] P. Starkey. *A software framework for control and automation of precisely timed experiments*. PhD thesis, Monash University (2018). (in preparation). [pp 2 and 10]

[11] The HDF Group. *Hierarchical Data Format, version 5*, (1997–2018). http://www.hdfgroup.org/HDF5/. [p 3]

[12] C. Klempt, T. van Zoest, T. Henninger, O. Topic, E. Rasel, W. Ertmer, and J. Arlt. *Ultraviolet light-induced atom desorption for large rubidium and potassium magneto-optical traps*. Physical Review A **73**, 013410 (2006). DOI: 10.1103/PhysRevA.73.013410. [p 6]


```
rev:     156 (128ef198499e)
author:  chrisjbillington
date:    Sat Jun 02 17:54:23 2018 +1000
summary: Added history and attribution section
```

[13] J. D. Hunter. *Matplotlib: A 2D graphics environment*. Computing In Science & Engineering **9**, 90 (2007). DOI: 10.1109/MCSE.2007.55. [p 6]

[14] L. Campagnola. *Pyqtgraph, version 0.10.0*, (2016). http://www.pyqtgraph.org. [pp 6 and 19]

[15] W. McKinney. *Data Structures for Statistical Computing in Python*. In S. van der Walt and J. Millman (editors), *Proceedings of the 9th Python in Science Conference*, pages 51–56 (2010). [pp 6 and 11]

[16] F. Perez and B. E. Granger. *IPython: A System for Interactive Scientific Computing*. Computing in Science Engineering **9**, 21 (2007). DOI: 10.1109/MCSE.2007.53. [p 6]

[17] S. P. Johnstone, A. J. Groszek, P. T. Starkey, C. J. Billington, T. P. Simula, and K. Helmerson. *Order from chaos: Observation of large-scale flow from turbulence in a two-dimensional superfluid*. arXiv:1801.06952 [cond-mat, physics:physics] (2018). ARXIV: 1801.06952. [p 8]

[18] P. Gill and P. Baird. *An Optical Lattice Clock with Neutral Strontium*. PhD thesis, University of Oxford (2016). [p 10]

[19] M. Gancarz. *The UNIX Philosophy*. Digital Press, Newton, MA, USA (1995). [p 10]

[20] P. B. Wigley, P. J. Everitt, A. van den Hengel, J. W. Bastian, M. A. Sooriyabandara, G. D. McDonald, K. S. Hardman, C. D. Quinlivan, P. Manju, C. C. N. Kuhn, I. R. Petersen, A. N. Luiten, J. J. Hope, N. P. Robins, and M. R. Hush. *Fast machine-learning online optimization of ultra-cold-atom experiments*. Scientific Reports **6**, 25890 (2016). DOI: 10.1038/srep25890. [p 10]

[21] R. Speirs, (2018). Private Communication. [p 11]

[22] T. E. Oliphant. *Guide to NumPy*. CreateSpace Independent Publishing Platform, USA, 2nd edition (2015). [p 11]

[23] Brian E. Granger and contributors. *Pyzmq, version 17.1.0*, (2018). http://pyzmq.readthedocs.io. [p 11]

[24] Andrew Collette and contributors. *H5py, version 2.8.0*, (2018). http://www.h5py.org. [p 11]

[25] P. Hintjens. *ZeroMQ: The Guide*, (2010). http://zguide.zeromq.org. [p 12]

[26] Zachtronics Industries. *SpaceChem*, (2011). http://www.zachtronics.com/spacechem. [p 12]

[27] C. J. Billington. *Zprocess, version 2.4.12*, (2018). http://bitbucket.org/cbillington/zprocess. [pp 12 and 19]

[28] Monash University School of Physics and Astronomy. *Measurement of β-ray spectra*, (2016). Undergraduate laboratory materials. [p 12]

[29] The Qt Company. *Qt, version 5.11*, (2018). http://www.qt.io. [pp 12 and 13]

[30] P. T. Starkey and C. J. Billington. *Qtutils, version 2.1.0*, (2018). http://bitbucket.org/philipstarkey/qtutils. [pp 12, 19, and 21]

[31] The GNOME Project. *GTK+, version 3*, (2018). http://gtk.org. [p 13]

```
rev:     156 (128ef198499e)
author:  chrisjbillington
date:    Sat Jun 02 17:54:23 2018 +1000
summary: Added history and attribution section
```

[32] Riverbank Computing. *PyQt, version 5.10.1*, (2018). http://gtk.org. [p 13]

[33] The GNOME Project. *GNOME 3.28.2*, (2018). http://gnome.org. [p 13]

[34] Continuum Analytics. *Anaconda Distribution, version 5.1*, (2018). http://anaconda.org. [p 13]

[35] Torsten Bronger, Gregor Thalhammer, Florian Bauer, and Hernan E. Grecco. *Pyvisa, version 1.9.0*, (2018). http://pyvisa.readthedocs.io. [p 17]

[36] Peter Johnson, FRC Team 294. *Pynivision, version 2015.0.0*, (2015). http://pyvisa.readthedocs.io. [p 17]

[37] P. A. Altin, M. T. Johnsson, V. Negnevitsky, G. R. Dennis, R. P. Anderson, J. E. Debs, S. S. Szigeti, K. S. Hardman, S Bennetts, G. D. McDonald, L. D. Turner, J. D. Close, and N. P. Robins. *Precision atomic gravimeter based on Bragg diffraction*. New Journal of Physics **15**, 023009 (2013). DOI: 10.1088/1367-2630/15/2/023009. [p 20]

[38] Monash Univeristy. *The labscript suite*, (2018). http://bitbucket.org/philipstarkey/qtutils. [p 19]

[39] S. F. Owen and D. S. Hall. *Fast line-based experiment timing system for LabVIEW*. Review of Scientific Instruments **75**, 259 (2003). DOI: 10.1063/1.1630833. [p 19]

```
rev:     156 (128ef198499e)
author:  chrisjbillington
date:    Sat Jun 02 17:54:23 2018 +1000
summary: Added history and attribution section
```

This page intentionally left blank

Word count

```
Total
Words in text: 63218
Words in headers: 505
Words outside text (captions, etc.): 10599
Number of headers: 135
Number of floats/tables/figures: 48
Number of math inlines: 1966
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
  9497+81+1478 (26/5/7/0) 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:     156 (128ef198499e)
author:  chrisjbillington
date:    Sat Jun 02 17:54:23 2018 +1000
summary: Added history and attribution section
```