DISS. ETH NO. 24645

A Software System for Automating Lab Experiments with Liquid-Handling Robots

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH

(Dr. sc. ETH Zurich)

presented by

Ellis Whitehead

M.Sc., ETH Zurich

born on 16.06.1975

citizen of Germany

accepted on the recommendation of

Prof. Dr. Jörg Stelling
Prof. Dr. Natalio Krasnogor
Prof. Dr. Sven Panke

2017

Acknowledgements

My sincere gratitude goes to my supervisor Dr. Joerg Stelling for his support and guidance throughout my PhD research. Likewise, Fabian Rudolf and Dr. Hans-Michael Kaltenbach were essential advisors in the topics of biology and statistics, respectively. Prof. Ehud Shapiro and his group at the Weizmann institute, including Tuval Ben-Yehezkel, Ofir Raz, and Ehud Magal, shared resources and experience to help start this project. Furthermore, I'd like to extend thanks to all my colleagues at the Department of Biosystems Science and Engineering, especially those in Joerg Stelling's group for Computational Systems Biology, who provided frequent feedback and inspiration, and Dr. Daniel Meyer and Urs Senn in the robot facility, whose generous technical assistance made hundreds of experiments possible.

Financial support by the EU FP7 projects CADMAD (contract 265505) and ST-FLOW (contract 289326), as well as the National Competence Centre for Molecular Systems Engineering funded by the Swiss National Science Foundation (SNFS) are gratefully acknowledged.

To my wife, children and parents — thank you very much for your unwavering support, love, and patience.

Abstract

This thesis is concerned with the problem of automating liquid-handling robots in synthetic biology, which is especially relevant to facilitate experiments that are difficult or impossible to perform manually. Research tasks that are not highly standardized are still rarely automated in practice. Two main reasons for this are the substantial investments required to translate molecular biological protocols into robot programs, and the fact that the resulting programs are often too lab-specific to be easily reused and shared. Recent developments of standardized protocols and dedicated programming languages for liquid-handling operations addressed some aspects of ease-of-use and portability of protocols. However, they either focus on simplicity but disallow complex protocols, or they entail detailed programming and require advanced skills and efforts from their users.

The contributions of this thesis are the following. In Chapter 1, we introduce the problem and our aims, and we describe the hardware systems and biological methods used in the thesis. In Chapter 2, we describe Roboliq, a software system that (i) handles portable protocols employing a special data structure, high-level commands, and artificial intelligence methods, and (ii) facilitates a tight coupling of the design-execution-analysis cycle by supporting experimental design, simulation, and tidy data output. The chapter investigates the criteria for portability and shows how these can be fulfilled using the proper data structures and high-level commands. It also examines a generic and flexible system for converting high-level commands to the low-level commands required by robots. In Chapter 3, we present three proof-of-principle applications for the reproducible, quantitative characterization of biophysical characteristics of a GFP variant; they demonstrate the system's ability to handle experiments that involve complex pipetting, multi-day measurements and manipulations, and time-critical procedures. In Chapter 4, we present quality control experiments that characterize the pipetting performance of the robot; this is used for troubleshooting, error detection, and bias correction. The experiments are backed by a mathematical model, and we use Markov Chain Monte Carlo simulation to enable the analysis of complex experiments that are not amenable to traditional statistical methods. A selection of diagnostic protocols is also presented for evaluating various aspects of the robotic system and its equipment. Chapter 5 concludes our evaluation of Roboliq.

Zusammenfassung

Diese Arbeit beschäftigt sich mit dem Problem der Automatisierung von Liquid-Handling-Robotern in der synthetischen Biologie. Die Motivation für das Thema ist es, die Labor-Experimente zu erleichtern, die manuell schwierig oder unmöglich sind. Das Thema ist für Forschungsaufgaben, die nicht hoch standarisiert sind, besonders relevant, da sie in der Praxis noch selten automatisiert werden. Zwei wesentliche Probleme behindern deren Automatisierung: Zum Einen ist ein umfangreicher Aufwand erforderlich, um molekularbiologische Protokolle in Roboterprogramme umzusetzen. Zum Anderen sind die daraus resultierenden Programme oft zu laborspezifisch sind, um reibungslos wiederverwendet und geteilt zu werden. Die jüngsten Entwicklungen von standardisierten Protokollen und dedizierten Programmiersprachen für Liquid-Handling-Operationen befassten sich mit einigen Aspekten der Benutzerfreundlichkeit und der Portabilität von Protokollen. Allerdings konzentrieren sie sich entweder auf Einfachheit (und verhindern dabei komplexe Protokolle) oder auf eine detaillierte Programmierung (und erfordern fortgeschrittene Fähigkeiten und Bemühungen von ihren Benutzern).

Die Forschungsbeitrag dieser Arbeit ist wie folgt aufgeteilt. Kapitel 1 stellt die Probleme und die Ziele dar, sowie die in der Arbeit verwendeten Hardwaresysteme und biologischen Methoden. Kapitel 2 beschreibt das Software-Systems "Roboliq". Roboliq verarbeitet portable Protokolle anhand einer speziellen Datenstruktur, High-Level-Befehlen und Methoden der künstlichen Intelligenz. Es wird aufgezeigt, wie dieses Software-System durch die Unterstützung von Experimental Design, Simulation, und "tidy" Datenausgaben auch eine enge Kopplung des Design-Execution-Analyse-Zyklus ermöglicht. Es werden die Kriterien für die Portabilität erläutert, und es wird aufgezeigt, wie diese mit den passenden Datenstrukturen und High-Level-Befehlen erfüllt werden können. Das Kapitel untersucht ein generisches und flexibles System zur Umwandlung von High-Level-Befehlen in die von Robotern benötigten Low-Level-Befehle. In Kapitel 3 werden drei Anwendungen für die reproduzierbare, quantitative Charakterisierung biophysikalischer Eigenschaften einer GFP-Variante vorgestellt. Die Anwendungen zeigen die Fähigkeit des Systems, Experimente mit komplexen Pipettiervorgänge, mehrtägigen Messungen und zeitkritischen Prozeduren durchzuführen. Kapitel 4 stellt eine Qualitätssicherung vor, die die Pipettierleistung des Roboters charakterisiert. Die durchgeführten Experimente sind auf einem mathematischen Modell aufgebaut und werden für die Fehlersuche, Fehlererkennung und Verzerrungskorrektur verwendet. Um die Analyse komplexer Experimente zu ermöglichen, deren Analyse mit traditionellen statistischen Methoden nicht möglich ist, verwenden wir die Markov Chain Monte Carlo Simulation. Zur Bewertung verschiedener Aspekte des Robotersystems und seiner Ausrüstung wird eine Auswahl von Diagnoseprotokollen vorgestellt. Kapitel 5 schliesst die Evaluierung von Roboliq ab.

Contents

Acknowledgements **3**

Abstract **5**

Zusammenfassung **7**

1 Introduction **17**
1.1 Background . . . 17
1.2 Practical and Philosophical Aims . . . 19
1.2.1 Simpler Programming and Reproducibility . . . 20
1.3 Liquid Handling Robots and Laboratory Automation . . . 21
1.4 Experimental Techniques for Manipulating Liquids . . . 23
1.5 Contributions of Thesis . . . 24

2 Roboliq **25**
2.1 Introduction . . . 25
2.2 Results and Discussion . . . 25
2.2.1 Data Structure for Portable Protocols . . . 25
2.2.2 Information integration via automated planning . . . 27
2.2.3 Well and Liquid Selection . . . 29
2.2.4 Simplified programming . . . 30
2.2.5 Processing a Protocol . . . 31
2.2.6 Integrating the Design-Execution-Analysis Cycle with Data Tables . . . 32
2.2.7 Experimental Design Workflow . . . 34
2.2.8 Portability and Merging Data Structures . . . 34
2.2.9 Portability and Automated Planning . . . 37
2.2.10 Portability and its Limits . . . 38
2.2.11 Advanced Protocol Usage . . . 38
2.2.12 Software Implementation . . . 39
2.3 Conclusions . . . 39

3 Proof of Principle Experiments **41**
3.1 Introduction . . . 41
3.2 Biological Background . . . 41
3.2.1 Fluorescence . . . 41
3.2.2 Green Fluorescent Protein (GFP) . . . 42
3.2.3 pH and Acidity . . . 42
3.2.4 Folding and Unfolding of Proteins . . . 43
3.3 Results and Discussion . . . 43
3.3.1 Proof-of-principle applications . . . 43
3.3.2 Complex Pipetting . . . 45
3.3.3 Time Series and Long-Duration Experiments . . . 50

3.3.4 Time-Sensitive Experiments . . . 50
3.3.5 Metadata and Analysis . . . 51
3.3.6 Repeatability . . . 51
3.4 Methods . . . 51
3.4.1 Experimental methods . . . 51
3.4.2 Experimental Limitations . . . 52
3.4.3 Data analysis . . . 52
3.5 Conclusions . . . 54

4 Quality Control of Robot Performance 55
4.1 Introduction . . . 55
4.2 Theory . . . 57
4.2.1 Pipetting: Aspiration and Dispense . . . 59
4.2.2 Pipetting: Volume of Dispense . . . 59
4.2.3 Pipetting: Volume of Undesired Dilution . . . 61
4.2.4 Pipetting: Volume of Well . . . 62
4.2.5 Pipetting: Concentration in Well . . . 62
4.2.6 Weight . . . 62
4.2.7 Absorbance . . . 63
4.2.8 Absorbance of Empty Wells . . . 64
4.2.9 Absorbance of Water-Filled Wells . . . 64
4.2.10 Absorbance of Wells . . . 65
4.2.11 Z-level . . . 65
4.2.12 Computation of model parameters . . . 66
4.3 Results and Discussion . . . 66
4.3.1 Estimates of Pipetting Parameters . . . 66
4.3.2 Estimates of Reader Parameters . . . 68
4.3.3 Correction of Small-Volume Bias . . . 69
4.3.4 Estimates from Supplementary Protocols . . . 69
4.4 Materials and methods . . . 71
4.4.1 Methods Overview . . . 71
4.4.2 Protocol 1: characterization of “standard” pipetting configuration . . . 73
4.4.3 Protocol 2: General test of dispense volumes ($\leq 150\,\mu L$) and variances . . . 74
4.4.4 Correction and Validation Protocol . . . 76
4.4.5 Supplementary and Diagnostic Protocols . . . 76
4.5 Conclusion . . . 80

5 Conclusion 81
5.1 Conclusions . . . 81
5.2 Future Work . . . 82

A Hardware Setup in this Thesis 83

B Robot configuration 87
B.1 Evoware configuration . . . 87
B.2 Components . . . 89
B.2.1 `roboliq` . . . 89
B.2.2 `schemas` . . . 90
B.2.3 `objects` . . . 90
B.2.4 `commandHandlers` . . . 90
B.3 Logic components . . . 91
B.3.1 `predicates` . . . 91
B.3.2 `objectToPredicateConverters` . . . 92
B.3.3 `planHandlers` . . . 92
B.4 Conclusion . . . 93

C Advanced Protocols **95**
C.1 Imports . . . 95
C.2 Parameters . . . 95
C.3 Objects . . . 96
C.4 Data . . . 97
C.4.1 `Data` type . . . 97
C.4.2 `data` property . . . 97
C.4.3 `data.*` commands . . . 98
C.4.4 `data()` directive . . . 99
C.5 Scope . . . 99
C.6 Substitution . . . 99
C.6.1 Scope `$`-substitution . . . 99
C.6.2 Template substitution . . . 101
C.6.3 Directive `()`-substitution . . . 101

D Software Implementation Details **103**
D.1 Well and Liquid Selection . . . 103
D.2 Commands . . . 105
D.3 Core Algorithms . . . 107
D.4 Software Dependencies . . . 107
D.5 JSON Schemas . . . 109
D.6 Input Formats . . . 109
D.6.1 Data Structure for Well Contents . . . 110
D.6.2 Backend Compilers and the `agent` Property . . . 111
D.6.3 Execution . . . 111

E Design Tables **113**
E.1 First examples . . . 113
E.2 Nested branching . . . 115
E.3 Hidden factors . . . 117
E.4 Actions . . . 118
E.4.1 `allocateWells` . . . 118
E.4.2 `range` . . . 118
E.4.3 `calculate` . . . 119
E.4.4 `case` . . . 119
E.5 Step and data nesting . . . 120

F Additional Figures **121**

G Quality Control of Evaporation **123**
G.1 Results . . . 123
G.2 Methods . . . 123

Bibliography **127**

List of Figures

1.1 Roboliq's place in the Scientific Method. 20
1.2 Tecan Evoware Freedom 200 22
1.3 Technical Errors 23

2.1 Structure of Roboliq protocols 26
2.2 Flow diagram for processing a protocol with Roboliq 27
2.3 Example of integrating different sources of information 28
2.4 Well selection phrases 30
2.5 High-level commands and a cascade of sub-steps 31
2.6 Tidy measurements 33
2.7 The role of simulated measurements 33
2.8 Examples of "diffs" 35
2.9 Example of merging using key/value pairs 36
2.10 Using dictionaries to represent arrays 37

3.1 Jablonski diagram of absorbance, non-radiative decay, and fluorescence 42
3.2 Ribbon Diagram of Green Fluorescent Protein (GFP) 43
3.3 pH stability of sfGFP at constant ionic strength 44
3.4 Rate of sfGFP folding 46
3.5 Unfolding of sfGFP 47
3.6 Example of equivalent mixture data tables 48
3.7 Time series graphs for the pH and unfolding experiments 50

4.1 Bias and variance 56
4.2 Bayesian network representation of the pipetting model 58
4.3 Labware models, labwares, and wells 59
4.4 Illustrations of model variables 60
4.5 Illustration of unintended dilution within pipetting tips 61
4.6 Nuisance factors that affect absorbance measurements 64
4.7 Graphs of pipetting performance 68
4.8 Change in absorbance when water is added to empty wells 70
4.9 Small-volume bias correction 70
4.10 Bias and spread of larger dispense volumes 71
4.11 Change in absorbance when water is added to 16ul dye aliquot 72
4.12 Z-level measurements 72
4.13 Operating range of absorbance reader 75
4.14 Spread of absorbance measurements vs shaking time 79

A.1 Labware frequently used in this thesis 84
A.2 Tecan robot arm, degrees of freedom 84
A.3 Measurement equipment used in this thesis 85
A.4 Downholders 86

D.1 Pseudocode for merging objects and loading protocols 108
D.2 Pseudocode for expanding the steps of a protocol 108
D.3 Example of a JSON Schema definition in Roboliq 110

F.1 Design table that generated Table 3.1 122

G.1 Evaporation rates over wells and sites 124

List of Tables

3.1 Excerpt of mixture table from pH experiment. 49

4.1 Estimates of pipetting parameters 67
4.2 Estimates of absorbance parameters 69

D.1 List of Roboliq's standard commands 105
D.2 List of low-level commands for Tecan Evoware 107

Chapter 1

Introduction

If you stop experimenting, you've given up on life!

– Kevin Dunn

1.1 Background

In systems biology, the central idea is that biological systems can be modeled using mathematical and computational frameworks [1]. Biological systems are often represented as networks, where the nodes of the network are biological entities (such as genes, proteins, and cells) that are connected by their interactions [2, 3]. Such networks are complex and partially unpredictable [4], and systems biology attempts to analyze them by developing theoretical models to explain the complete system, whose properties may emerge unexpectedly through the complex interactions of the individual parts [5]. Its many topics of research and application include personalized medicine [6], vaccine development [7], and environmental cleanup [8].

The complementary field is synthetic biology, which builds new systems based on theoretical models and can test how well the results correlate with predicted outcomes [9, 10]. The basic idea is that novel biological systems can be engineered by modifying DNA [11], and one of its philosophical tenets is that constructing such systems can be the most efficient way to validate our hypotheses [12]. Synthetic biologists are diligently testing and expanding our models of cellular function, while simultaneously developing the tools to bring elements of the engineering discipline to biology [13].

Many have recognized the potential for laboratory automation to substantially increase the efficiency, reliability, and reproducibility of the design-build-test engineering cycle in synthetic biology [14, 15]. In comparison to lab technicians, robots can improve upon the precision of pipetting [16] and other operations. Repeatability is also improved because automation robots exhibit much less variability between runs [17] — their timing, precision, and movements tend to be nearly identical each time. The running time of robotic experiments can also be much longer [18] because the robots do not get bored, hungry, tired, or have other responsibilities to attend to (though they do require regular maintenance). Additionally, more complex experiments can be performed [19] because the robot will not get confused and mistakenly transfer the wrong liquid or forget a step.

Although robots bring along their own substantial challenges [20], practical realizations of their potential are starting to emerge now. Examples include automated manufacturing of rationally engineered transcription activator-like effector nuclease (TALEN) variants [21], and automated multiplexed genome engineering in yeast [22]. These and other examples rely on academic or commercial 'biofoundries' [23, 24]: centralized facilities that automate build (e.g., DNA assembly) and test (e.g., circuit characterization) steps. They are modeled on the success of foundries in electronic chip manufacturing where users submit the virtual results of computer-aided design to receive physical constructs and their characterization data in a time- and

cost-efficient manner [25]. In synthetic biology, however, decentralized automation through liquid handling robotics and associated devices in individual laboratories [14] is a plausible alternative because the required investments in automation hardware are far less prohibitive than in chip manufacturing.

Centralized and decentralized models have different tradeoffs in two main characteristics relevant for the practical use of laboratory automation: ease-of-use and flexibility. Users of biofoundries can easily define experimental protocols as sequences of highly standardized tasks, for example, in DNA assembly. Supported by graphical user interfaces (GUIs) [26, 27], protocol definition is intuitive, and it requires neither programming expertize nor knowledge about the 'black box' hardware that will perform the physical operations. Ease-of-use, however, comes with restricted flexibility for a biofoundry's user: it is difficult or impossible to realize non-standard operations, to rapidly develop or optimize protocols in view of technological advances, e.g., in DNA assembly methods [28], and to establish adaptive protocols, protocols that are not simply linear but contain decision points based on measurement data or fault detection mechanisms. Decentralized robots in individual laboratories, in contrast, provide the highest flexibility in principle, but not necessarily in practice. Programming of standard commercial liquid handling robots to define protocols is possible through dedicated GUIs or programming languages. However, both approaches are relatively low-level and vendor-specific — for example, while vectors for movements of robotic arms are pre-programmed, they still need a full specification of every individual liquid transfer. In addition, one may need substantial programming skills to exploit the flexibility associated with the robot not being a 'black box' — and once a potentially complex protocol is established, one cannot easily transfer it to a different machine. This represents a major challenge for decentralized automation in synthetic biology: the diversity of biology requires high flexibility in protocol development and implementation, but currently the effort to set up a protocol in an individual laboratory can only be justified for highly repetitive tasks.

To make decentralized laboratory automation for synthetic biology viable, others [14, 25, 29] and we therefore argue that it is necessary to establish (i) portability of automation protocols, meaning that protocols require at most minor modifications to operate properly in different laboratories with typically different hardware; this implies that many details needed for the execution of an experiment are omitted from a portable protocol, and also that hardware capabilities are compatible and hardware-specific methods defined; (ii) ease-of-use for defining most protocols, standardized ones but also those relevant in more flexible settings such as protocol development; and (iii) flexibility in terms of advanced programming capabilities that enable complex protocols including, for example, real-time adjustments of operations depending on the status of a build process.

Suitable data formats for capturing experimental protocols are a first requirement for achieving these goals. Three main approaches have been proposed. First, extensions of minimum information checklists [30], intended to ensure that reports on experiments contain sufficient information to evaluate data and conclusions, resulted in simple table-based formats for describing experiments, such as ISA-TAB [31]. However, ISA-TAB and similar formats do not contain sufficient information for automation purposes, so they cannot provide standards for describing the steps of a protocol. Secondly, ontologies for experiments such as FuGE [32] and EXACT2 [33] define controlled vocabularies to reduce ambiguity and support the reuse of knowledge; these are important future elements of automated experimentation, but ontologies represent domain knowledge indirectly, which makes it technically and conceptually challenging to develop software around them [29]. Programming languages are a third alternative for protocol definition and automation. For example, the BioCoder [34] and Antha [29, 35] languages allow to precisely specify protocols in a very flexible manner—but these are complex programming languages. Hence, portability, ease-of-use, and flexibility are intimately linked aspects of data and software formats for laboratory automation, with various implicit trade-offs.

For synthetic biology, many automation solutions exist that are tailored to a particular laboratory, machine, or protocol [34, 36–38]. Regarding more general solutions, PR-PR [14, 39] is arguably the reference for a programming language that aims to reconcile portability and ease-of-use. It is high-level in the sense that the user defines protocol steps, not individual machine actions, in a script; a compiler then uses the script to generate specific instructions for the machine in its lower-level language, optimizing, for example, sequential liquid transfers on the way. PR-PR's design thereby enables aspects of portability (because different compilers can translate a protocol to different machines), of ease-of-use (through more intuitive high-level programming), and flexibility (for example, by importing modules for additional devices). The design, however, has substantial disadvantages. PR-PR is not a *complete* programming language. While the

high-level scripting makes protocol definition accessible to more users, it restricts the complexity of protocols one can define—conditions, feedback, or complex data processing are not possible [29]. In addition, while portability between physical devices has been demonstrated [39], PR-PR protocols do not automatically adapt to new platforms and their physical or operational limitations, and custom data structures similar to minimal information lists make information re-use difficult.

To help reduce the substantial tradeoffs between portability, ease-of-use, and flexibility in current solutions for laboratory automation, we here present a system for the programming of liquid-handling robots called Roboliq (a combination of "robotic" and "liquid"). We based its design on three key insights: First, one can easily merge information from multiple sources by using an appropriate data structure, which allows us to combine high-level, portable protocols, machine configurations, and user input seamlessly. Second, as proposed in [28], methods from Artificial Intelligence (AI) can automatically supply many details omitted in portable protocols by logically reasoning about the possible ways to fulfill tasks, and then to optimally plan the actual experimental implementation under operational constraints. Finally, one can achieve full flexibility to also realize complex protocols by interfacing with a complete programming language. More specifically, we developed a data structure and software pipeline to transform portable protocols to low-level robot instructions. Most of the transformation is automated by methods from the field of Automated Planning (a sub-discipline of AI) [40], with additional queries made to the user for any specifications that cannot be inferred or planned automatically.

1.2 Practical and Philosophical Aims

Scientific progress can be made on various fronts, such as making discoveries and developing new theories, methods, and tools [41]. In this context, Roboliq is a tool to make automated experiments with liquid-handling robots more efficient.

Experiments are so important because they are one of the few fundamental ways we have to acquire knowledge. Experimenters perturb the world and evaluate its response [42]. And whether looking for a faster route to work or trying to optimize cell growth characteristics, we are all experimenting. Ironically, we are not very good at it — upon reflection, this is because it involves many advanced skill sets, including design, execution, and analysis [43].

Roboliq's place within the scientific method is illustrated in Figure 1.1. After formulating an interesting question and deciding that an experiment is the best way to answer it, there is a clear chronological progression through the method: design the perfect experiment, perform it, analyze the measurements, draw conclusions, and share the results with others.

Mentally, however, one needs to manage the complex interplay among these stages. Roboliq plays a role in the six highlighted elements Figure 1.1:

- Execution: we aim to simplify the programming of reproducible experiments.
- Sharing: we aim to provide a portable way to specify protocols that can be shared between labs.
- Design → Execution: we aim to let users build their protocols upon design specifications.
- Execution → Analysis: we aim to output measured data that includes all design factors to facilitate analysis.
- Execution → Design: we aim to quantify the robot's pipetting performance and let this knowledge inform design decisions.
- Analysis → Design: we aim to simulate execution measurements so that analysis can validate the design's statistical power before execution.

This thesis focuses on the stages of design, execution, and analysis. Ideally, these stages should be well coordinated, where the design is based on both execution capabilities and analysis goals; execution should follow logically from the design, and the initial data import for analysis should be straightforward.

In practice, the links between these stages are typically very weak (see [44] for a treatment of common pitfalls). For example, execution often involves many factors that were not in the design, and the analyst often has to

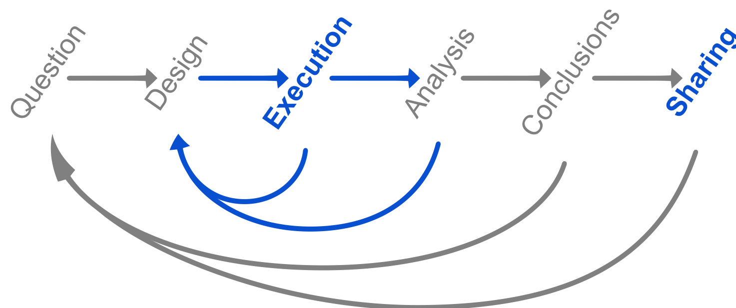


Figure 1.1: **Roboliq's place in the Scientific Method.** This is a sketch of stages of the scientific method. Important stages include question formulation, literature review, hypothesis formulation, experimental design, experiment execution, data analysis, conclusions, communication, peer review, and replication (not all of which are illustrated here). Chronologically, one progresses through the scientific method from the left (Question) to the right (Sharing), as indicated by the straight arrows. But there are also many feedback loops that are often overlooked, as indicated by the curved arrows. Roboliq is involved in the blue-colored elements and is especially embedded in the design-execution-analysis cycle.

manually associate each measurement with its design factors. In the initial stages of my PhD work, I also encountered these problems in my own robot experiments. In order to help us run better experiments, there was a clear need to strengthen the links between the stages (see Sections 2.2.6 and 2.2.7).

1.2.1 Simpler Programming and Reproducibility

We want to make it easier to program experiments relative to vendor-provided interfaces, and this is discussed in Section 2.2.4 and Chapter 3. More importantly, we want to make it easier to program *useful* and *reproducible* experiments, because an experiment whose results cannot be properly analyzed and reproduced is usually worthless, having wasted time and resources without producing real information. The "crisis of non-reproducibility" has been widely discussed in recent years, along with the generally poor design and analysis of experiments in many fields [45–52]. Indeed, there are strong arguments that most published conclusions are false [53]. Roboliq was designed with these issues in mind, and we strove to support experimenters in their pursuit of reproducibility.

There are multiple types of reproducibility. The first type is *reproducible analysis*, but even this is difficult to achieve without the right tools. It is often not possible to reproduce the analysis of published studies, especially for older ones where the original data is either no longer available, on inaccessible storage media, or in old software that can no longer be obtained [45].

A fully reproducible analysis uses programming code to transform raw data into figures and tables and produces a report, by following these steps: 1) Code processes the raw data and metadata into a "tidy" form for analysis [54]; 2) Code performs analysis on the data; 3) Code incorporates analysis into a report.

The advantages are significant. Accuracy is enhanced because modifications cascade through analysis. Clarity is enhanced because explicit instructions are used rather than a possibly vague description of analysis. Trustworthiness is enhanced because it gives others confidence in the figures and tables. And extensibility is enhanced because the analysis can be easily extended later by oneself or others.

We made a concerted effort to ensure that Roboliq can save its measurements already in "tidy" form. These topics are discussed in further detail in Section 2.2.6.

The second type of reproducibility is obtaining similar results when re-running your own experiments. Chapter 3 offers some practical examples of such reproducibility, and Chapter 4 focusses on quality control to help troubleshoot instances where the robot's results are inconsistent.

The third type of reproducibility is achieved when others can run your experiments in their labs and obtain similar results. This is a very important topic, but it is partially outside the scope of this thesis and belongs to future work. One way that Roboliq does help is by standardizing the protocol format. As discussed in [55], protocols written in prose form contain a lot of implicit knowledge and rely on unspecified lab conditions, and it can be very difficult for another lab to figure out how to successfully run the experiment. Even published protocols suffer from this problem [56]. Roboliq's approach to this topic is discussed in further detail in Section 2.2.8.

1.3 Liquid Handling Robots and Laboratory Automation

The first industrial robot was arguably designed in 1937 [57], and the first patent for an industrial robot was granted in 1961 in the USA [58]. By the 1970s, the industry was growing rapidly in many countries around the world, leading to the first liquid handling robots [59]. Today many companies and laboratories have developed automated systems for liquid handling. These systems commonly include pipetters to transfer liquids, labware to hold the liquids, liquid manipulator (such as vacuum filters, centrifuges, and shakers), robotic arms or conveyors to move labware around, measurement equipment, and of course the computers and software to control execution.

The experiments in my research were run on four Tecan Evoware robots. The primary robot is pictured in Figure 1.2, and further details of our setup can be found in Appendix A.

Labware. The most important types of labware for our experiments are microwell plates, tubes, and troughs. The tubes and troughs often hold source liquids, whereas most of the mixing tends to be done in microwell plates.

Robotic Arms. Robotic arms are used for moving labware (especially plates) around the workspace.

Robotic Pipetter. Robotic pipetters are used to transfer and mix liquids. Liquid handling robots typically use a motorized pump and syringe to aspirate and dispense liquids from the pipetting tips. They may also be able to perform z-level detection (i.e. detect the liquid level in a well) and vibrate to shake off droplets that may stick to the tip. Pipetter systems can be water-filled or air-filled and have fixed tips or disposable tips. The advantage of water-filled systems is greater accuracy due to water's lower elasticity; the disadvantage is higher maintenance requirements and the possibility of diluting or contaminating transferred aliquots with the system water. The advantage of fixed tips is significantly improved accuracy, lower operating costs, and greater space efficiency (no need to put replacement tips on the bench); the disadvantage of the fixed tips is that they often need to undergo rinsing or decontamination between pipetting operations, which can be time-consuming. In this thesis, the primary robot has water-filled syringes with fixed tips and z-level detection.

Measurement Equipment. Measurements are a crucial element of experimental work. In this thesis, we will be concerned with measuring fluorescence, absorbance, weight, and liquid level. An advantageous property of the optical measurement equipment (for fluorescence, absorbance and OD readouts) is the speed with which 96 or 384 wells can be individually measured. A scale, in contrast, can measure an entire plate, but not the individual wells. Such characteristics need to be considered when designing experiments.

Manipulating Equipment. Most experiments in synthetic biology require additional types of manipulators, for which external equipment is required. Examples include mixing (e.g. with a shaker or vortex), separating (e.g. with a centrifuge or vacuum filter), and temperature control (e.g. with a refrigerator, incubator, or PCR machine). Air exchange may also need to be controlled by application or removal of a seal.

Technical Challenges Due to Lack of Sensor Feedback. Our Tecan robots are based on an outdated design philosophy: that the robot and its environment behave exactly as expected. Such a robot has very few sensors to determine the true state of the world, so it goes through its instructions just assuming that everything is OK [60]. This inevitably leads to absurd behavior sometimes, including such things as:

- aspirating from empty wells

Figure 1.2: **Tecan Evoware Freedom 200.** (Top) Top-view of the robot's pipetting area. From left-to-right: the washing station; tube holders; a large white trough; three 100 mL trough holders (empty); then 8 plate holders and an Eppendorf tube holder. The plate holder with blue edges is also a shaker. (Bottom) Front-view of the robot. From top-to-bottom: three robotic arms; the bench area with labware holders, two red thermocyclers, and a silver optical reader; system liquid supply, liquid waste containers, vacuum controller, and centrifuge.

Figure 1.3: **Technical Errors.** (A) All wells should be the same, but we see different volumes and concentrations due to mixing of system water with the well contents during mixing; we were able to partially solve the problem after discussions with Tecan, but eventually concluded that we could not use the pipetter to mix plate contents by aspirating and re-dispensing the contents. (B) A critical misplacement results in the transporter cracking the plate. (C) The plate has popped out of place after the pipetter retracts.

- leaking syringes dripping randomly into wells
- droplets sticking to the tip rather than being fully dispensed in the well
- droplets shooting out of the well due to excessive dispense speed
- dispensing at a location where there is no plate
- mixing of system water with well contents (Figure 1.3A)
- wrongly gripped plates being dropped during transport
- gripper not grabbing the plate at all yet continuing through the motions
- misplacing plate after transportation (Figure 1.3B)
- improper peeling of plate
- plate sticking to tip after aspirating from a sealed plate (Figure 1.3C)

This design philosophy produces what we might call “stupid robots”, since they cannot respond to changes in their own operating state or their surroundings. A new generation of robot designs should be able to take advantage of better feedback and of modern computer vision algorithms [61–63]. But for the purpose of this thesis, we need to minimize the likelihood of these errors through additional testing, quality control, and experience.

1.4 Experimental Techniques for Manipulating Liquids

The experiments in this thesis apply a basic set of techniques for manipulating well contents.

Creating a mixture. The most basic manipulation involves transferring an aliquot from one well to another. By transferring different sources into a single well, we are able to create a mixture.

Ensuring well-mixedness. It is often important to ensure that mixtures are well-mixed – we found that small aliquots do not mix quickly by themselves, so special action must be taken. The three main techniques applied in this thesis are 1) using only large volumes, 2) dispensing a large volume after the small volume, or 3) shaking the plate.

Preventing evaporation. In long-running experiments, it may be necessary to minimize evaporation. Sometimes it is sufficient to simply have the robot place a lid on the plate after pipetting operations. Otherwise, we used the integrated sealers to apply an adhesive seal to the plate. The advantage of lids vs seals is that they are more easily removed and do not require extra sealing equipment; the disadvantage is that they are not as effective and require bench space.

Removal of condensation. In long-running experiments with sealed plates, it can be important to remove

condensation from the seal, especially before making optical readout or working with very small volumes. In our experiments involving proteins, we briefly centrifuged the plates to remove condensation.

Temperature control. Due to the temperature-dependence of biochemical reactions, it is sometimes necessary to cool or heat labware. For the experiments in this thesis, we used the temperature control in our centrifuge and optical reader.

1.5 Contributions of Thesis

The primary aim of this research was to facilitate portable, reproducible lab automation with liquid-handing robots. To this end, we developed the relevant concepts and implemented a software system.

The concepts and implementation are described in Chapter 2, where we investigate the criteria for portability, high-level programming, and reproducibility. The contributions of this chapter follow logically from the criteria: (i) a data structure for protocols that can be split into multiple files so the portable and non-portable parts of an experiment can be specified separately; (ii) the usage of artificial intelligence to enable concise, high-level commands that adapt to the low-level requirements of different labs; (iii) a software system to process the protocols and produce executable robot programs; and (iv) software features to help tightly couple the design-execution-analysis stages of experimental science.

We validated the system in Chapter 3, where we present three proof-of-principle applications to demonstrate Roboliq's benefits for experiments that are difficult to perform manually because of their complexity, duration, or time-critical nature.

Chapter 4 contributes a mathematical model of pipetting performance, as well as a set of sophisticated quality control protocols and their analyses. The quality control protocols characterize a robot's pipetting performance, support maintenance, detect errors, and correct pipetting bias. For one of our robots, we discuss the results in detail.

Finally, Chapter 5 concludes the thesis.

Chapter 2

Roboliq

2.1 Introduction

Roboliq aims to make it easier to use liquid handling robots for automation in biological laboratories. It integrates well into the design-execution-analysis cycle of experimentation, provides a format for writing formal protocols, helps make protocols portable between labs, simplifies some aspects of robot programming, and compiles the protocols for execution by liquid handling robots.

To simplify programming, Roboliq was designed to gracefully handle high-level commands. By keeping track of the state of equipment and labware, Roboliq commands can automatically open and close equipment when necessary, figure out when to move plates around, and perform complex pipetting procedures with just a few lines of code.

To support portability, Roboliq was designed to access low-level details outside of the protocol. A *portable protocol* can be viewed from two perspectives: 1) a protocol that can be run in different labs, or 2) a protocol that does not rely on any lab-specific information that would prevent it from running in a different lab. The first perspective is the goal, and the second perspective is a necessary condition. Our solution was to use a data structure that: 1) can represent both protocols and lab configurations, and 2) can be seamlessly split and merged, so that independent protocol and configuration files can be joined to contain all the information necessary to run the protocol in a given lab.

In this chapter, we will discuss the aspects of Roboliq's design, features, and implementation that were most important for our research aims. We begin with the software design and protocol format to give an overview of how the software system functions, followed by a discussion of the aspects of robot programming that Roboliq simplifies. The next two sections delve into Roboliq's role in the design-execution-analysis cycle and how it can facilitate an efficient workflow for complex experiments. The following sections discuss how portability is achieved and its limits. The chapter then concludes with sections on advanced protocol usage and relevant implementation details.

2.2 Results and Discussion

2.2.1 Data Structure for Portable Protocols

Protocols essentially specify a set of objects and a sequence of tasks to perform on those objects. Specifying *portable* protocols involves a paradox, however, because portability implies the omission of many details that are required to actually execute an experiment, such as knowing the robot's bench layout and choosing the labware. It is therefore critical to employ data structures that can integrate information from multiple sources—one of which is the portable protocol, while other sources supply the missing information.

```
roboliq: v1

description: |
  Example of a protocol specification.
  We prepare balance plate for certifugation by
  filling it with water, sealing, and then moving
  it to the centrifuge.

objects:
  trough1:
    type: Labware
    description: container for water
  balancePlate:
    type: Plate
    description: balance plate for centrifugation
  water:
    type: Liquid
    wells: trough1(all)

steps:
  1:
    command: pipetter.distribute
    source: water
    destinations: balancePlate(all)
    volumes: 70ul
  2:
    command: sealer.sealPlate
    object: balancePlate
  3:
    command: transporter.movePlate
    object: balancePlate
    destination: CENTRIFUGE_BAY3
```

Software version (required)

Descriptive information on the protocol (optional)

Objects used in the protocol in a set or hierarchy of key/value pairs (required)

Each object has a name, a type, and additional property values.

List of operations to be performed in the protocol, organized in steps (required)

Commands have parameters and sub-steps are numbered.

Figure 2.1: **Structure of Roboliq protocols.** The example protocol (left) prepares a balance plate for use in centrifugation. The `roboliq` field requires a version number for the data structure to define which version of the specification the protocol was written for. The `description` field holds an optional free-form text to annotate the protocol. The `objects` field contains the protocol's named objects, which may be physical materials or abstract data. For example, `balancePlate` is the name of a plate object with two of its own fields: `type` (required) specifies the type of object and `description` (optional) holds documentation text. The `steps` field contains the protocol's steps, which should be numbered. They can include descriptions, sub-steps, and commands. Here, a single top-level step (without a description) has three sub-steps. The first sub-step for the `pipetter.distribute` command requires several parameters: `source` specifies which liquid to distribute, `destinations` defines the wells to distribute to (all wells on `balancePlate`), and `volumes` instructs the robot to distribute 70 µL to all destination wells.

To represent objects and actions, we wanted a standard data format to facilitate interfacing with external software. Additionally, the format should accommodate complex protocols, be human readable, and allow for concise protocol specifications. We chose JSON (JavaScript Object Notation) [64], a *de facto* modern standard for data interchange on the web that is also used in Transcriptics' Autoprotocol language and related GUIs [26]. JSON data structures are simple to create, read, and edit in all popular programming languages. Roboliq also supports the closely related YAML format [65]; it is easier for humans to read and write, which is why we use it here for all illustrations.

Figure 2.1 shows an example of a Roboliq protocol that fills a balance plate for centrifugation and places it into a centrifuge. Roboliq protocols are organized in sections, and the example illustrates the use of four of them. A mandatory `roboliq` section specifies the version of Roboliq the protocol was written for, to allow for backward compatibility. A `description` section is optional; it contains free-form comments about the protocol. An `imports` section (not shown in Figure 2.1) holds an array that lists the external modules required by a protocol. Roboliq loads such external JSON, YAML, or JavaScript modules before processing the protocol's remaining fields to allow the re-use of already created sub-protocols. The `objects` section defines a hierarchy of key/value pairs for objects used in the protocol. These may include physical materials, such as chemical substances and labware, and abstract data, such as a list of volumes for pipetting. Each object has a name, a type, and additional property values. In the example in Figure 2.1, the first object is named `trough1`, it is a `type` of `Labware`, and it has a `description`. The `steps` section defines the protocol's operations on the objects. It has several optional keys such as a `description` to document the step's purpose.

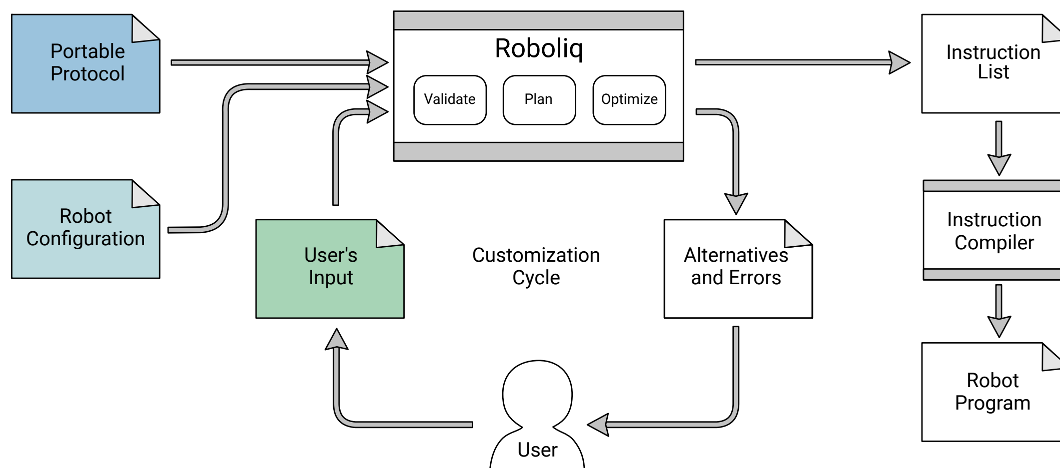


Figure 2.2: **Flow diagram for processing a protocol with Roboliq.** Inputs to Roboliq are shown as colored rectangles, intermediary results as white rectangles, and gray bars indicate software modules.

The `command` key identifies a command that a step should execute. Most commands have parameters, which should also be included in the step definition. Steps may have numbered sub-steps that share the basic structure. An example of a step is in Figure 2.1, where the first of the three sub-steps instructs the liquid handling device to distribute 70 µL of water to all wells on the balance plate.

Such portable protocols clearly leave many questions open that are critical for the protocol's execution: Where are the trough, plate, and handling devices located on the robotic workbench? Should the water be distributed sequentially or in parallel, into how many wells, and by which physical device? Such information needs to be integrated with the portable protocol, and our data structure based on nested key/value pairs allows for seamless information integration.

2.2.2 Information integration via automated planning

Roboliq is built around the central idea that a portable protocol can be automatically combined with more lab-specific data to produce an executable automation program according to the scheme in Figure 2.2. Because portable protocols lack the low-level details required for automation, we extract this information from two sources: the robot configuration file and the user's input (see below). The robot configuration is a file that specifies a robot's capabilities, bench layout, and logical setup for planning tasks — it is protocol-independent and only has to be established once per robot. Roboliq validates each step of the protocol. If a step encodes a high-level command, automated planning can find methods to complete the task as well as a method's parameter values required for a valid command. For pipetting procedures, the sub-steps are optimized for efficiency. As part of the customization cycle, Roboliq outputs a list of any errors for correction and any missing information for the user to supply. Also, Roboliq may list possible alternatives to its automated choices; the user can then override them or otherwise tailor the final results. With an error-free protocol, Roboliq generates a complete instruction list as input to the instruction compilers that produce low-level robot programs for execution.

Hierarchical Task Networks (HTNs) are Roboliq's core method for automated planning. In general, the HTN approach reduces high-level tasks, such as protocol steps, to a network of lower-level tasks, such as unit operations using specific methods and devices. Automated planning algorithms then consider constraints on operations and resources as specified in the methods to yield detailed schedules of tasks through optimizations on the task network [40]. In Roboliq, automated planning relies on an established algorithm [66] that represents states and actions; through forward simulations, it supports inference, evaluation of alternative methods, and first-order predicate logic for reasoning about tasks and actions.

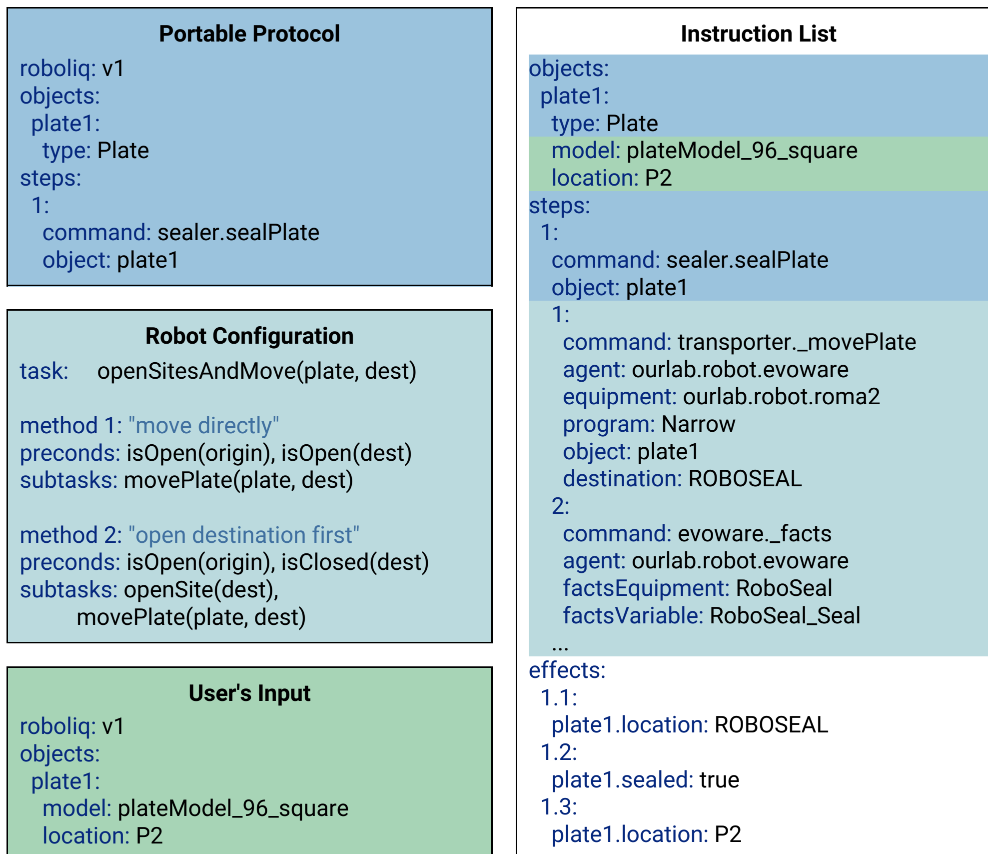


Figure 2.3: **Example of how Roboliq integrates different sources of information (left) to generate a specific instruction list (right) for the sealing of a plate.** Colors for inputs correspond to those in Figure 2.2, and coloring in the instruction list indicates the origin of the respective instruction. The blocks for the robot configuration and the instruction list show only relevant excerpts.

Figure 2.3 shows an example of how Roboliq integrates inputs and performs planning to translate a high-level portable protocol into a low-level instruction list. The portable protocol defines a plate (`plate1`) and a single step to seal the plate. The robot configuration input contains the specification of a task "openSitesAndMove" that will move a plate to a destination site, while first ensuring that both the origin and destination are open (a site can be closed, for example, when inside a centrifuge). Two method definitions follow. The first method has the precondition that both sites are already open. If the condition holds, then the task gets decomposed into the listed subtasks, here to move the plate directly to the sealer. Otherwise, the next method is checked, and if none of the methods meet their preconditions, Roboliq issues an error. The user's input block contains settings for the plate model and its initial location; during the customization cycle, Roboliq informs the user about the missing information, which the user can then fill in by entering the values in a YAML file such as the one shown in Figure 2.3. In the resulting instruction list, the portable protocol provided the plate object and sealer command; the user defined the plate model and bench location; and the robot configuration provided the logic for sealing the plate using concrete low-level commands. Note that a different robot configuration or initial robot state could result in different sub-steps in the instruction list. Here, the planner inserted the first sub-step (`transporter._movePlate`) because it recognized that the plate must be moved to the sealer site before sealing. The second sub-step (`evoware._facts`) is a low-level command for the target robot to run the sealer, as specified in the robot configuration. Finally, the effects section of the instruction list contains simulation data that allows us to follow state changes step-by-step. Overall, this protocol illustrates how automated planning enables substantial gains regarding protocol readability and portability. The gains are even more pronounced for the more complex applications below, such as using automated expansions of high-level steps to generate dilution series.

Roboliq's features covered so far focused on portability and ease-of-use, where JSON data structures are employed for data processing and exchange. Several arguments speak against more complex formats, such as programming languages like BioCoder [34]: programming language-based protocols limit automated validation to prevent crashes or infinite loops at execution, and they are nearly impossible to integrate with third-party software for creating, analyzing, and modifying protocols. However, to enable flexibility as discussed above, programming is also a necessary component of automation. To accommodate this, Roboliq supports the use of JavaScript as well as several advanced fields in the protocol structure (see Section 2.2.11). We chose JavaScript because all major web browsers have a JavaScript programming environment − this currently makes JavaScript the most accessible programming language, guarantees strict security, and facilitates the development of custom user interfaces. Note that Roboliq's JavaScript support is primarily intended for the robot configuration file, allowing technical specialists or advanced users to adapt generic commands to a specific machine in sophisticated ways. This design also makes it possible to modify and extend Roboliq's command set in a modular manner and to exchange modules between laboratories.

2.2.3 Well and Liquid Selection

When working with liquid handling protocols, one often needs to specify sets of wells and liquids. There are three things to consider: well selection, well ordering, and liquids that are in multiple wells. Well selection involves concisely specifying the desired wells on the desired labware. Well ordering involves whether to move across a plate by row or column first, whether to randomize the order, and whether certain spacing constraints apply (to optimize pipetting in cases where the pipetters are too large to access neighboring wells simultaneously). Liquid well selection involves allowing the use of any wells for that source (such as having multiple tubes of buffer source) so that the pipetting algorithms can pick the optimum ones.

Roboliq has a parser that accepts a string and returns a list of wells and liquids. Labware and liquids are indicated by name, and the wells on a labware are specified in parenthesis after the labware. The `+` symbol concatenates multiple labware and liquids. Detailed examples and the parser grammar are listed in Section D.1.

Examples of well selection phrases are illustrated in Figure 2.4. The well specification starts with `all` or a well name, and a well name is formed by a letter and number, where the letter is the row and the number is the column. For example, a 96-well plate with 8 rows and 12 columns has wells `A1` through `H12`, and a 384-well plate with 16 rows and 24 columns has wells `A1` through `P24`. The well specification can have

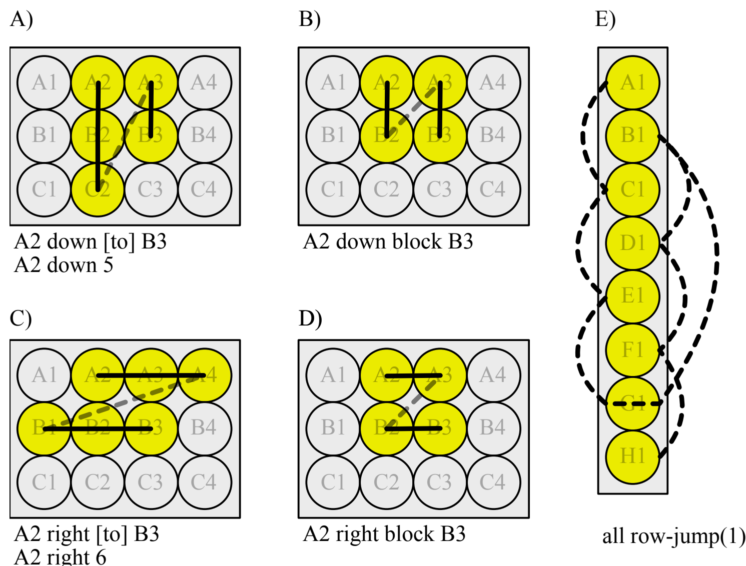


Figure 2.4: **Well selection phrases.** Each panel shows a labware and a well selection phrase. Panels A-D show 3x4-well plates, and panel E shows an 8x1 tube holder. Panels A and C each list two selection phrases that are equivalent. The yellow wells are the selected ones, and the lines show the order of selection. A) Selection order is A2, B2, C2, A3, B3. B) Selection order is A2, B2, A3, B3. C) Selection order is A2, A3, A4, B1, B2. D) Selection order is A2, A3, B2, B3. E) Selection order is A1, C1, E1, G1, B1, D1, F1, H1.

multiple elements separated by commas (see example 5 in Section D.1). To specify a set of adjacent wells, one can use any of these phases:

- <well1> down|right [to] <well2>
- <well1> down|right <count>
- <well1> down|right block <well2>

where <well1> is the starting well, <well2> is the final well, and <count> is the number of wells to select. One of the direction indicators, down or right, must be used in the phrase, but the to preposition is optional. The down direction means that wells will be selected by first moving down the column from <well1> until <well2> or <count> are reached; if the bottom of the plate is reached, the selection continues in the next column. Analogously, the right direction selects rightwards from the starting well. The down block and right block phrases are used for selecting a rectangular set of wells; well1 specifies the upper left well of the selection, and well2 specifies the bottom right well.

Two phrase modifiers are also available:

- random(<seed>): randomizes the order of the selected wells using the randomization seed <seed>.
- row-jump(<size>): staggers the row selections by jumping <size> rows at a time.

We have found this setup to accommodate all of our use-cases so far.

2.2.4 Simplified programming

Programming liquid handling robots can be unexpectedly difficult for users. In the case of Tecan's control software *Evoware Standard*, they provide a simple "visual programming" environment that allows non-programmers to create protocols. Although the environment is adequate for certain applications, it has various shortcomings, including: 1) Evoware commands are very low-level and provide little protection from even serious errors (e.g., it will try to pipette from a plate even after putting a lid on); 2) It does not keep

```
1:
  command: absorbanceReader.measurePlate
  object: plate1

    ⇘

      1:
        command: transporter.movePlate
        object: plate1
        destination: ourlab.mario.site.READER

          ⇘

            1:
              command: equipment.open
              equipment: ourlab.mario.reader

                ⇘

                  1: ...
```

Figure 2.5: **High-level commands and a cascade of sub-steps**. This figure illustrates Roboliq's simple mechanism for high-level commands. Here the `absorbanceReader.measurePlate` command generates sub-steps, the first of which moves the plate to the reader via the `transporter.movePlate` command; That command, in turn, generates more sub-steps, starting with `equipment.open` to open the reader.

track of the state of an experiment (e.g., whether a device is open, where the plates are on the bench), so this knowledge is only in the programmer's head, inevitably leading to mistakes in long protocols; 3) A particular bench location can only be assigned to a single labware model during a script, and to change a labware model, one has to first delete all commands that use that site, then change the labware model, and finally, recreate all the deleted commands. From the perspective of Software Engineering, Tecan's software fails many common criteria for well-designed systems, and it exhibits many deficits in user-friendliness, functionality, and stability. Some other manufacturers appear to have designed better programming languages than Tecan, but the high precision of Tecan's liquid handling hardware provides a counter-balance to their software deficits.

Roboliq was designed to make programming simpler using high-level commands that know the state of the experiment. This is accomplished by plugging in *command handlers* that can themselves output new sub-steps for Roboliq to subsequently process. Programmers can create commands in a very modular fashion: their command handlers can do any processing required and emit any required sequence of sub-commands. For example, as illustrated in Figure 2.5, the command for measuring absorbance checks whether the plate is already in the reader, and if not it emits a sub-step to transport it there. The transport command, in turn, checks whether the reader is already open, and if not, it emits a sub-command to open the reader prior to transport.

In principle, this is very similar to building a library of functions in a standard programming language, where some functions call other functions. The difference is that high-level command handlers return a list of sub-steps rather than calling them directly, and Roboliq handles further processing and tracks the state changes.

2.2.5 Processing a Protocol

Roboliq relies on a small library of core algorithms, helper functions, basic parsers, and glue code to bind the system together. The core algorithms are `mergeObjects`, `loadProtocol`, and `expandSteps`, whose pseudocode is shown in Figures D.1 and D.2. The `mergeObjects` algorithm merges two JSON/JavaScript objects together so that information can be combined from the robot configuration, the portable protocol, the user's choices, and any imported modules. The `loadProtocol` algorithm loads the given protocol into memory, recursively loads its requirements, and then merges all loaded protocols. Finally, the `expandSteps` algorithm is where most of the work takes place: it starts with the fully merged protocol and iterates over

each step. For steps that specify a command, it calls the appropriate command handler with the current state and logic data, and then recursively calls `expandSteps` on the command handler's output to handle dynamically generated sub-steps.

When processing protocols, most of the complexity lies in the command handlers, which may need to figure out how to fill in missing information. Because command handlers receive the state of protocol objects and logical predicates, they can make use of algorithms from Automated Planning to find methods and parameters to complete their tasks. For this purpose, Roboliq includes a SHOP2 [66, 67] implementation of the Hierarchical Task Network planning approach [68] that supports sophisticated logical queries and planning.

2.2.6 Integrating the Design-Execution-Analysis Cycle with Data Tables

As discussed in Section 1.2, a major aim of Roboliq is to help tightly integrate the design, execution, and analysis stages of an experiment. The construct that enables this is the *data table*.

By formulating the experimental design as a table, Roboliq can 1) adapt command execution accordingly, 2) automatically produce "tidy" measurement outputs with measurement values matched to their corresponding design factors, and 3) produce simulated measurement output so that the analysis methods can be tested before execution.

The *tidy* data table is conducive to efficient and reproducible data analysis [54]. Each row of a tidy table represents one observation, and each column represents a design factor or measured value. The tools for analyzing and visualizing such datasets are very powerful [69].

Roboliq's data tables are made of rows and columns. Internally, these are stored as a JSON array of objects, where each object represents a row, and each object property represents a column. Data tables can be constructed in one of four ways:

1. Write the table in JSON format.
2. Write a *design specification*, which is a concise format for specifying potentially large and complex data tables (See Appendix E).
3. Generate the table as part of a JavaScript input.
4. Generate the table externally (e.g., in R), and load it into Roboliq.

Roboliq allows commands to access an entire table at once or to select subsets of the table (using the `where` query of the `data` command property), and the column values of the selected rows are available for use as variables. It also provides commands to loop over individual rows (`command: data.forEachRow`) or over groups of rows (`command: data.forEachGroup`). This facilitates tightly integrating experiment design and execution by binding command arguments directly to the design. Examples are provided in Appendix E.

Roboliq produces tidy measurement outputs by joining raw measurement data with the selected design rows. It then saves the measurements along with all the corresponding design factors. (More technically, it performs a *left join* [70] between the raw measurement data and selected rows by a factor such as the measured well or labware.) The saved output is in JSON format and can be read by any modern programming language for immediate analysis. Normally, this completely avoids the laborious data wrangling process [71, 72], or at least simplifies it to just renaming and reformatting some columns.

Roboliq also supports measurement simulations by specifying a formula for the property `output.simulated` (in `mathjs` format). This can either be a calculation or the name of a column in the data table that was designated for simulated measurement values. When Roboliq compiles the protocol, it will create an extra directory for the simulated measurement files. The simulated measurements have the same format as the real measurements after execution (but note that the order of the measurements may be different). One can, therefore, program the analysis using the simulated data and troubleshoot it before execution (depicted in Figure 2.7). Sometimes this reveals mistakes in the design or protocol, thereby enabling corrections and avoiding wasted lab resources.

well	absorbance
A01	0.1532
A02	0.2670
A03	0.3768
B01	0.2062
B02	0.3128
B03	0.3947

⊛

well	reagent1	reagent2
A01	10 ul	10 ul
B01	10 ul	20 ul
A02	20 ul	10 ul
B02	20 ul	20 ul
A03	30 ul	10 ul
B03	30 ul	20 ul
A04	40 ul	10 ul
B04	40 ul	20 ul

⇒

well	absorbance	reagent1	reagent2
A01	0.1532	10 ul	10 ul
A02	0.2670	20 ul	10 ul
A03	0.3768	30 ul	10 ul
B01	0.2062	10 ul	20 ul
B02	0.3128	20 ul	20 ul
B03	0.3947	30 ul	20 ul

Figure 2.6: **Tidy measurements.** The left table holds the values acquired from measurement equipment, such as the sample absorbance readouts shown here; the rows are arranged in the order the measurements were taken. The middle table holds the design that was constructed as part of the design phase; the rows are arranged in the order specified during design, which can be different than the measurement order. The blue symbol between the first two tables represents a "left join" using a "join key" (the `well` field in this example). The right table holds the results of the join. It is a tidy table that has all the measurements and their corresponding factors. Note that although the design table has a different ordering than the measurement table, the result is in the same order as the leftmost table. The extra rows in the design table are ignored. By joining the measurements and design, Roboliq makes it possible to start analyzing measurements immediately without first performing tricky data munging.

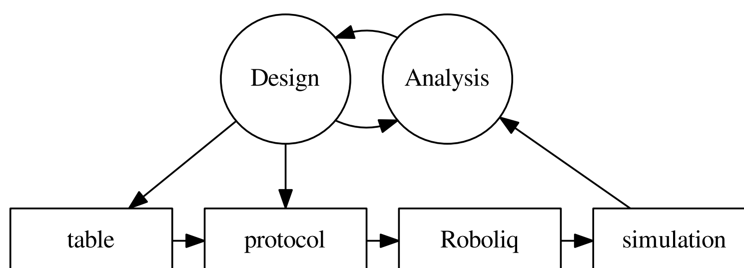


Figure 2.7: **The role of simulated measurements.** Roboliq can generate simulated measurements to help troubleshoot the design-execution-analysis cycle before execution even takes place. During the design phase, the user produces a design table, a Roboliq protocol, and the analysis script, and makes sure that they all work together. The design table is an input to the protocol, the protocol is compiled by Roboliq and produces simulated data, and the simulated data is the input to the analysis script. The analysis helps the user determine whether to continue to the execution phase; if problems arise, then he iterates through the design phase again to alter the design table, protocol, or analysis script.

2.2.7 Experimental Design Workflow

Several of the experiments in this thesis are too complex to apply the traditional methods from Design of Experiments, so it was necessary to apply a more involved process of experimental design, and I would like to highlight the elegance of the workflow and how Roboliq facilitates it. The steps were as follows:

1. Define the model/hypothesis
2. Design an experiment
3. Choose parameter values for simulation
4. Repeat N times:

- Simulate experiment
- Analyse simulated data to obtain parameter estimates

5. Analyse the means and variances of the parameter estimates, comparing them to the chosen parameter values
6. If the simulations show that the experiment can **not** recover the original parameters, try to determine why and refine the experiment or model, then try again
7. Perform the experiment
8. Analyze the experimental data (same analysis as performed on the simulated data)
9. If the analysis results are unsatisfactory, try to determine why and refine the experiment or the model, then start again

There are two key characteristics of this workflow. First, it tightly integrates the scientific cycle of hypothesis-experiment-analysis, iterating through until all three work together. Second, the vast majority of the work is completed in silico before the actual experiment is performed, significantly increasing the likelihood of a successful experiment. By completely preparing and testing the analysis before the experiment, we minimize problems of overly complex models and poor experimental designs whose power is inadequate to achieve the required confidence intervals.

Roboliq was designed to facilitate this cycle: 1) It accepts experimental designs and simulated data generated in R. The protocols written in Roboliq can conveniently use this exported data to control pipetting and measurement actions. 2) It can use simulated data from R to generate its own simulated measurement data files, which can then be checked to verify that the protocol does what we intended it to do. 3) The same format for simulated R data can be produced by Roboliq's simulations and measurements, allowing the same analysis functions to be used with any of those three sources.

2.2.8 Portability and Merging Data Structures

Mergable Data Structures

Since a portable protocol omits lab-specific information, Roboliq must be able to merge this information from separate files. A lot of work has gone into merging data structures; Two examples include version control systems [73, 74] and distributed databases [75]. Version control systems, such as git, allow multiple people to edit the same files independently, and they automatically merge the separate changes when possible. The merging algorithm finds the changes that each user made to each file, and tries to apply those changes to its own version of the file – only in the case of conflicting changes is manual intervention necessary. Distributed databases have the same need to merge independent changes into the same records.

There are many algorithms for merging [76–78], but perhaps the most widely used one is the "diff" algorithm [79] and its complementary "patch" utility. As input it takes both the original file before editing and the current version after editing; It then finds which lines have changed and produces a description of the differences. See Figure 2.8 for an example. Later, the patch utility can apply the diff on another computer. This lets people exchange just the changes they've made while editing text files, rather than sending the entire file. The advantages are that the recipient can 1) quickly identify and review the changes, and 2) apply the diff without overwriting their own edits to the same file. Analogous algorithms for JSON objects have also been developed, such as [80]. Figure 2.8D shows such a diff in JSON format.

A)

```
obj1:
  A: text1
  B: [1, 2, 3, 4]

obj2:
  a: 1
  b: 2
  c: 3

obj3:
  d: 4
  e: 5
```

B)

```
obj1:
  A: text2
  B: [2, 3, 4, 5]

obj2:
  a: 1
  b: 2
  c: 3

obj3:
  d: 4
  f: 6
```

C)

```
2,3c2,3
<   A: text1
<   B: [1, 2, 3, 4]
---
>   A: text2
>   B: [2, 3, 4, 5]
12c12
<   e: 5
---
>   f: 6
```

D)

```
[
  {"op":"replace", "path":"/obj1/A","value":"text2"},
  {"op":"remove", "path":"/obj1/B/0"},
  {"op":"add", "path":"/obj1/B/3", "value":5},
  {"op":"remove", "path":"/obj2/e"},
  {"op":"add", "path":"/obj2/f", "value":6}
]
```

Figure 2.8: **Examples of "diffs".** The top panels, A and B, show the "before" and "after" versions of two sample YAML documents. (C) The line-oriented output of the "diff" algorithm. The numbers indicate the lines of the original and new files; lines that start with `<` indicate that the line was removed in the new version, and `>` indicates that the line was added in the new version. (D) The JSON output of a diff algorithm based on JSON Patch. The diff is an array of objects; the `op` property can be `add`, `remove`, or `replace` and indicates which modification took place; the `path` property is a pointer to where the modification took place; the `value` property provides the new value.

A)

```yaml
obj1:
  A: text1
  B: [1, 2, 3, 4]

obj2:
  a: 1
  b: 2
  c: 3

obj3:
  d: 4
  e: 5
```

B)

```yaml
obj1:
  A: text2
  B: [2, 3, 4, 5]

obj3:
  e: null
  f: 6
```

C)

```yaml
obj1:
  A: text2
  B: [2, 3, 4, 5]

obj2:
  a: 1
  b: 2
  c: 3

obj3:
  d: 4
  f: 6
```

Figure 2.9: **Example of merging using key/value pairs.** Continuing from the previous figure, we see how we can merge two YAML objects: Merging panels A and B produces C. (A) The original data (copied from Panel A of the previous figure). (B) The new data. It has the same formatting and syntax as the original version. (C) The final data (copied from Panel B of the previous figure). `obj1.A` and `obj1.B` were replaced, `obj2.e` was removed, and `obj2.f` was added.

Roboliq's Data Structure

We decided not to use the diff/patch strategy to merge our data structure instances because diff files are verbose and difficult to write by hand. Instead, we picked a very simple but powerful merging algorithm that can be applied to nested key/value dictionaries: start with the original data and add all key/values pairs from the diff; The original keys are overwritten, but if the value is `null`, the key is removed. If a value is itself a key/value dictionary, then the merging algorithm recurses into it. This means that every data instance can be written according to the same rules and we avoid the necessity for a separate diff format (Figure 2.9).

For Roboliq's data structure, we use a JSON/JavaScript object a key/value dictionary. This allows us to apply the above merging algorithm, where the keys are the properties described in Sections 2.2.1, 2.2.11 and Appendix B.

Transforming Arrays to Dictionaries

The rules for merging key/value pairs are simple and unambiguous, even for very complex key/value structures. The tradeoff is that they cannot be applied to manipulating arrays — they treat arrays as basic values (like strings and numbers) and only allow for replacing the entire array, but not for adding or removing elements. So even though an array is sometimes the intuitive choice for a property, we may need to formulate it as a key/value dictionary instead. The primary example is the `steps` property: one would expect an array of steps, but instead, the steps are explicitly numbered. Depending on their usage, arrays should be transformed to dictionaries by one of the following methods.

Usage 1: The array can be treated as an atomic value whose elements do not need to be manipulated individually during the merging process. This applies to some of Roboliq's command parameters, such as the `sources` list of the `pipetter.pipette` command. In this case, the array does not need to be transformed to a dictionary.

Usage 2: The array's elements may have a unique identifier that can be used as a key. This applies to Roboliq's `objects` list, for example, where each object has a unique name. In such cases, the transformation from an array to a key/value dictionary is straightforward (Figure 2.10A).

Usage 3: The array's elements do not have unique identifiers, but they have a specific ordering and so their index can be used as a key. This applies to Roboliq's `steps` list, for example, where each step uses explicit numbering as a key (Figure 2.10B). There is a distinct drawback to this approach: re-arranging the order of elements requires explicitly renumbering them. In practice, I found this tolerable, because protocols tend not to have too many steps.

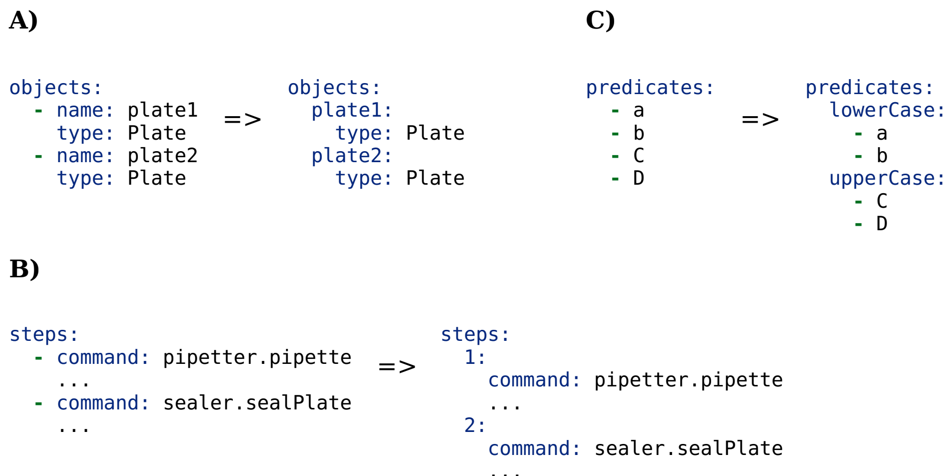


Figure 2.10: **Using dictionaries to represent arrays.** (A) The array on the left has values with unique identifiers; such arrays can be transformed to dictionaries by using the unique identifiers as keys for the values, as shown on the right. (B) The array on the left has a unique ordering; such arrays can be transformed to dictionaries by using the indexes as keys, as shown on the right. (C) The array on the left could be a concatenation of smaller arrays; such arrays can be transformed to dictionaries by using an arbitrary but unique key for each sub-array.

Usage 4: Merging should append arrays rather than modify them. In Roboliq, this only applies to the `predicates` list, which holds the logical information for the AI algorithms. In this case, each sublist needs to be given a unique key, so that merging results in a dictionary of sublists whose keys will be ignored (Figure 2.10C). For internal processing, Roboliq transforms the dictionary back into an array by appending the sublists in the order that they were merged.

Using the key/value dictionaries as described above, we can support all the value types required in Roboliq's protocol data structure. The lets us split the information necessary for running protocols among separate files in a modular manner. For example, the robot configuration can be in one file, a generic protocol in another file, and user-defined modifications to that protocol in a third file. By using the data structure described in this chapter, Roboliq can seamlessly merge them so that all necessary information is in one place.

2.2.9 Portability and Automated Planning

An important property of high-level commands is that the user can omit some details. Roboliq uses algorithms from artificial intelligence to fill in those details, and the AI's logic can be specified in the configuration file for each lab.

We had two criteria for choosing an AI algorithm. Most importantly, the results should make sense to the human mind, so that users can troubleshoot problems that might arise in the experiment. Secondly, it should be sufficiently fast so that users do not get frustrated waiting for a solution.

We experimented with several AI algorithms, including STRIPS, PSP, and Graphplan [81]. Those algorithms produced plans that were technically correct, but sometimes very counterintuitive, which led to a lot of confusion. PSP initially appeared to be an exceptionally elegant solution, but its calculation time was unacceptable for large protocols. In personal communications, the author of Nonlin [82] recommended HTN (Hierarchical Task Network) [40]. It is not the most powerful planning algorithm, in the sense that other algorithms can find solutions that HTN cannot, but it is tailored to fast simulations of intelligent agents, and it generates plans that are compatible with human intuition.

In HTNs, high-level tasks (such as the plate transport command) are decomposed into lower-level tasks, such as unit operations using specific methods and devices. Constraints on operations and resources are specified

in the methods, and detailed schedules of tasks result from optimizations on the task network. HTN allows for inference, evaluation of alternative methods, and first-order (predicate) logic for reasoning about tasks and actions. Automated planning in Roboliq relies on a JavaScript implementation of an established HTN library [66, 67]. We ended up choosing HTN for high-level planning because:

1) Setup is relatively intuitive because it is easy for us to break tasks down into smaller sub-tasks.
2) The resulting plans are usually easy to understand because they follow from the task/subtask hierarchy.
3) HTN is relatively fast.

Based on the robot configuration file, Roboliq uses HTN for the following purposes:

- to choose which equipment to use in commands
- to fill in omitted parameter values in commands when possible
- to find valid transportation paths for labware
- to automatically open and close equipment when necessary
- to verify compatibility of labware, sites, and equipment

We had also experimented with using the A* algorithm [83] to optimize several basic aspects of pipetting:

- tip choice
- liquid class choice
- parallel pipetting (using more than one tip at a time)
- source choice, when multiple sources of the same liquid are available

However, the results were sometimes too counterintuitive, so we reimplemented the optimization using simpler greedy methods. Regardless of which AI algorithms are used, we found the main drawback is that it can be very difficult to troubleshoot errors in the AI's configuration file.

2.2.10 Portability and its Limits

One of our primary aims for Roboliq was to develop portable protocols, but our initial goals turned out to be somewhat naive. We found that the differences in how labs need to run protocols are often so significant that the value of portability disappears. This is because portability pre-supposes the lack of lab-specific information, so the more universal a protocol should be, the vaguer it becomes and the more decisions it requires from the users before execution. We, therefore, revised our aims and focused on the more modest goal of portability between labs using similar equipment and methods.

It requires more care to write a portable protocol than a protocol that only runs in one lab. In practice, I usually first wrote and tested protocols for a specific robot, and only if I wanted to run the same protocol on another robot later, then I would make it portable. In order to achieve portability, it is necessary to split the protocol into two parts: the portable and the lab-specific components. The lab-specific components should be moved to a separate file, and to run the protocol in the same lab as before, one passes both files to Roboliq for merging. To run the protocol in a different lab, the portable file can be shared and appropriately adapted by their users.

2.2.11 Advanced Protocol Usage

While simple protocols were described in Section 2.2.1, advanced protocols can make use of additional elements. The `imports` property accepts a list of external modules in order to allow for reuse. The `parameters` property accepts a map from parameter names to values for parameterizing a protocol. There are additional object types available for more specialized use-cases, and variable substitution allows one to construct strings, perform calculations, and build new objects.

In Section 2.2.6, we described data tables and their role in the design-execution-analyze cycle. Roboliq has several constructs to support them. The `Data` object type is used for defining or loading a data table. Within steps, the `data` property activates a table or rows for use in the current step and sub-steps. The `data()`

function can transform data tables into other shapes or formats when necessary. And finally, the `data.*` commands allow for iterating over rows or groups of rows during execution.

The details of advanced protocols are described in Appendix C.

2.2.12 Software Implementation

Here we briefly describe several of the more salient implementation topics and where to find further information.

Commands. Roboliq comes with a set of built-in commands (Tables D.1 and D.2), some of them high-level, and some low-level. The low-level commands are not intended for regular use, but they are available for the sake of flexibility. Low-level commands normally require values for all of their parameters, whereas the high-level commands can fill in many missing parameters via automated planning.

Software dependencies. Roboliq takes advantage of various external JavaScript libraries for handling file format, math expressions, command validation, and more. The details are in Section D.4.

Schemas. In Roboliq protocols, the objects and steps are declared as JSON objects, and their structures are defined according to JSON Schema [84] with a couple of extensions to accommodate the requirements of Roboliq. JSON Schemas define which properties a JSON object must or can have, and what kinds of values those properties should have. Further details can be found in Section D.5.

Input formats. Roboliq accepts protocols in four input formats: JSON, YAML, and JavaScript. JavaScript files should either export a protocol object or a function to construct the protocol object (see Section D.6).

Data structure for well contents. Roboliq tracks the contents of wells with each pipetting operation. Naive approaches to this are inefficient and lead to rounding errors, so we designed a more efficient data structure in the form of a recursively nested array. The details of its construction are described in Section D.6.1.

The `agent` property in commands. Most low-level commands require an `agent` property to indicate which robot should execute the command. Multi-agent protocols are possible, and backend compilers select which commands to compile based on the agent value (Section D.6.2)

Execution. Execution involves various input and output files, as well as callbacks for Roboliq to process measurements. The details are described in Section D.6.3

2.3 Conclusions

Roboliq aims to make it easier to automate liquid-handling robots. It provides a format for writing formal protocols, simplifies some aspects of robot programming, and compiles the protocols for execution by liquid handling robots. Simple protocols can be written using four properties: `roboliq`, `description`, `objects`, and `steps`. The `roboliq` property indicates the version of Roboliq, and `description` document what the protocol does. The `objects` specify the labware and liquids used in the protocol. And `steps` is a list of numbered commands; the steps can be nested and documented as the user sees fit. Online resources include the source code [85], the user manual [86], the object and command documentation [87], and the API documentation [88].

Roboliq's approach to portability involves four components: 1) a data structure that can merge its information for various files, 2) artificial intelligence algorithms that can fill in the missing details for high-level commands, 3) a software architecture that gathers the protocol data and processes the commands recursively until only low-level commands remain, and 4) a back-end compiler to compile the low-level commands to the robot language.

Finally, Roboliq supports an integrated scientific workflow. One starts by designing the experiment in a principled manner to create a "design table" that describes all the factors and treatments of the experiment. The table is saved in JSON format and imported by the Roboliq protocol. Next, the steps of the protocol are

written using the `data` property and functions, allowing the table values to be substituted into the steps in a flexible manner. By assigning the `simulated` property in measurement steps, Roboliq can output simulated data that is in the same tidy form as real measurement data. Next, one loads these simulations into the analysis software (e.g., R) and develops the analysis routines. This provides the opportunity to refine the design, execution, and analysis before ever turning on the robot. In practice, this workflow was very efficient for us.

Roboliq aims to make it easier to use liquid handling robots for automation in biological laboratories. It integrates well into the design-execute-analyze cycle of experimentation, provides a format for writing formal protocols, helps make protocols portable between labs, simplifies some aspects of robot programming, and compiles the protocols for execution by liquid handling robots.

Chapter 3

Proof of Principle Experiments

3.1 Introduction

This chapter aims to demonstrate Roboliq's usefulness where automation has the clearest benefits. **Complex pipetting.** Complex pipetting arises when testing multiple mixture factors, such as in protocols for screening, quality control, and optimization of growth media. The number of unique well mixtures grows exponentially with the number of factors and their distinct testing levels, and it quickly becomes impractical or impossible for humans to execute [89, 90]. **Time-critical and long-duration experiments.** A robot's precise timing of measurements and other operations can reduce variability and significantly improve the statistical power of the final data analysis. The timing of a lab technicians, on the other hand, can be quite inconsistent [91, 92]. **Metadata.** Enormous amounts of data can be collected in automated experiments; while organizing the data and its associated metadata is not a reasonable task for humans, software may be able to handle it automatically [19]. **Repeatability.** Robotic procedures can be repeated with much greater precision than humans can repeat their movements and timing. Automation can, therefore, help minimize the negative influence of "human factors" on repeated experiments [14].

Most use cases for liquid-handling robots involve at least one of the above motivations, so here we investigate a Roboliq application that exemplifies all of them: the quantitative biophysical characterization of fluorescent proteins. The application consists of three distinct experiments, each of which is difficult to perform manually due to involving complex mixtures, time-critical procedures, and long duration. And while typical automation solutions in synthetic biology focus on highly standardized protocols to build synthetic constructs, testing tasks like these are harder to automate because the assays are more specialized and involve devices beyond the core liquid handling capabilities of a robotic workstation (e.g., measurement devices).

3.2 Biological Background

The application's task is to characterize a set of green fluorescent proteins (GFPs). This section provides an overview of the biology involved – since the biology is not the focus of this thesis, however, the overview will stick to the basics and omit most caveats, exceptions, and subtleties.

3.2.1 Fluorescence

Fluorescence is a mechanism whereby a substance emits light in response to light being shined on it [93, 94], as illustrated in the Figure 3.1. The energy of the emitted light is less than the absorbed light, so it has a longer wavelength than the light that causes the original excitation. The fluorescence can only continue in the presence of excitation.

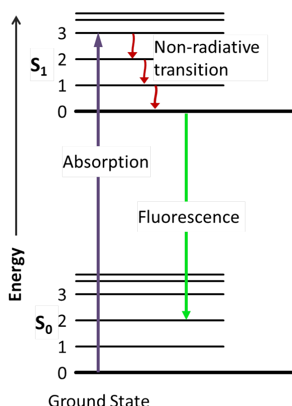


Figure 3.1: **Jablonski diagram of absorbance, non-radiative decay, and fluorescence.** This diagram represents the energy changes of an electron that lead to fluorescence. The electron starts in the ground state; it rises to an excited energy state after absorbing light; molecular vibrations result in small energy losses until a larger drop returns the electron towards its ground state. The large drop results in the emission of a photon of lesser energy than the initial excitation. (Image under public domain license CC0 v1.0.)

The fluorescence readers used in this thesis shine a beam with a predetermined wavelength into a well containing fluorescence proteins, and they then measure the emitted light at another predetermined lower wavelength. Normally the wavelengths are the chosen to maximize the emission intensity.

3.2.2 Green Fluorescent Protein (GFP)

The GFP structure (Figure 3.2) is a kind of barrel with a fluorescent core in the middle and a tail at the end that can be attached to other proteins through genetic engineering [95]. This construct is used extensively in genetic engineering, helping researchers detect the presence of a protein of interest by fluorescent measurements [97]. A lot of engineering has gone into GFP variants to alter its properties, such as producing different colors [98].

In these experiments, we look at five green fluorescent proteins: sfGFP, N149Y, Q204H, Q204H+N149Y, and tdGFP. These data were acquired as part of an effort to obtain a pH stable GFP variant (i.e., fluorescence intensity should hold steady over a larger range of pH levels) [99]. The starting protein was sfGFP (superfolder GFP) [100] from which a pH-stable variant bearing four mutations was obtained. A mutational analysis showed that only the combined effect of the N149Y and Q204H mutation contributed to the observed increase in pH stability. However, those mutations turned out to be dimerizing, but tagging proteins need to be monomeric [101]. To artificially monomerize the double mutant (Q204H+N149Y), it was encoded as a single chain that folds back onto itself [102]. This produced the tdGFP variant, a "tandem dimer GFP". Here, we use the part of the raw data alluding to the initially mentioned five fluorescence protein variants. The full description of how tdGFP was obtained, characterized, and optimized to become a useful tool is described elsewhere [99].

3.2.3 pH and Acidity

The pH scale indicates how *acidic* or *basic* a solution is. More precisely, it is a measure of the concentration of protons (H+). In "pH", the "p" stands for potency, and the "H" stands for the hydrogen ion [103]. The scale is typically described as extending from 0 to 14: neutral solutions have pH 7, acidic solutions have pH below 7, and basic solutions have pH above 7. Cells typically have a pH slightly above 7 [104].

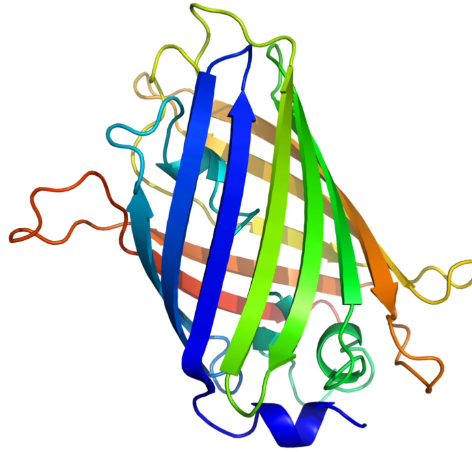


Figure 3.2: **Ribbon Diagram of Green Fluorescent Protein (GFP)** Produced from PDB 1EMA, rendered in PyMOL. © Richard Wheeler, CC BY-SA 3.0.

3.2.4 Folding and Unfolding of Proteins

For a protein to become operative, it first needs to *fold* into its functional 3D shape [105]. Conversely, when a protein *unfolds*, it loses its function. The protein's surrounding conditions influence folding in a complex fashion, and a change of conditions (e.g., pH, temperature) can lead a protein to unfold partially or completely.

To force unfolding, we will use a *denaturing* reagent (e.g., acids, urea, and ethanol) [106]. With regard to fluorescent proteins, denaturants can cause the barrel to open or the core to shift, thereby deteriorating the fluorescent activity.

3.3 Results and Discussion

3.3.1 Proof-of-principle applications

To provide new proof-of-principle applications that exemplify Roboliq's broad applicability, we automated a series of three experiments for the quantitative, biophysical characterization of superfolder green fluorescent protein (sfGFP) [100]; part of the raw data reported here was previously published in a screening application for fluorescent proteins [99]. While typical automation solutions in synthetic biology focus on highly standardized protocols to build synthetic constructs, the testing tasks we describe here are more difficult to automate because the assays are more specific and because the tasks involve devices beyond the core liquid handling capabilities of a robotic workstation (for example, measurement devices). In addition, we designed the experimental series such that the assays emphasize features where automation has clear benefits over manual experimentation: complex pipetting series, time-critical procedures, and long-duration measurements with regular interventions.

The first type of experiment aimed to characterize the pH stability of sfGFP. The protocol (Figure 3.3A,B; see Methods for details on experiments and data analysis) involves a complex pipetting application in which multiple factors are systematically manipulated to obtain pH levels ranging from 3.75 to 8.5 while keeping the ionic strength of the mixture constant. This required the use of multiple buffer systems (Acetate, MES, HEPES, PIPES) and adjustments of the concentrations of the buffers' acid and base components in each well of a microtiter plate to achieve the same total volume per well. For Roboliq to create the mixtures, it needs to be given an appropriate specification, such as the one shown in Figure 3.3B. Note that the format allows

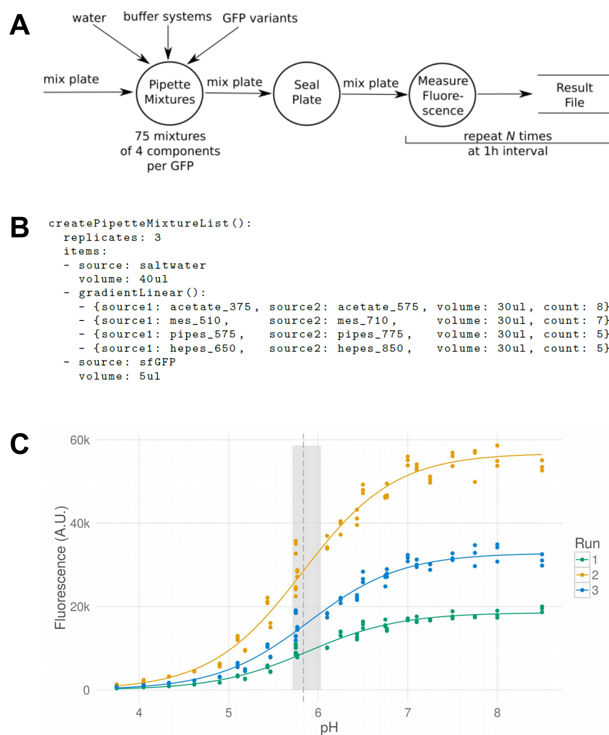


Figure 3.3: **pH stability of sfGFP at constant ionic strength.** (**A**) Data flow diagram of the protocol. For each GFP variant, 25 pH levels are pipetted in triplicate on a microwell plate. The plate is then sealed, and fluorescence is measured. Optionally, measurements can be repeated to verify that the fluorescence readout is stable. (**B**) Roboliq specification of the well mixtures: there will be three replicates per condition, and each well will contain 40 µL of salt water, 30 µL of one of four buffer systems, and 5 µL of sfGFP solution. The `gradientLinear()` field creates linear gradients for each of the four buffer systems: for example, acetate will be tested at 8 different concentrations ranging from pure `acetate_375` (acetate at pH 3.75) to pure `acetate_575`. (**C**) Measurement of pH stability. Ionic strength is held constant over the entire continuum by utilizing four buffer systems, for three experimental runs, each with three replicates per pH value. The sigmoid fits are drawn as solid lines, the overall pKa estimate of 5.68 is indicated by a dashed vertical line, and its 95% confidence interval of 5.74 to 5.97 is shaded gray.

for a concise definition of the mixture list, sparing the user the tedious task of calculating the mixtures for each well. In independent experiments performed as much as six months apart, the fluorescence readouts (Figure 3.3C) yielded very consistent estimates of pKa values for the three individual runs. We obtained an overall pKa estimate of 5.86 with a 95% confidence interval (CI) of 5.74–5.97 and a 95% prediction interval of 5.66–6.06. To our knowledge, these are the first estimates of pKa for sfGFP.

Next, we ran a time-critical assay to determine the refolding kinetics of sfGFP (Figure 3.4A,B). We started the assay by adding a denaturant to the protein sample, waited for unfolding to complete, diluted out the denaturant by adding a large volume of salt water, and finally, measured the appearance of fluorescence. The critical challenge is that we need to start measuring fluorescence within a fraction of a second after refolding begins. To automate all measurements, we manually set up the fluorescence reader for a single set of wells and saved the reader program in the file referenced by `programTemplate!` in step 4 of Figure 3.4B. Roboliq uses this as a reader template, inserting the correct set of wells for each readout. The resulting measurements demonstrate striking reproducibility: the three replicates are indistinguishable once normalized (Figure 3.4C), leaving reader error as the dominant noise source. From the initial slope of the kinetics (Figure 3.4D), we estimate the mean initial folding rate as $14.1\,\%\,s^{-1}$ (standard error of $0.57\,\%\,s^{-1}$), with a 95% prediction interval of $13.0\,\%\,s^{-1}$ to $15.3\,\%\,s^{-1}$, which is very consistent with the previously reported value of $13.5\,\%\,s^{-1}$ [100]. In Figure 3.4E, we see the fits for the refolding curves of all five GFP variants. Four of the variants appear to fold at similar rates — only tdGFP diverges significantly. Biologically, we suggest that tdGFP's slower folding is related to the tandem dimerization discussed in Section 3.2.2.

Finally, we measured the evolution of sfGFP during protein unfolding, which belongs to another class of tedious assays, namely long running experiments with periodic human intervention. The assay (Figure 3.5A) subjects sfGFP to a range of denaturant concentrations and then regularly measures its fluorescence for two days; this provides time-resolved denaturation curves and allows us to estimate the protein folding energy ΔG. Roboliq simplifies several aspects of this experiment, similar to the pH experiment. Additionally, the repeated measurements and their timings are simple to specify (see Figure 3.5B). During execution, the software correctly instructed the robot through 48 cycles of measurement without errors, demonstrating reliability for long-term procedures that resulted in the kinetics shown in Figure 3.5C. A final measurement after five days allowed enough time for the mixture to reach equilibrium [107]. From the data at equilibrium, we estimated a denaturation midpoint of 3.87 M and a change of Gibbs free energy ΔG of $8.02\,\text{kcal}\,\text{mol}^{-1}$ (Figure 3.5D). Previously, values for ΔG of $9.5\,\text{kcal}\,\text{mol}^{-1}$ [100] and for the saturation midpoint of $\approx$4 M [108] were reported.

Strong dimerization interactions require higher concentrations of denaturant to unfold, so the unfolding curves provide some indication of relative dimerization strength when considered in combination with chromatography and PAGE tests described elsewhere [99]. The data shown in Figure 3.5E are consistent with our expectations: 1) The monomeric protein (sfGFP) requires the lowest denaturant concentrations, 2) The tandem dimerizing protein (tdGFP) is shifted most to the right, and 3) The other dimerizing variants are nested between sfGFP and tdGFP. }

Overall, we conclude that the proof-of-principle application demonstrates the high replicability and low variance that laboratory automation can achieve in quantitative assays for component testing. All three experiments also show that Roboliq helps to efficiently automate a useful subset of experimental procedures with good reproducibility.

3.3.2 Complex Pipetting

Complex pipetting tasks arise when working with many wells that each need different mixtures. Such complexity frequently arises in protocols for characterization, screening, optimization, and designed experiments with multiple mixture factors.

The first experiment in this chapter demonstrates a complex pipetting series with 375 well mixtures. The mixture table has 375 rows, one for each well; it was generated from a 38-line specification using Roboliq's design language (show in Figure F.1), but the method of generation is not particularly important. Table 3.1 shows an excerpt of the mixture table for sfGFP with the acetate buffer systems. The first three columns

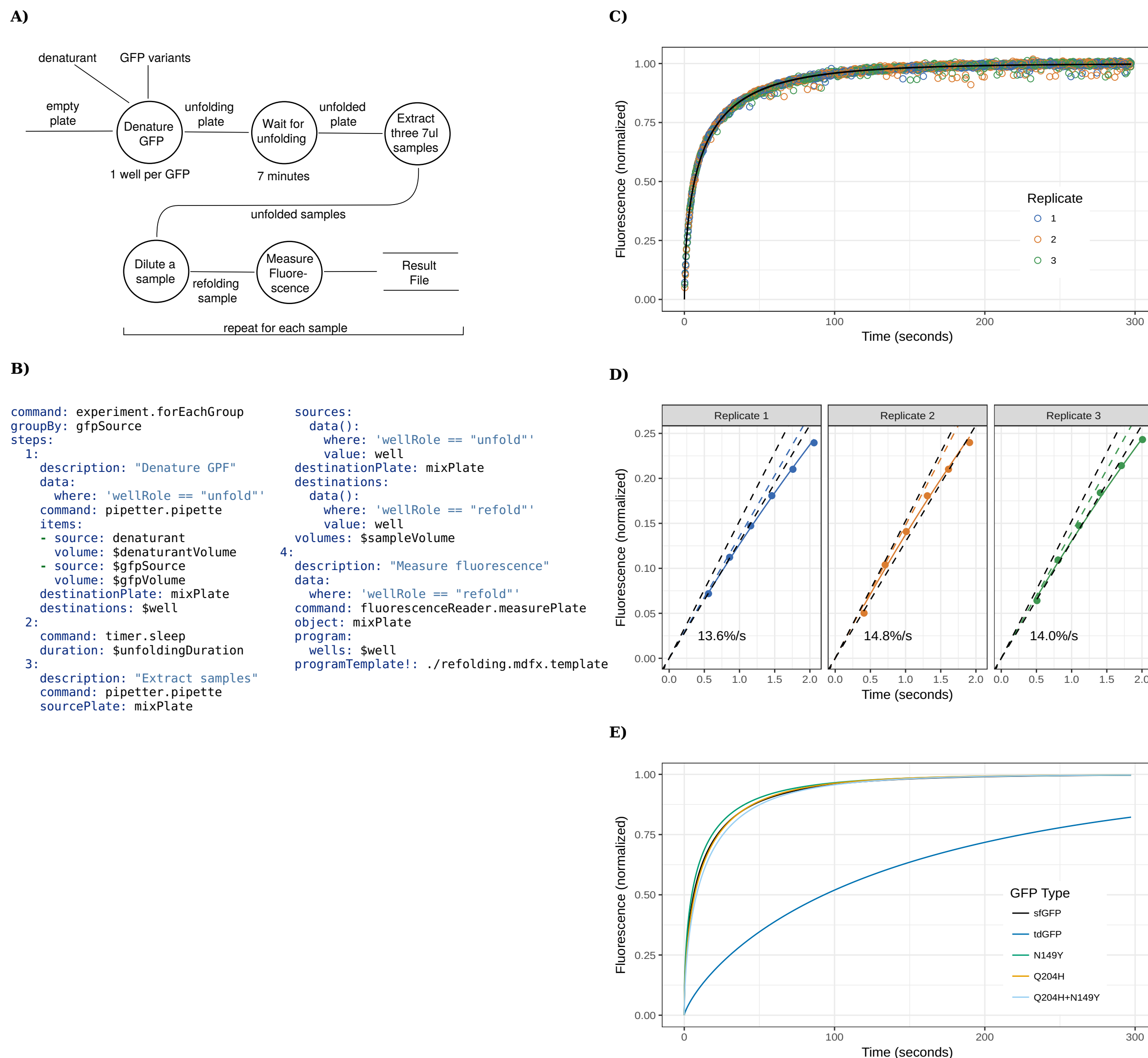


Figure 3.4: **Folding rate.** (**A**) Data flow diagram of the protocol. For each GFP variant, the protein is first unfolded by mixing it with the denaturant and waiting for 7 minutes; then several small samples are extracted; sequentially, each sample is diluted to initiate refolding, and its fluorescence is measured continuously for 5 min. (**B**) Roboliq specification of the experimental steps to (1) create the denaturing mixture, (2) wait, (3) extract samples, and (4) measure fluorescence. The **$**-prefix indicates variable substitution, which allows the user to adapt the protocol more easily. In step 4, Roboliq generates the measurement program from a template file, automatically inserting the correct wells. (**C,D**) Fluorescence during refolding of sfGFP. The three replicates are distinguished by color. Measurements were recorded every 0.3 s. The solid black line represents a stretched exponential fit. (**D**) The initial folding rates of the three replicates. The dashed black line shows the 95% prediction interval of 13.0 to 15.3 $\%\,s^{-1}$; the dashed color line shows the estimated slope for the given replicate; the solid colored line shows the exponential fit over the first 2 s; the dots are the measured values; the estimated rates are indicated in percent recovery per second. (**E**) Fluorescence during refolding of all five GFP variants. Each line shows the stretched exponential fit for one variant.

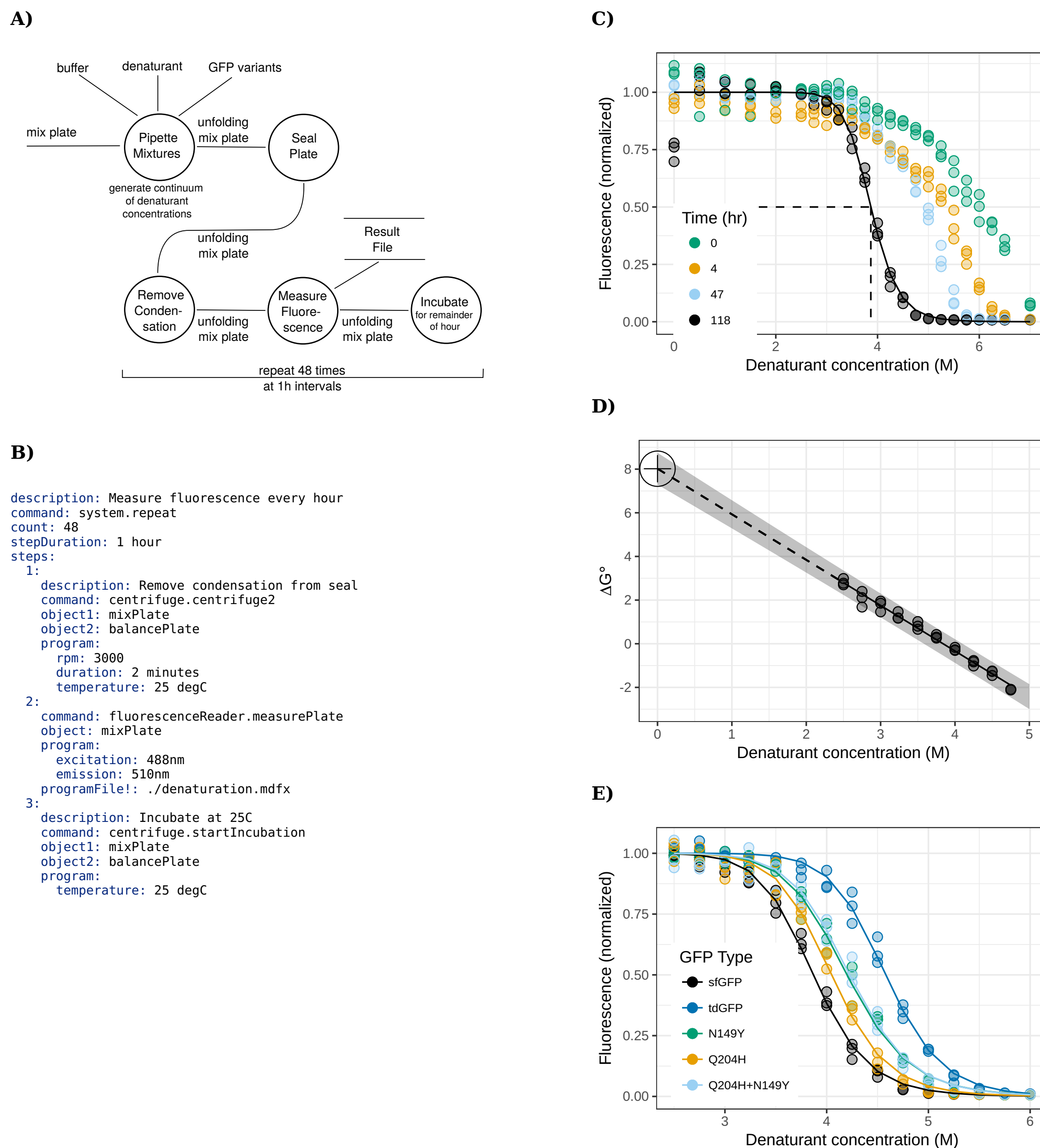


Figure 3.5: **Unfolding of fluorescent proteins.** Fluorescence measured over a range of denaturant concentrations, allowing for the quantification of sfGFP stability to denaturant — in other words, the Gibb's free energy ΔG. (**A**) Data flow diagram of the protocol. Each GFP variant is mixed with 23 different denaturant concentrations in 3 replicates per concentration; the plate is sealed to prevent evaporation; the measurement cycle is repeated 48 times, whereby the plate is briefly centrifuged to remove condensation, fluorescence is measured, and the plate is incubated for the remainder of the hour. (**B**) Excerpt from the unfolding protocol of the command for measuring fluorescence over time. The `system.repeat` command ensures that its sub-steps execute 48 times and that each cycle lasts 1 hour. (**C,D**) Fluorescence measurements and estimation of ΔG for sfGFP. (**C**) A representative subset of the data colored by time. The solid line shows the sigmoid fit for the last measurement. The dashed line indicates the denaturant midpoint of 3.87 M. (**D**) Estimation of ΔG by extrapolating the linear regression of $\Delta G°$. The linear fit is indicated by the line, with $\Delta G = 8.02\,\mathrm{kcal\,mol^{-1}}$ and the shading indicates the 95% confidence interval. (**E**) Denaturant curves of all five GFP variants after five days, colored by variant. Points are measurements, and lines are fits. The graph shows differences in denaturant concentration required to unfold the proteins.

A) Column-oriented mixture table

source	destination	volume	layer
reagent1	A1	30 ul	1
reagent2	A1	10 ul	2
water	A1	60 ul	3
reagent3	B1	20 ul	1
reagent4	B1	30 ul	2
water	B1	50 ul	3

```
command: pipetter.pipette
items: mixtureTable1
```

B) Different column names

well	liquid	volume	layer
A1	reagent1	30 ul	1
A1	reagent2	10 ul	2
A1	water	60 ul	3
B1	reagent3	20 ul	1
B1	reagent4	30 ul	2
B1	water	50 ul	3

```
command: pipetter.pipette
items:
  data():
    source: mixtureTable2
    map:
      source: $liquid
      destination: $well
      volume: $volume
      layer: $layer
```

C) Row-oriented mixture table

well	liquid1	volume1	liquid2	volume2	liquid3	volume3
A1	reagent1	30 ul	reagent2	10 ul	water	60 ul
B1	reagent3	20 ul	reagent4	30 ul	water	50 ul

```
command: pipetter.pipette
items:
  data():
    source: mixtureTable3
    map:
      - {source: $liquid1, destination: $well, volume: $volume1, layer: 1}
      - {source: $liquid2, destination: $well, volume: $volume2, layer: 2}
      - {source: $liquid3, destination: $well, volume: $volume3, layer: 3}
    flatten: true
```

Figure 3.6: **Example of equivalent mixture data tables.** The tables show three different formulations of the same mixture specifications, and the code below each table shows how the formulation can be used with the `pipetter.pipette` command. Note that the optional `layer` column can be used to alter the pipetting order: adjacent items in layer 1 will be transferred first, then in layer 2, and so on. A) Here the mixture table is expressed in the same format as the `pipetter.pipette` command expects for its `items` property. Each row indicates a single liquid transfer from `source` to `destination` at the given `volume`. The `pipetter.pipette` command can assign the table (named `mixtureTable1`) directly to the `items` property. B) Here the mixture table uses different column names. To assign it to `items`, the rows must first be mapped to rename the columns. C) Here the mixture table has one row per well with columns to indicate the liquids and volumes of three mixture components. Each row of the table (named `mixtureTable3`) must be mapped to three rows, one for each of the three liquids, and the `flatten: true` property flattens the results to a single array of 6 rows.

(`gfp`, `buffer` and `pH`) are the treatment factors that will be used for analysis. They tell us which of the five GFP variants to mix in, which of the four buffer systems to use, and which pH level the well will have. `rep` indicate the replicate number (1 to 3); `acid` is the name of the acid source for the chosen buffer, `base` is the name of the base source, and `acid vol` and `base vol` tell how much of each source to dispense; `order` randomizes the pipetting order; `well` is the well assigned to the mixture.

Table 3.1: Excerpt of mixture table from pH experiment.

gfp	buffer	pH	rep	acid	base	acid vol	base vol	order	well
sfGFP	acetate	3.75	1	acetate_375	acetate_575	30 ul	0 ul	345	I22
sfGFP	acetate	3.75	2	acetate_375	acetate_575	30 ul	0 ul	34	B03
sfGFP	acetate	3.75	3	acetate_375	acetate_575	30 ul	0 ul	213	E14
sfGFP	acetate	4.04	1	acetate_375	acetate_575	25.7 ul	4.3 ul	63	O04
sfGFP	acetate	4.04	2	acetate_375	acetate_575	25.7 ul	4.3 ul	191	O12
sfGFP	acetate	4.04	3	acetate_375	acetate_575	25.7 ul	4.3 ul	300	L19
sfGFP	acetate	4.32	1	acetate_375	acetate_575	21.4 ul	8.6 ul	251	K16
sfGFP	acetate	4.32	2	acetate_375	acetate_575	21.4 ul	8.6 ul	64	P04
sfGFP	acetate	4.32	3	acetate_375	acetate_575	21.4 ul	8.6 ul	107	K07
sfGFP	acetate	4.61	1	acetate_375	acetate_575	17.1 ul	12.9 ul	78	N05
sfGFP	acetate	4.61	2	acetate_375	acetate_575	17.1 ul	12.9 ul	206	N13
sfGFP	acetate	4.61	3	acetate_375	acetate_575	17.1 ul	12.9 ul	154	J10
sfGFP	acetate	4.89	1	acetate_375	acetate_575	12.9 ul	17.1 ul	142	N09
sfGFP	acetate	4.89	2	acetate_375	acetate_575	12.9 ul	17.1 ul	298	J19
sfGFP	acetate	4.89	3	acetate_375	acetate_575	12.9 ul	17.1 ul	313	I20
sfGFP	acetate	5.18	1	acetate_375	acetate_575	8.6 ul	21.4 ul	37	E03
sfGFP	acetate	5.18	2	acetate_375	acetate_575	8.6 ul	21.4 ul	106	J07
sfGFP	acetate	5.18	3	acetate_375	acetate_575	8.6 ul	21.4 ul	259	C17
sfGFP	acetate	5.46	1	acetate_375	acetate_575	4.3 ul	25.7 ul	52	D04
sfGFP	acetate	5.46	2	acetate_375	acetate_575	4.3 ul	25.7 ul	68	D05
sfGFP	acetate	5.46	3	acetate_375	acetate_575	4.3 ul	25.7 ul	235	K15
sfGFP	acetate	5.75	1	acetate_375	acetate_575	0 ul	30 ul	160	P10
sfGFP	acetate	5.75	2	acetate_375	acetate_575	0 ul	30 ul	247	G16
sfGFP	acetate	5.75	3	acetate_375	acetate_575	0 ul	30 ul	113	A08

The execution time was approximately 1 h for the robot to prepare the 375 mixtures with 1500 liquid transfers. This is much faster than our lab technician could have done it, but more importantly, the robot avoids the human mistakes that inevitably arise when managing so many dispenses with varying sources and volumes. The fact that Roboliq easily handles randomization is another valuable feature that helps reduce confounding and makes analysis more robust. Our conclusion from this (and many other experiments) is that Roboliq and our Tecan robots support complex pipetting well in practice.

Complex mixtures can be specified with Roboliq's high-level `pipetter` commands. There are various approaches, but here we will describe an approach using the `pipetter.pipette` command, data tables (Section 2.2.6), and variable substitution (Section C.6).

First, we need to specify a data table describing the mixtures. The table can either be written by hand, written in Roboliq's concise design language (see Appendix E), or produced by other software. Three different examples are shown in Figure 3.6; the examples each describe the same two mixtures with three components, but in different formulations. The user is free to decide on the best formulation of their data tables because table mapping and substitution can translate it to the format required for the pipetting commands.

`pipetter.pipette` accepts an optional `items` table that specifies one transfer per row; the most important columns are `source`, `destination`, and `volume`. We pass our data table to the `items` property by mapping it to the expected format. In the simplest case, the data table has the same format as the `items` property

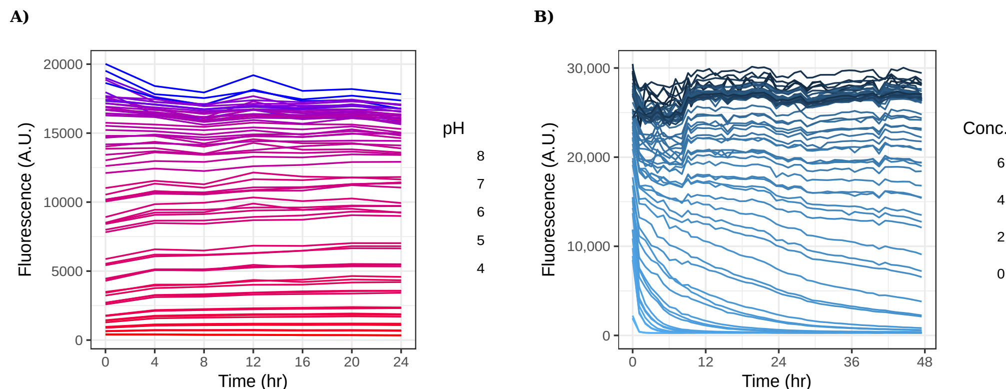


Figure 3.7: **Time series graphs for the pH and unfolding experiments.** These two graphs show time series data for wells with the sfGFP protein. Each line represents the measurements in a single well. **A)** Time series for pH experiment. Colors represent pH level. **B)** Time-course for unfolding experiment. Colors represent the molar denaturant concentration.

and can be assigned directly (Figure 3.6A). If the columns in the table have different names, the `data()` function can rename them via the `map` property (Figure 3.6B). If the data table describes the mixtures using one row per well instead of one row per liquid transfer, then the mapping is only slightly more complicated (Figure 3.6C). Should the need arise, Roboliq can accommodate many other formulations as well.

3.3.3 Time Series and Long-Duration Experiments

Many kinds of experiments involve measuring wells repeatedly over time. Long-running measurements present a challenge for people because we cannot go without sleep for too long. Such experiments are therefore one of the typical applications of lab automation. All three of the experiments in this chapter recorded time series, two of which extended over multiple days (see Figures 3.4C, 3.7).

Roboliq uses loops to run time series. The main loop commands are `data.forEachRow`, `data.forEachGroup`, and `system.repeat`. The `data` commands can execute a series of sub-steps that are adapted to selected rows in a data table, whereas `system.repeat` simply repeats a series of sub-steps a certain number of times. The loop commands accept an optional `iterationDuration` property to specify the timing between iterations.

We conclude from these experiments (and many others) that Roboliq provides a practical means to execute long-running experiments on our Tecan robots and collect the measurements for convenient analysis.

3.3.4 Time-Sensitive Experiments

We consider here two types of time-sensitivity: speed and accuracy. The experiments in this chapter demonstrate both types. Speed is required to detect the folding rate of our GFPs – the proteins are first held in an unfolded state until their denaturant is diluted out, at which point we need to measure the increase in fluorescence as quickly as possible. With the equipment on our robot, we were able to start the measurements within a fraction of a second after dilution, which allows for good estimates of the folding rate. It would take much longer to start the measurements manually, making any estimates of the folding rate less accurate.

In our first run of the pH experiment, we measured the wells every 4 hours for a total of 21 measurements over a period of 80 hours. To evaluate the timing accuracy, we consider the measurement times of well A1. Relative to the first measurement, the measurement times in hours were: 3.999, 7.999, 12.000, 16.000, 20.001, 24.002, 27.999, 32.002, 36.000, 40.001, 44.002, 48.002, 52.002, 56.002, 60.000, 64.000, 68.002, 72.002, 76.001, 80.004. On overage, the duration between measurements was 4 hours and 0.51 seconds, with a standard

deviation of 3.77 seconds. The timing results of the other experiments were similar. We conclude from these experiments that Roboliq can control timing accurately on our Tecan robots.

3.3.5 Metadata and Analysis

Without metadata, measurements are just a random collection of numbers. For almost any analysis, we need the metadata to tell us which conditions were applied to each well. Correctly matching the measurements and metadata can be very challenging and tedious, especially when dealing with large datasets, a large number of factors, or randomization. Roboliq's solution was described in detail in Section 2.2.6, and no manual matching of measurements and metadata was required for the fluorescence and absorbance charts in this thesis. For the automated measurements in this chapter, we conclude that Roboliq can automatically match measurements with their metadata and thereby facilitate analysis of both small and large datasets (Section 3.4.2 has a note about limitations).

3.3.6 Repeatability

Being able to faithfully repeat an experiment is perhaps the foremost requirement of experimental science. Robots can improve the repeatability of experiments because they exhibit less variability than people do. When re-running a robot script, a robot will pipette more accurately, transport labware with the same movements, and exhibit very consistent timing.

Low operational variability is a quality of the robot rather than of Roboliq. Nonetheless, we consider it here to evaluate whether our robots can deliver on the promise of repeatability. Figures 3.3C and 3.4 demonstrate excellent repeatability. The curves in Figure 3.3C are in fact from three runs that took place months and weeks apart from each other, using different stock solutions and in somewhat different environmental conditions due to the different seasons; nonetheless, the pKa estimates were very close. In Figure 3.4, the repeats were generated within the same experiment, and the results are virtually indistinguishible.

The repeatability of an experiment depends on many factors, so we can only conclude here that our Tecan robots demonstrate good repeatability for these experiments. Indeed, we've encountered poor repeatability many times due to poor experimental design, excessive biological variability, or attemping to use the robot for tasks that it has not been optimized for.

Roboliq can help in two ways, however. First, Roboliq integrates well into a good experimental design workflow, as described in Section 2.2.7. This integration makes a principled design workflow much easier to follow, and it improves the likelihood of obtaining statistically and scientifically significant results. Secondly, in Chapter 4 we will discuss quality control to ascertain the robot's pipetting performance. Qualitatively, we can use this information to avoid procedures that the robot cannot perform well. Quantitatively, we can use the pipetting parameters in our design workflow and to help determine required sample counts.

3.4 Methods

3.4.1 Experimental methods

Cloning, protein expression, and purification are described in detail in [99]. In short, the sfGFP DNA sequence was cloned with a PCR added 6x His tag as EcoRI/HindIII fragment behind the tac promoter in vector pKQV5 (Ptac, lacIq, ampicillin resistance gene, pBR322 origin of replication). The protein was expressed in standard LB media using *E. coli* JM101 (glnV44 thi-1 Δ(lac-proAB) F'[lacIqZΔM15 traD36 proAB+]) and expression was induced by the addition of 0.2 mM isopropyl-β-D-thiogalactopyranosid (IPTG). Expression was done in 50 mL LB containing 100 µg/mL kanamycin and 200 µM IPTG. Cells were harvested, lysed by sonication, and purified using an ethanol extraction method [109]. The protein was stored as a $1\,\mathrm{g\,L^{-1}}$ PBS

solution. Purity was checked by SDS-PAGE and was determined to be greater than 95% (see supplement of [99]).

The robot used was a Tecan EVO200 equipped with eight liquid handling channels (4 x 1 mL, 4 x 50 µL syringes) and two different robotic gripper arms. The systems liquid was deionized H_2O. We used fixed reusable tips and thoroughly washed them using two active wash Tecan stations. The relevant integrated peripherals were: a M200 Tecan fluorescence reader, a Hettich Rotanta 46 RSC centrifuge with integrated thermal control, a Roboseal sealer equipped with adhesive seal membrane suitable for RT-PCR from 3M, and a series of custom made holders for 1.5, 15 and 50 mL tubes. The system was controlled by the manufacturer's Tecan Freedom Evoware software.

pH stability. For the pH experiment, the robot had two sources each of four buffers, one with high pH, and the other low pH; intermediate pH levels were obtained by appropriately mixing the two sources. Each buffer thus had a viable range of pH levels, with partial overlap for some buffers (Acetate: 3.75 to 5.75, MES: 5.10 to 7.10, PIPES: 5.75 to 7.75, HEPES: 6.50 to 8.50). In each well, the robot mixed 40 µL of saline solution, 0-30 µL of buffer base, 0-30 µL of buffer acid, and 5 µL of protein, for a total volume of 75 µL. Samples were prepared in triplicate. The readout was performed on a Tecan M200 reader with an excitation wavelength of 488 nm and an emission wavelength of 510 nm.

Refolding. In the refolding assay, the robot dispensed 7 µL of sfGFP into an empty well, diluted it with 68 µL denaturing buffer (GuHCl; quanidinium hydrocholoride 7.5M), mixed by aspiration (3 x 50 µL), and allowed the protein to unfold at room temperature for 7 minutes. It then transferred three 4.5 µL samples to empty wells on a 384-well plate (flat bottom, black) and placed the plate into a Tecan Infinite M200 reader. For each of the three wells, fluorescent readouts were performed for five minutes every 0.3 s (excitation wavelength 475nm, emission wavelength 510nm bandwidth 20nm), and after two seconds the well received an injection of 86 µL diluent to permit refolding. Of the three experiments in this chapter, this is the most difficult one to perform manually because the measurements need to start within a fraction of a second after dilution.

Unfolding. For the unfolding assay, the robot mixed buffer (25mM Tris, pH 7.5) and denaturant (GuHCl) in a 384 well plate to a volume of 70 µL to obtain a range of denaturant concentrations from 0 M to 7 M. (We initial used urea as the denaturant, but the variability among replicates was too high.) The robot then added 5 µL of fluorescent protein, mixed by aspiration, and sealed the plate. Measurements were taken hourly for two days as follows: the plate was briefly centrifuged to return condensation on the seal back to the well, then fluorescence was measured, and finally the plate was returned to the centrifuge for incubation at 25 °C for the remainder of the hour. The original purpose of the 2-day time series was to provide time-resolved denaturation curves, but this measurement duration was too short for the mixtures to reach equilibrium. Had we measured long enough, it would have been possible to also estimate the rates of unfolding.

3.4.2 Experimental Limitations

Timing control. Roboliq is able to control timing quite accurately (Section 3.3.4), but there are limits: the timing cannot be faster than the equipment allows. As long as we work within the system's capabilities, however, the timing was shown to be very consistent, even for fairly complex operations.

Automatic Metadata. Roboliq's ability to automatically add metadata to measurements is an important advantage for analysis. There is a limitation, however: Roboliq currently only matches metadata with our *automated* measurements. Our robots do not have integrated scales, for example, so the weight measurements we will show in Chapter 4 were taken manually and then manually entered for analysis.

3.4.3 Data analysis

pH stability. For the pH experiment, the estimates of pKa were calculated by fitting sigmoid curves to each of the three runs using nonlinear mixed effects (NLME) [110] on this model:

$$f(pH) = \frac{a}{1 + \exp(-b(pH - pKa))} + \epsilon\,,$$

where pH is the independent variable, a is the height of the curve, b determines the slope around $pH = pKa$, pKa is the parameter of interest, and $\epsilon = \mathcal{N}(0, \sigma^2)$ is the error term with constant variance. The parameters a, b, and pKa are all random effects, and thus allowed to vary between the replicates. %The average pKa estimate is 5.86 with a 95% confidence interval of 5.74-5.97 and a 95% prediction interval of 5.66-6.06. The estimated pKa values for the three individual runs are 5.87, 5.78, and 5.92 (with 95% confidence intervals of 5.80-5.92, 5.72-5.84, and 5.84-5.96, as calculated by nonlinear least squares). The average pKa estimate has a 95% prediction interval of 5.66-6.06.

Folding rate. The data as shown in Figure 3.4C appears to have some outliers. We wanted to remove the outliers before fitting the stretched exponential to the data. Outliers were identified by the following procedure: (i) Remove the data points before dilution. (ii) Sum the time series of the three replicates together; each time point was summed, resulting in a single time series. (iii) Estimate the zero fluorescence level as the sum of measurements at the time point just before dilution. (iv) Subtract the zero fluorescence level from the summed series. (v) Fit a stretched exponential model [111] to the series: $f^{+}(t) = a \cdot (1 - exp(-k \cdot t^{c})) + \epsilon$. (vi) Identify outliers as the data points lying more than 2% below the fitted curve; this removed 81 of 2973 points. After the outliers were removed, we wanted to see how well the time courses of each replicate overlap, so we normalized each replicate by fitting to the linear model $\bar{f}(t) = (f^{+}(t)/a) + b + \epsilon$.

The initial refolding rate is usually estimated by the derivative of the curve where fluorescence recovery begins, but the derivative of a stretched-exponential model is undefined at $t = 0$. Therefore, we instead fit a non-stretched exponential ($c = 1$) to the initial 2 seconds of the data. We also need to account for the fact that the dilution operation took a fraction of a second to perform because that time could be significant on the scale of 2 seconds. The resulting model to estimate the initial folding rate was:

$$\bar{f}(t) = 1 - exp(-k \cdot (t + \delta)) + \epsilon_2$$

where k is the folding rate and δ is the time shift attributed to the dilution operation. Using NLME estimation with random effects k and δ, the average k value is $14.1\,\%\,s^{-1}$ with a 95% CI of $13.4 - 14.8\,\%\,s^{-1}$ and a 95% prediction interval of $13.0 - 15.3\,\%\,s^{-1}$. The average time shift δ is 0.19 s with a 95% CI of 0.079 – 0.31 s. The estimates for the individual replicates are 0.26 s, 0.11 s, and 0.21 s.

We also tried fitting the data to a model of sums of exponentials, but in contrast to a previous report [100], we did not find evidence that a three-exponential model was a reasonable choice: while such a model can fit the data exquisitely well, the slightest change to weighting leads to drastic parameter shifts.

ΔG of unfolding. For the unfolding experiment, we fit the following sigmoid model to the last data set measured after five days:

$$f(c) = \frac{a}{1 + (c/c_0)^h} + \epsilon\,,$$

where c is the input concentration of GuHCl denaturant, a is the amplitude, c_0 is the denaturation mid-point we are interested in, and h determines the slope around $c = c_0$. For the fitting, we used non-linear least squares instead of NLME because the data was taken from a single run. Note that the models here and for the pH experiments are mathematically equivalent; different re-parameterizations are for convenience.

We obtained an estimate of ΔG by fitting the data points around $c = c_0$ (from 2.5 M to 4.75 M) to a linear model:

$$-R \cdot T \cdot \log(a/f(c) - 1) = c + \epsilon$$

where R is the gas constant ($0.001\,987\,\mathrm{kcal\,K^{-1}\,mol^{-1}}$), T is the temperature in Kelvin ($25\,°C = 298.15\,K$), a is the curve's height from the previous sigmoid fit, f is the fluorescence measurement, and c is our denaturant concentration. Extrapolating from the data back to $c_0 = 0$, we get $\Delta G^{\circ} = 8.02\,\mathrm{kcal\,mol^{-1}}$, with a 99% prediction interval of $7.29 - 8.75\,\mathrm{kcal\,mol^{-1}}$.

3.5 Conclusions

We conclude that the proof-of-principle application demonstrates the high replicability and low variance that can be achieved through laboratory automation in quantitative assays for testing. All three experiments also show that Roboliq helps to efficiently automate a useful subset of experimental procedures with good reproducibility by handling complex pipetting series, time-critical procedures, long-duration experiments, matching of metadata and measurements, and facilitating a workflow that is conducive to repeatability.

Chapter 4

Quality Control of Robot Performance

4.1 Introduction

Lack of quality control in laboratories leads to false conclusions [112, 113]. Before drawing scientific conclusions based on experimental data, one should know the quality of the experiment's procedures, equipment, and measurements [114, 115]. Being able to predict the pipetting error in an automated protocol is helpful for deciding whether a robot can be used for a specific experiment, detecting when pipetting errors occur, detecting when maintenance is required, trouble-shooting unexpected results, correcting dispense volume bias, and improving analysis with more accurate estimates of performance and confidence intervals. Such quality assessments also play a role in better experimental designs [116] and enhancing the trust others can have in your results [117].

Our initial motivation for developing quality control routines arose because certain automated experiments were failing, and we had to track down the error sources. The search was complicated by the fact that the real-world performance characteristics for pipetting sometimes diverged significantly from those stated by the manufacturer. Therefore, we aimed to develop semi-portable methods to rigorously quantifying performance, to detect errors, and to correct biases. The performance quantification should assess measurement error, pipetting bias and variance (Figure 4.1), and undesired dilution during liquid transfers. To this end, we developed four inter-related components:

1. A mathematical pipetting model for liquid handling with fixed tips on our Tecan Freedom Evo robots;
2. Automated protocols for quality control using a balance and absorbance reader;
3. Analysis procedures to estimate the model parameters and their confidence intervals;
4. Software functionality to integrate the procedures in Roboliq.

The system we present here has several advantages over previous methods of quality control for liquid handling robots: It uses Markov Chain Monte Carlo (MCMC) analysis to allow for more complex protocols that obtain more information in less time [118, 119]; The protocol scripts are semi-portable due to the use of Roboliq; A range of protocols is provided for more diagnostic analysis; A JavaScript software module is provided with Roboliq that helps generate efficient MCMC models; and Roboliq can automatically correct for dispense bias when executing protocols.

The protocols were developed for the type of equipment and labware we use in our lab. They underwent extensive optimization that renders them less portable than typical Roboliq protocols. In particular, the protocols were developed on the following setup: Tecan Evoware robots, fixed tips, volumes between 3 and 1000 µL, pipetters with 4-8 syringe heads, 96-well plates, an integrated absorbance reader, and a balance that can weigh in 1 mg steps. Extending the protocols to other setups and volume ranges will require appropriate adaptation.

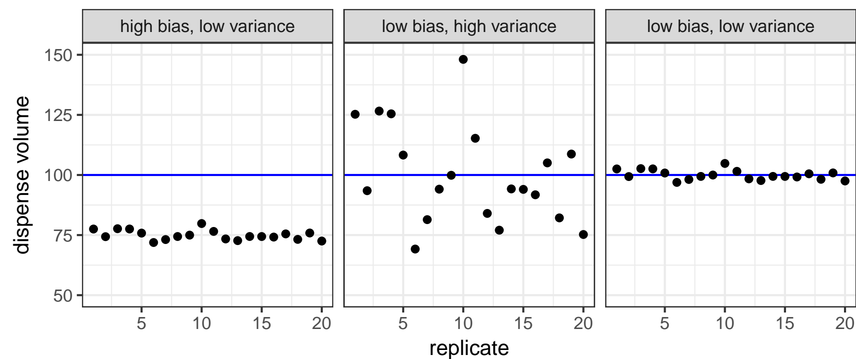


Figure 4.1: **Bias and variance.** In the context of dispense volumes, *bias* refers to how close the average of the true dispense volumes is to the desired volume — if they are close, then the bias is said to be low. *Variance* refers to how close the true dispense volumes are to each other for a given desired dispense volume — if they are close, then the variance is said to be low. **Left)** Shows an example of high bias but low variance — the volumes are close together, but the average is far from the desired volume of 100 µL. **Middle)** Shows an example of low bias but high variance — the volumes are spread very far apart, but their average is close the desired volume. **Right)** Show an example of both low bias and low variance, with all volumes close to the desired volume.

4.2 Theory

This section describes the physical systems (pipetting, weighing, measuring absorbance, z-level detection) and the mathematical formulas we used to model them. The model's Bayesian network is shown and explained in Figure 4.2. Below we list the model equations altogether; they will be explained in detail in the following sections.

$$\mathbf{VD}_\delta = d \cdot (1 + \beta_{p,d}) \cdot (1 + \epsilon(\mathrm{VD1}^{\sigma}_{p,s})) + \epsilon(\mathrm{VD0}^{\sigma}) \quad \text{dispense volume} \tag{4.1}$$

$$\mathbf{VU}_\delta = \mathrm{VU}^{\mu}_{p,d} \cdot (1 + \epsilon(\mathrm{VU}^{\sigma})) \quad \text{undesired dilution volume} \tag{4.2}$$

$$\mathbf{V}_{w,n} = \begin{cases} 0 & n = 0 \\ \mathbf{V}_{w,n-1} - \mathbf{VD}_\delta & w \text{ is source} \\ \mathbf{V}_{w,n-1} + \mathbf{VD}_\delta + \mathbf{VU}_\delta & w \text{ is destination} \end{cases} \quad \text{well volume} \tag{4.3}$$

$$\mathbf{C0}_k \sim \text{user-specified distribution} \quad \text{source concentration} \tag{4.4}$$

$$\mathbf{C}_{w,n} = \begin{cases} 0 & n = 0 \\ \mathbf{C}_{w,n-1} & w \text{ is source} \\ \frac{\mathbf{C0}_k \cdot \mathbf{VD}_\delta + \mathbf{C}_{w,n-1} \cdot \mathbf{V}_{w,n-1}}{\mathbf{V}_{w,n}} & \text{otherwise} \end{cases} \quad \text{well concentration} \tag{4.5}$$

$$\mathbf{G0}_l \sim \text{uniform distrubtion} \quad \text{empty labware weight} \tag{4.6}$$

$$\mathbf{G}_{l,N} = \mathbf{G0}_l + \rho \cdot \sum_w \mathbf{V}_{w,n} \quad \text{labware weight} \tag{4.7}$$

$$\mathbf{g}_{l,N} = \mathbf{G}_{l,N} + \epsilon(\sigma_g) \quad \text{weight measurement} \tag{4.8}$$

$$\mathbf{AL}_l = \mathrm{AL}^{\mu}_m + \epsilon(\mathrm{AL}^{\sigma}_m) \quad \text{labware absorbance} \tag{4.9}$$

$$\mathbf{AW}_w = \mathbf{AL}_l + \epsilon(\mathrm{AW}^{\sigma}_m) \quad \text{empty well absorbance} \tag{4.10}$$

$$\mathbf{A0}_w = \mathbf{AW}_w + \mathrm{A0}^{\mu}_m + \epsilon(\mathrm{A0}^{\sigma}_m) \quad \text{water-filled absorbance} \tag{4.11}$$

$$\mathbf{A}_{w,n} = \begin{cases} \mathbf{AW}_w & n = 0 \\ \mathbf{A0}_w + \mathbf{V}_{w,n} \cdot \mathbf{C}_{w,n} & \text{otherwise} \end{cases} \quad \text{well absorbance} \tag{4.12}$$

$$\mathbf{a}_{w,n} = \mathbf{A}_{w,n} + \epsilon(\sigma_a) \quad \text{absorbance measurement} \tag{4.13}$$

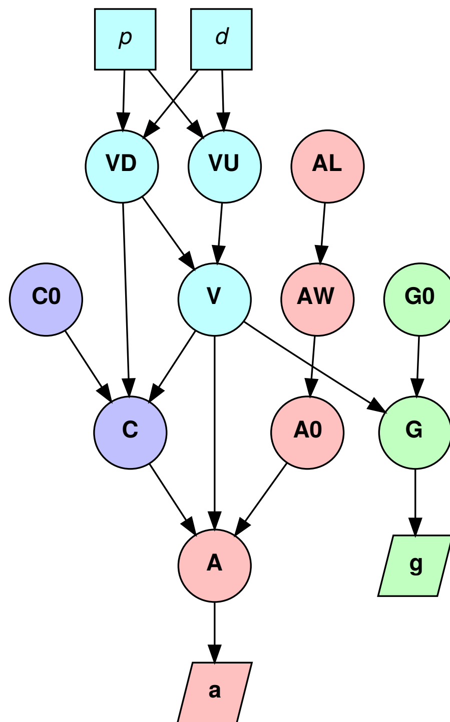
$$\epsilon(\sigma) \sim \mathcal{N}(0, \sigma^2) \quad \text{normal error} \tag{4.14}$$

A note about notation: Random variables are typeset in bold — hidden random variables are uppercase, and measured random variables are lowercase. Fixed parameters are typeset in lower-case Roman letters, except for two fixed parameters that use Greek letters: ρ is the liquid density, δ is a "super-parameter" that represents the set of all parameters for a single dispense. ϵ indicates an error term. All other uses of Greek letters indicate parameters we want to estimate (e.g. β and σ_a, $\mathrm{VD1}^{\sigma}_{p,s}$ and $\mathrm{A0}^{\mu}_m$). The fixed parameters have the following meanings:

- l: labware (e.g., a 96-well transparent bottom plate)
- m: labware model (i.e., a type of plate or tube)
- w: well on labware (e.g., A11)
- s: a subclass of p
- k: liquid (e.g., water, buffer, dye)

The relationship between w, l, and m is illustrated in Figure 4.3. There are actually two wells for each pipetting action — a source well and a destination well — but we do not distinguish them here. There are also two index variables:

- n: the number of dispenses in a well
- N: index for a step an experiment



Control factors:

- *p*: pipetting parameters ("liquid class")
- *d*: desired dispense volume

Volume variables:

- **VD**: volume of dispense
- **VU**: volume of undesired dilution
- **V**: volume of well

Concentration variables:

- **C0**: concentration of source liquid
- **C**: concentration of well

Absorbance variables:

- **AL**: mean absorbance of empty labware
- **AW**: absorbance of empty well
- **A0**: absorbance of water-filled well
- **A**: absorbance of well with dye
- **a**: absorbance measured

Weight variables:

- **G0**: weight of empty labware
- **G**: weight of non-empty labware
- **g**: weight measured

Figure 4.2: **Bayesian network representation of the pipetting model.** In this graph, the node shapes have the following meanings: squares represent control factors, circles represent hidden variables, and parallelograms represent observables. The colors indicate different aspects of the model: cyan for volumes, purple for concentrations, red for absorbance, and green for weight. To dispense an aliquot, the user specifies the desired volume (d) and the pipetting parameters (p). When the robot dispenses the aliquot, it will consist of a volume of the source liquid (**VD**) and possibly an undesired volume of system water (**VU**), and one or more dispenses accumulate in a well to form the well volume (**V**). Each source liquid has a concentration (**C0**), and the concentration in a well is dependent on the concentration of the source dispenses and the undesired dilution (**C**). Absorbance is built on a chain of variables: each labware has an average absorbance of its empty wells (**AL**), each well has its own base absorbance (**AW**), the water volume in a well shifts the dye-free absorbance (**A0**), the well absorbance increases proportional to the amount of dye in the well (**A**), and the absorbance measurement is subject to measurement errors (**a**). The weight variables consist of the empty labware weight (**G0**), the labware weight with liquid (**G**), and the weight measurements that are subject to measurement error (**g**).

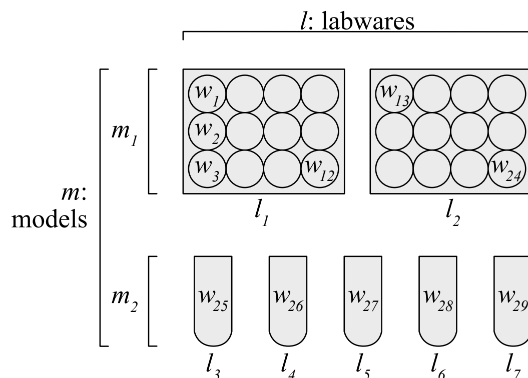


Figure 4.3: **Labware models m, labwares l, and wells w.** An experiment may use multiple labware models m, such as the plate model m_1 and the tube model m_2 shown here. An experiment may use any number of labwares l of a given model, such as the two plates l_1 and l_2 of m_1 and the 5 tubes of m_2. Each labware may have one or more wells w, such as the 29 wells shown here.

4.2.1 Pipetting: Aspiration and Dispense

In contrast to most liquid-handling robots, Tecan pumps water instead of air within their syringes. There are trade-offs to this choice. Because water compresses much less than air, its use allows for more precise and consistent control of pipetted volumes. On the other hand, it adds complexity to the system and may result in some dilution of aspirated aliquots with the syringe water.

The Tecan Evoware robot first fills a syringe by pushing water through it (which is discarded in a waste receptacle), then retracts the water slightly to leave a small air gap at the end of the tip. The air gap should maintain a separation of the syringe water from any aspirated liquids. To aspirate a liquid from a well, Evoware places the tip in the liquid and retracts the syringe water to draw in the well liquid. To dispense the liquid, the syringe water is pressed down enough to expel the aspirated liquid, while the air gap prevents the dispense of the water in the syringe.

For Evoware to perform a pipetting operation, the user must specify the liquid source, destination, volume, tip, and something Evoware calls the "liquid class". A "liquid class" is basically a method for pipetting; For example, liquid classes can be specialized for dispensing water from above a well without bringing the tip in contact with the well's contents (we call this "air" dispense). An Evoware liquid class is composed of several "subclasses", each one covering a certain range of volumes (Figure 4.4A). In this thesis, we will deal with three subclasses for the following ranges: 3–15 µL, 15–500 µL, and 500–1000 µL.

Since each Evoware liquid subclass has its own set of parameters, we will analyze and control the pipetting performance of each subclass separately. In particular, we want to ascertain the true volumes dispensed and the unintended dilution that takes place within the tips due to the imperfect separation between the syringe water and the pipetted aliquot (Figure 4.5).

For larger volumes (>=150 µL), we find both the volumes dispense and the dilution factors. For lower volumes, however, there are technical and practical challenges to determining unintended dilution using our equipment. Therefore, the concept of "volume" here will refer to the volume of the source aliquot that actually gets delivered to the destination well - but if dilution takes place, the total volume dispensed will be higher.

4.2.2 Pipetting: Volume of Dispense

Equation (4.1) is the true dispense volume, and it is perhaps the most important variable in the model. It represents the amount of source liquid that ends up in the dispensed well. Here we repeat the equation for convenience and explain its composition:

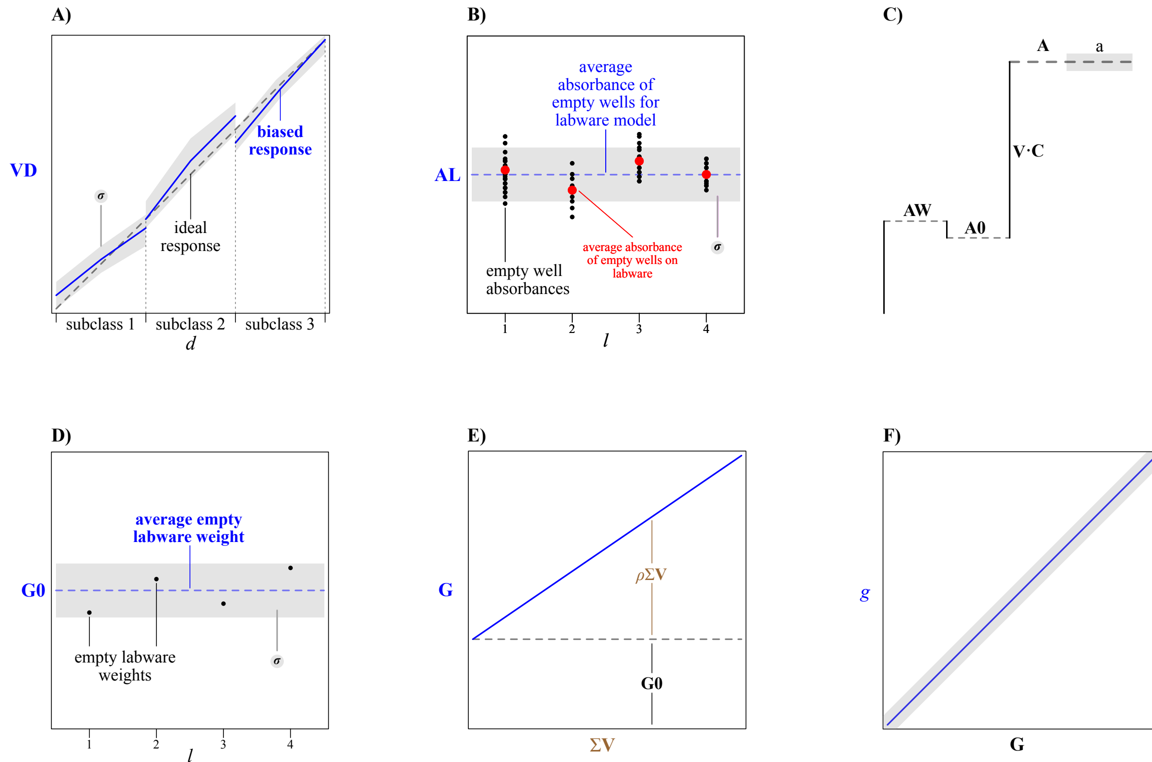


Figure 4.4: **Illustrations of model variables.** In these illustrations, the shaded regions represent variance. **A)** Dispense volume **VD** as a function of desired volume d. Here we illustrate a liquid class with three subclasses. The x-axis represents the desired dispense volume. The y-axis represents the true dispense volume. The dashed line is the ideal 1-to-1 response where $\mathbf{VD} = d$. The blue lines represent the average true volume dispensed; in practice, there will probably be some bias such that the true volume tends to be higher or lower than the desired volume. **B)** Empty well absorbance. The x-axis represents different plates of the same labware model; the y-axis represents absorbance. The black dots indicate the absorbance of empty wells. The red dots show the mean absorbance of empty wells on an individual plate. **C)** Absorbance **A** begins with the empty well absorbance **AW**, drops slightly to the water-filled absorbance **A0**, and increases proportionally to the dye in the well $\mathbf{V} \cdot \mathbf{C}$. The measured absorbance **a** has a spread around the true absorbance **A**. **D)** Distribution of empty labware weights **G0** around an average weight for the labware model. **E)** Labware weight **G** as the sum of the empty labware weight plus the weight of liquid dispenses. The x-axis represents the liquid volume in the labware. **F)** Measured weight **g** as function of true labware weight **G**.

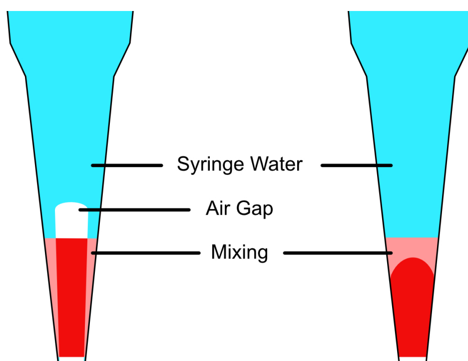


Figure 4.5: **Illustration of unintended dilution within pipetting tips.** (Left) Syringe water (blue) is retracted to pull up the aliquot (red), but some water may stick to the sides and mix with the aliquot (pink). (Right) Under certain circumstances, the air gap between syringe water and aliquot can be compromised, leading to direct contact and mixing of the two liquids.

$$\mathbf{VD}_\delta = d \cdot (1 + \beta_{p,d}) \cdot (1 + \epsilon(\mathrm{VD1}^\sigma_{p,s})) + \epsilon(\mathrm{VD0}^\sigma) \quad \text{dispense volume} \quad (4.1^*)$$

- δ is the dispense specification, including volume, source, destination, and pipetting parameters.
- d is the desired volume that is specified by the user in δ.
- p is the selected pipetting parameters (Tecan calls this the *liquid class*).
- $\beta_{p,d}$ is the bias parameter representing the percentage of bias. It is index by p and d. The QC protocols in this chapter provide estimates for selected volumes in each subclass. If a non-tested volume is used, then $\beta_{p,d}$ is not a parameter, but rather a linear interpolation/extrapolation using the two closest d values for which we have estimates.
- $d \cdot (1 + \beta_{p,d})$ is the biased dispense volume. For example, if $d = 100\,\mu\text{L}$ and the bias is $5\,\%$, then the value here would be $105\,\mu\text{L}$.
- $\mathrm{VD0}^\sigma$ and $\mathrm{VD1}^\sigma_{p,s}$ are the variance parameters, broken into constant and proportional components. $\mathrm{VD0}^\sigma$ is the constant spread over pipetting operations — physically, this can be attributed to the Robot's step motors in the syringe pumps $\mathrm{VD1}^\sigma_{p,s}$ is the proportional spread that scales with d — we find that the spread is larger for larger volumes than smaller ones.
- $\epsilon(...)$ represents a normal random spread with the given standard deviation.

The bias β is shown in Figure 4.4A where the colored lines diverge from the dashed line; The spread is shown as the shaded regions.

4.2.3 Pipetting: Volume of Undesired Dilution

Equation (4.2) is the unintended volume of system water that gets dispensed along with the source liquid. In the ideal case, this value is 0.

$$\mathbf{VU}_\delta = \mathrm{VU}^\mu_{p,d} \cdot (1 + \epsilon(\mathrm{VU}^\sigma)) \quad \text{undesired dilution volume} \quad (4.2^*)$$

- $\mathrm{VU}^\mu_{p,d}$ is the mean volume for liquid class p and desired volume d. This parameter is similar to $\beta_{p,d}$ (described in the previous section) insofar as it may be either a parameter or a function, depending on the value of d.
- VU^σ is the proportional spread of undesired dilution volumes.

4.2.4 Pipetting: Volume of Well

Equation (4.3) is the total volume in a well — it represents the sum of the first n dispenses in the well.

$$\mathbf{V}_{w,n} = \begin{cases} 0 & n = 0 \\ \mathbf{V}_{w,n-1} - \mathbf{VD}_\delta & w \text{ is source} \\ \mathbf{V}_{w,n-1} + \mathbf{VD}_\delta + \mathbf{VU}_\delta & w \text{ is destination} \end{cases} \quad \text{well volume} \quad (4.3^*)$$

- w is the well.
- n tells us to consider the first n dispenses in well w.
- If $n = 0$, the well volume is 0 µL.
- If w is the source for the nth dispense, we subtract the true dispense volume from the prior volume.
- Otherwise, we add the volumes of the source liquid and undesired system water to the prior volume.

In the second case, the volume may become negative in source wells, but this is not problematic because we do not need to track their volume. The case also ignores the fact that the volume removed from the source could be larger than $\mathbf{VD}_\delta$, but we decided to accept this simplification because we are more interested in characterizing dispenses than aspirations.

The model is time invariant and assumes that no evaporation or other time-related phenomena affect the well volume. The protocols in this chapter respect the assumption by avoiding long time durations in which evaporation could become significant.

4.2.5 Pipetting: Concentration in Well

The QC experiments use two liquids: water and dye. The dye is used for the absorbance readouts, and we calculate its concentration in terms of its absorbance per microliter. For general experiments, we would need a better way to express concentration, since this concentration value would vary depending on labware; we only use a single labware model for the readouts, however, so the simplification is reasonable. Equations (4.4) and (4.5) model concentration.

$$\mathbf{C0}_k \sim \text{user-specified distribution of source concentration} \quad (4.4^*)$$

- k is a liquid — either water or dye. The model can only handle a single dye stock, but any number of dilutions of that stock may be used as dye sources.
- $\mathbf{C0}_k$ is the variable for the concentration of liquid k. For water, the concentration is $0\,\mu\text{L}^{-1}$. For dye sources, the user can either specify a fixed value, a normal distribution, or a uniform distribution. If the concentration is not fixed, our estimation routines will try to estimate it.

$$\mathbf{C}_{w,n} = \begin{cases} 0 & n = 0 \\ \mathbf{C}_{w,n-1} & w \text{ is source} \\ \frac{\mathbf{C0}_k \cdot \mathbf{VD}_\delta + \mathbf{C}_{w,n-1} \cdot \mathbf{V}_{w,n-1}}{\mathbf{V}_{w,n}} & \text{otherwise} \end{cases} \quad \text{well concentration} \quad (4.5^*)$$

- If the well is empty, the well concentration is $0\,\mu\text{L}^{-1}$.
- If w is the source for the nth dispense, we leave its concentration unchanged.
- Otherwise, we add the amount of dye in the current dispense δ to the amount that was already in the well, then divide by the new well volume.

$\mathbf{C}_{w,n}$ is only calculated for wells that receive dispenses, not for source-only wells.

4.2.6 Weight

We can use weight measurements to help determine well volumes, if we have the following pieces of data:

- weight of container before dispense
- weight of container after dispense
- temperature of environment
- density of liquid

Since liquid density is dependent on temperature, we need to record the laboratory temperature and look up the density of water in an appropriate table. (We did not measure a difference in the weight of water-filled vs. dye-filled tubes, so we used the well-known density of water for both liquids.) We can then calculate the liquid volume in the labware as:

$$\text{total volume} = \frac{\text{weight after} - \text{weight before}}{\text{density at temperature}}$$

Equations (4.6) – (4.8) model labware weight and balance measurements, and they are illustrated in Figure 4.4D,E,F.

$$\begin{aligned}
\mathbf{G0}_l &\sim \text{uniform distrubtion} && \text{empty labware weight} && (4.6^*)\\
\mathbf{G}_{l,N} &= \mathbf{G0}_l + \rho \cdot \sum_w \mathbf{V}_{w,n} && \text{labware weight} && (4.7^*)\\
g_{l,N} &= \mathbf{G}_{l,N} + \epsilon(\sigma_g) && \text{weight measurement} && (4.8^*)
\end{aligned}$$

- $\mathbf{G0}_l$ is the weight of the empty labware before any liquid is dispensed into it. We used a uniform distribution because we found that our estimator quickly converged, but different experiments might require a more informative prior.
- l is the labware being weighed.
- N is an index to a step in the protocol.
- $\mathbf{G}_{l,N}$ is the weight of the labware. It is comprised of the empty weight the sum of the weights of all well volumes on the labware at the step where the measurement was taken.
- ρ is the density of water at lab temperature.
- $\sum_w$ iterates over all wells on labware l.
- $\mathbf{g}_{l,N}$ is the measured weight. This is comprised of the labware weight plus a normal error term.
- σ_g is the measurement error of the balance.

4.2.7 Absorbance

There is a straightforward way to estimate the expected dispense volume for a desired volume d: instruct the robot to dispense d multiple times onto a plate, then use the increase in the plate's weight to calculate the average volume dispensed. Estimating the variance, however, is not practical with a balance in our lab, because it would require many measurements, and our robots do not have an integrated balance.

Instead, we make extensive use of absorbance readouts for these experiments. With the absorbance readouts, we can determine relative dispense volumes and variances of many dispenses at once (e.g., by using 96 wells on a 96-well plate). Absorbance is a very complex phenomenon, however, involving these factors (Figure 4.6):

- measurement error of the absorbance reader
- absorbance of the plate material itself
- impact of the liquid on the light beam
- the shape of the liquid surface ("meniscus")

For the other factors, we developed experiments to determine their magnitudes. However, we omit the meniscus effects from most analyses. To minimize the variance introduced by meniscus effects, we take two steps: 1) The main experiments always measure absorbance at the same volume level or very close to it; 2) We shake the plate before absorbance readouts. We found that the shaking significantly reduced the variance of random meniscus shifts among wells.

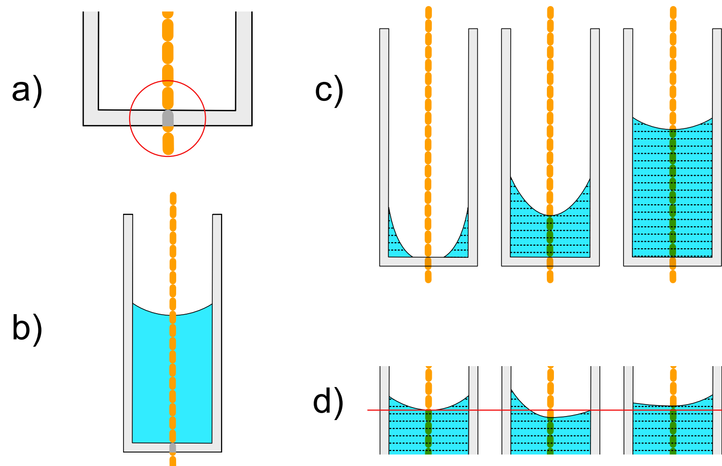


Figure 4.6: **Nuisance factors that affect absorbance measurements.** Absorbance is impacted by various factors, several of which are illustrated here. A) The plate material itself will absorb some light. B) A water-filled well will have a different readout than an empty one. In our experiments, absorbance measurements in water-filled wells were lower than in empty wells. This phenomenon was not examined further but accounted for in the analyses. C) The shape of the meniscus (the liquid surface) changes as the well is filled, which alters the percentage of the liquid under the beam, and that changes the absorbance readout accordingly. D) The shape of the meniscus can also vary in a random way, such as being shifted more to one side of the well.

There is one more important factor: how well-mixed the liquids are. We will omit the last factor from our models by assuming that liquids are well-mixed; we try to ensure this by proper shaking and mixing based on experience.

4.2.8 Absorbance of Empty Wells

Each labware model has a certain "base absorbance" of its empty wells (Figure 4.6A and 4.4B, Equation (4.9) and (4.10)). Individual plates of a given model will each have their own average absorbance. And within a plate, the absorbance of individual wells will also vary.

$$\begin{aligned} \mathbf{AL}_l &= \mathrm{AL}^{\mu}_m + \epsilon(\mathrm{AL}^{\sigma}_m) && \text{labware absorbance} && (4.9^*) \\ \mathbf{AW}_w &= \mathbf{AL}_l + \epsilon(\mathrm{AW}^{\sigma}_m) && \text{empty well absorbance} && (4.10^*) \end{aligned}$$

- m is the labware model of l.
- AL^{μ}_m is the parameter for the mean average absorbance among all labwares of model m.
- AL^{σ}_m is the spread of the base absorbance on all labwares of model m.
- $\mathbf{AL}_l$ is the mean absorbance of all wells on labware l.
- AL^{σ}_m is the spread of absorbance among empty wells of labware model m.
- $\mathbf{AW}_w$ is the absorbance of well w when empty.

4.2.9 Absorbance of Water-Filled Wells

We found that the act of putting water into an empty well changes its absorbance readout (Equation (4.11)). While the change is not very large, but it becomes proportionally significant when using low dye concentrations. To calculate the impact of dye on absorbance, we need to subtract this $\mathbf{A0}_w$ variable from the measurement $\mathbf{a}_{w,n}$.

$$\mathbf{A0}_w = \mathbf{AW}_w + \mathrm{A0}^\mu_m + \epsilon(\mathrm{A0}^\sigma_m) \quad \text{water-filled absorbance} \quad (4.11^*)$$

- $\mathrm{A0}^\mu_m$ is the average change in absorbance relative to the empty-well absorbance ($\mathbf{AW}_w$) when a well is filled with water to the reference volume. For the experiments in this chapter, we used 300 µL well volumes.
- $\mathrm{A0}^\sigma_m$ is the spread of absorbance changes at the reference volume.
- $\mathbf{A0}_w$ is the absorbance of dye-less well w when filled to the reference volume.

4.2.10 Absorbance of Wells

Finally we consider the total well absorbance and then the absorbance measurement (Figure 4.4C):

$$\mathbf{A}_{w,n} = \begin{cases} \mathbf{AW}_w & n = 0 \\ \mathbf{A0}_w + \mathbf{V}_{w,n} \cdot \mathbf{C}_{w,n} & \text{otherwise} \end{cases} \quad \text{well absorbance} \quad (4.12^*)$$

- $\mathbf{A}_{w,n}$ is the absorbance of well w after n dispenses.
- In the first case, where $n = 0$ and nothing has been dispensed into the well, the absorbance is the empty-well absorbance ($\mathbf{AW}_w$).
- Otherwise, the total absorbance is the sum of the water-only absorbance ($\mathbf{A0}_w$) and the absorbance of the dye in the well ($\mathbf{V}_{w,n} \cdot \mathbf{C}_{w,n}$).

$$a_{w,n} = \mathbf{A}_{w,n} + \epsilon(\sigma_a) \quad \text{absorbance measurement} \quad (4.13^*)$$

- $\mathbf{a}_{w,n}$ is the absorbance measured in well w after n dispenses. This is simply the well absorbance $\mathbf{A}_{w,n}$ plus measurement error.
- σ_a is the measurement error.

Note that the experiments in this chapter often performed replicate absorbance measurements. Replicates are not explicitly indexed here because our estimation method does not require any special specification for replicate measurements.

4.2.11 Z-level

In Evoware, a z-level measurement is performed by lowering the pipetter tip into a well until a change in electrical resistance is detected in its conductive tips. Evoware then calculates the liquid volume based on the height of detection — unfortunately the calculations are very poor due to software limitations for specifying the well's shape. The major factors that impact z-level measurements are:

- the labware model
- the position of the labware on the lab bench
- the volume within the well

The labware model impacts measurements because different labware models have different well shapes and bottom offsets (i.e., the z-level corresponding to an empty well). The bench position impacts measurements because different positions have different altitudes; different positions may also have labware holders that raise certain labware models relative to others. For example, a labware holder may have a raised center to help ensure certain labwares slip into place reliably, but the raised center prevents other labware models from fully dropping into the holder.

To fit the z-level parameters, we use a simple linear model:

$$z_{w,n,h} = \mathrm{Z}^\alpha_m \cdot \mathbf{V}_{w,n} + \mathrm{Z}^\beta_{m,h} + \epsilon(\sigma_{z,m})$$

- $z_{w,n,h}$ is a z-level measurement at well w and bench position h for the first n dispenses.

- Z^{α}_{m} is the scaling relationship between z-level and volume for labware model m. Ideally, it will be close to 1, indicating that z-level tracks volume closely.
- $Z^{\beta}_{m,h}$ is the z-level offset of an empty well for labware model m at bench position h.
- $\sigma_{z,m}$ is the measurement error for model m.

4.2.12 Computation of model parameters

Depending on the complexity involved, our parameter estimations are calculated using standard linear regression, linear mixed effects method, or MCMC.

MCMC. MCMC stands for Markov Chain Monte Carlo [118]. It is a simulation technique for finding parameter estimates in models that are too complex to apply linear statistical methods [119]. It varies the model parameters in a random-walk fashion and calculates the likelihood of the measured data given those parameter values. By simulating many iterations while giving preference to parameter sets that result in higher likelihoods, we obtain a joint posterior probability distribution over the parameters, from which we can calculate their means and confidence intervals.

Although MCMC can solve statistical problems that are infeasible with traditional statistical approaches, it also brings its own set of challenges. The biggest problem for me was formulating the models to allow efficient exploration of the parameter space. When the search is not efficient, the simulation time quickly becomes prohibitive. A key part of the solution is to formulate a *centered parameterization* [120, 121], which places the centered parameters into a standard normal space that can be much more effectively explored by MCMC simulation. For example, a normal random variable μ would be decomposed as follows:

- μ_{loc}: an user input guessing the expected value of μ.
- μ_{scale}: a user input for the standard deviation around μ_{loc} which should be searched for the parameter value.
- μ_{raw}: the new standard normal parameter ($\mathcal{N}(0, 1)$).
- $\mu = \mu_{loc} + \mu_{raw} \cdot \mu_{scale}$: the calculated value of μ.

The second challenge was that some of these protocols involve a considerable number of hidden random variables and complex interactions. It became too complex to humanly construct error-free MCMC models. To solve this, it was necessary to add a library to Roboliq that helps generate the models for the excellent Stan MCMC solver [121].

Sample Sizes. Due to the nonlinearity of the models of the experiments, there is no straightforward method to calculate the sample sizes required to obtain good parameter estimates. Instead, we selected a sample size, ran a loop of 1000 simulations and analyses, and then evaluated the distributions of the parameter estimates. Depending on the results, we then changed the sample size and re-ran the loop until an acceptable trade-off was achieved between resource usage and precision of parameter estimates.

4.3 Results and Discussion

4.3.1 Estimates of Pipetting Parameters

Our estimates of the pipetting parameters are shown in Table 4.1, and the data for the estimates are shown in Figure 4.7A. The dispense bias ($\beta_{p,d}$) for the 150 µL dispenses in all liquid classes (Air, Dry, Wet) was around 3 % and statistically significant. The “Dry” liquid class shows statistically significant bias at most tested volumes, especially in the low volumes. The “Wet” liquid class, on the other hand, showed the least bias in the low volumes. The strong negative bias of the “Dry” dispenses at low volumes appeared to arise because some of the dispensed liquid stuck to the tip rather than all going into the well. This was probably the case with “Air” dispense as well, but to a lesser extent due to the faster rate of expulsion. The “Wet” dispenses were presumably able to dispense the entire volume due to their contact with water in the well.

Table 4.1: **Estimates of pipetting parameters.** These are the parameter values estimated by MCMC. Column p is the liquid class, s is the subclass index, d is the desired dispense in µL, "value" is the estimated parameter value, "se" is the standard error of the estimate, and the stars in the "p-value" column indicate statistical signficance (i.e. ***: $p < 0.001$, **: $p < 0.01$, * = 0.05) under the null hypothesis that the parameter is equal to zero.

parameter	units	p	s	d	value	se	p-value		description
$\beta_{p,d}$	%								dispense bias
		Air	1	3	-8.67	2.000	0.0000	***	
		Air	1	7	1.78	1.160	0.0625		
		Air	1	15	2.27	0.613	0.0001	***	
		Air	2	16	0.18	0.948	0.4247		
		Air	2	150	2.72	0.157	0.0000	***	
		Dry	1	3	-10.14	2.057	0.0000	***	
		Dry	1	7	-2.42	1.173	0.0196	*	
		Dry	1	15	2.36	0.900	0.0044	**	
		Dry	2	16	-0.09	1.067	0.4664		
		Dry	2	150	3.77	0.604	0.0000	***	
		Wet	1	3	-1.75	2.004	0.1913		
		Wet	1	7	0.33	1.190	0.3908		
		Wet	1	15	2.50	0.883	0.0023	**	
		Wet	2	16	-0.63	0.943	0.2520		
		Wet	2	150	2.83	0.457	0.0000	***	
$VD0^{\sigma}$	µL								dispense s.d. (constant)
					0.23	0.024	0.0000	***	
$VD1^{\sigma}_{p,s}$	%								dispense s.d. (proportional)
		A	1		0.41	0.268	0.0630		
		A	2		0.93	0.259	0.0002	***	
		D	1		0.52	0.342	0.0642		
		D	2		1.79	0.395	0.0000	***	
		W	1		0.53	0.323	0.0504		
		W	2		1.14	0.233	0.0000	***	
$VU^{\mu}_{p,d}$	µL								dilution volume
		A	2	150	1.10	0.217	0.0000	***	
VU^{σ}	%								dilution spread
					13.91	13.927	0.1590		

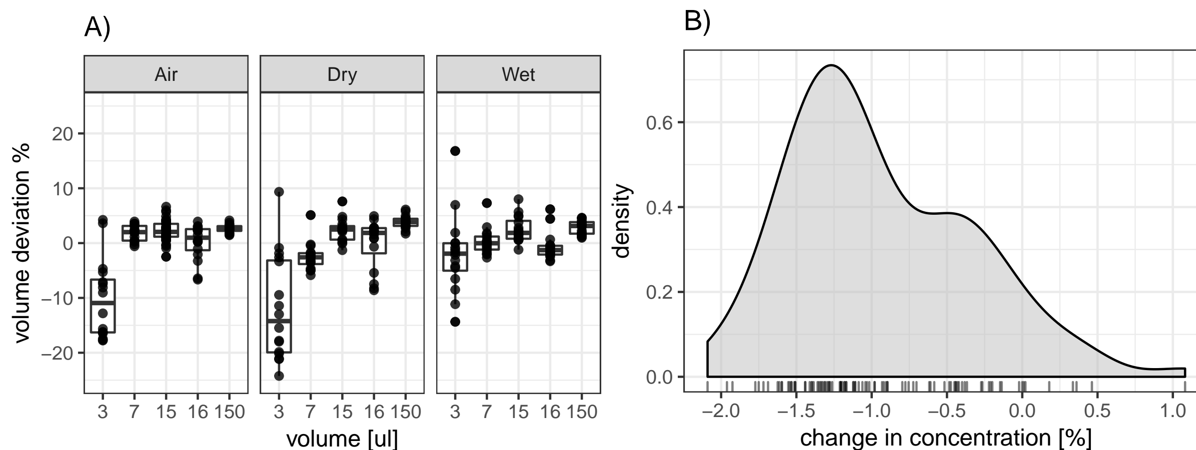


Figure 4.7: **Graphs of pipetting performance. (A) Deviation from desired dispense volume.** The x-axis represents volume in microliters; The y-axis represents the deviation from the desired dispense volume in percent. The panels are for the three liquid classes (air, dry, and wet dispense methods). Each dot represents one volume/liquid class combination, with 16 wells per combination. The 3 µL dry dispenses have the greatest deviation of about -10%, meaning that on average less than 2.7 µL dye gets dispensed into the well. For volumes >3 µL, the average bias is within about 3%. **(B) Dilution of successive liquid transfers.** This density plot shows the change in concentration between liquid transfers performed with the “Air” liquid class at 150 µL. The x-axis shows the percentage change in absorbance from one step to the next; The y-axis shows the density. The peak is around −1.25 %. The data includes points whose concentration appears to have *increased* — this is not possible and can be attributed to various aspects of pipetting and measurement error.

The constant dispense error ($\mathrm{VD0}^{\sigma}$) was estimated around 0.25 µL. The proportional dispense error ($\mathrm{VD1}^{\sigma}_{p,s}$) could only be estimated with statistical significance for subclass 2 and is around 1-2 %.

The unintended dilution for “Air” dispense at 150 µL was estimated around 1 µL. Due to the small value, its variance could not be reliably estimated without increasing the sample size significantly, which would not be worth the effort. Figure 4.7B shows the dilution data.

These results have several practical implications for protocols run on this robot. 1) The pipetting bias varies by a few percentage points between volumes ≥ 7 µL; ideally, it should be corrected in the control software or corrected by Roboliq, but such small differences can also be ignored for most applications. 2) The pipetting variance is very low, so we can be confident when using the robot for high-precision pipetting of volumes ≥ 7 µL. 3) Due to the high variance and bias of 3 µL dispenses, they should be avoided; when required, however, wet dispense is the most reliable method. 4) The unintended dilution of 1% at 150 µL can be safely disregarded for most applications, but it should be taken into account for series dilutions (e.g., a 10-step dilution series could be about 10% more dilute than expected).

4.3.2 Estimates of Reader Parameters

There was only one weight parameter: the measurement error σ_g has an esimated value of 0.75 mg with a 95% CI of 0.56 – 0.88 mg, while the balance measures in discrete steps of 1 mg.

The absorbance parameters are shown in table Table 4.2, and the data for $\mathrm{AW}^m u_m$ is shown in Figure 4.8. The measurement error ($\sigma_a \approx 0.006$) is worth discussing futher. First of all, it is a combination of both reader error and the meniscus variance that arises from dispenses and plate movements. We initially distinguished these two variances, but later merged them to simplify the model. Secondly, a common rule-of-thumb in our department is to keep the absorbance measurements above 0.1 to obtain reliable measurements. However, the reader error allows us to reliably measure much smaller absorbances, especially if we measure the empty-well

Table 4.2: **Estimates of absorbance parameters.** These are the parameter values estimated by MCMC. We only used one labware model m in these experiments, so the model column is omitted. "value" is the estimated parameter value, "se" is the standard error of the estimate, and the stars in the "p-value" column indicate statistical signficance (i.e. ***: $p < 0.001$, **: $p < 0.01$, * = 0.05) under the null hypothesis that the parameter is equal to zero.

parameter	value	se	p-value		description
AL_m^μ	0.04742	0.00021	0.0000	***	average model absorbance
AL_m^σ	0.00031	0.00035	0.1879		spread among labwares
AW_m^σ	0.00048	0.00003	0.0000	***	spread among wells
$A0_m^\mu$	-0.00838	0.00027	0.0000	***	water-filled absorbance change
$A0_m^\sigma$	0.00135	0.00022	0.0000	***	spread among water-filled wells
σ_a	0.00581	0.00014	0.0000	***	measurement error

absorbance first (Figure 4.13).

The other standard deviations, AL_m^σ and AW_m^σ are so small that one would normally remove them from the model. However, these values are actually much smaller than the true population averages, because I pre-screened the plates for low variances. There are also other labware models with much higher variances than the one we used.

4.3.3 Correction of Small-Volume Bias

Small aliquots are often the most critical and costly components in an experiment. Using the previously estimated parameters, we aimed to correct the dispense biases in the low-volume range of 3, 5, 7 µL, as shown in Figure 4.9A and B. This omitted the Wet liquid class from the test because its bias was not statistically significant. For the Air dispenses, the bias estimates were respectively -6.36%, -1.60%, and -1.91%; for the Dry dispenses, they were 2.05%, 0.49%, and 0.13%. Compared to Table 4.1 and Figure 4.7, bias was reduced in 3 of 4 cases; in the one case where it got worse, the difference was minor (from 1.78% bias to -1.91% bias for 7 µL Air dispense). Interestingly, the 3 µL dispenses are now biased up rather than down; the relatively large spread of their volumes indicates that we would generally want to avoid 3 µL dispenses on this robot. The 5 µL dispenses were practically unbiased, indicating that the correction was reasonable here, even though the volume was not previously tested; the spread is also good, from which we conclude that the robot can reliably pipette 5 µL. In real experiments, therefore, we avoided dispense volumes less than 5 µL.

4.3.4 Estimates from Supplementary Protocols

We developed a collection of supplementary protocols for further diagnostics, whose results are described here and in Appendix G.

Bias and Variance of Larger Volumes. In the main protocols, we only tested volumes up to 150 µL, because the well volumes are limited by the absorbance reader. But we also want to know the bias and variance of larger dispense volumes for experiments that require them. Figure 4.10 shows the bias and variance estimates for subclasses 2 and 3 of the "Air" liquid class as obtained by weight measurements. It includes the larger volumes that we did not test previously (450, 501, 750, and 950 µL). The figure supports our assumptions about subclasses: the bias and deviation are linearly dependent on d, and the subclasses are independent of each other. Note that these estimates are intended as diagnostic orientation points. Due to the low number of measurements per volumes, they are not as accurate as the main protocols, nor do they account for undesired dilution. Their value resides in determining whether the pipetting performance is reasonable for larger volumes: in this case, the bias and variance estimates are low enough not to require further fine-tuning for our applications.

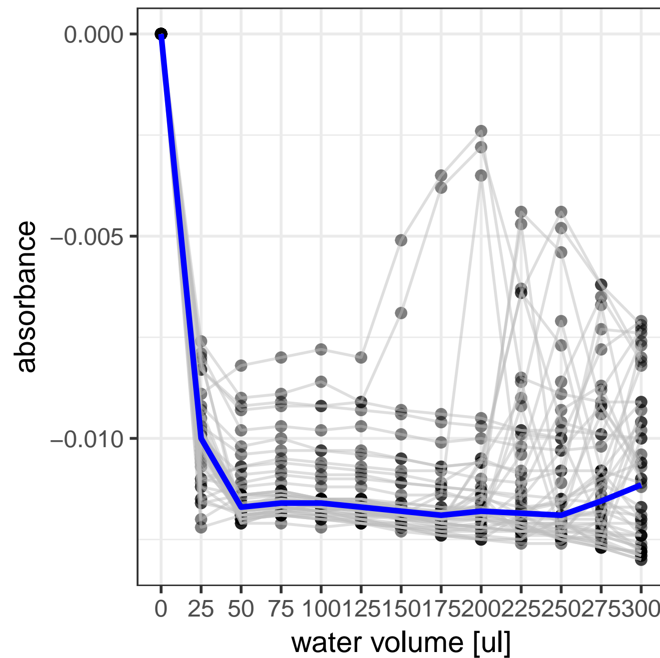


Figure 4.8: **Change in absorbance when water is added to empty wells.** The x-axis shows well volume; the y-axis shows the change in absorbance relative to the first empty well measurement; the dots show absorbance measurements; each gray line represents an individual well as water is added to it; the blue line follows the medians at each volume. Adding water to the empty wells reduced the absorbance measurement. Absorbance drops significantly after adding the first 25ul, and seems to stay about the same from 50ul on. The spread is lowest between 50ul and 125ul.

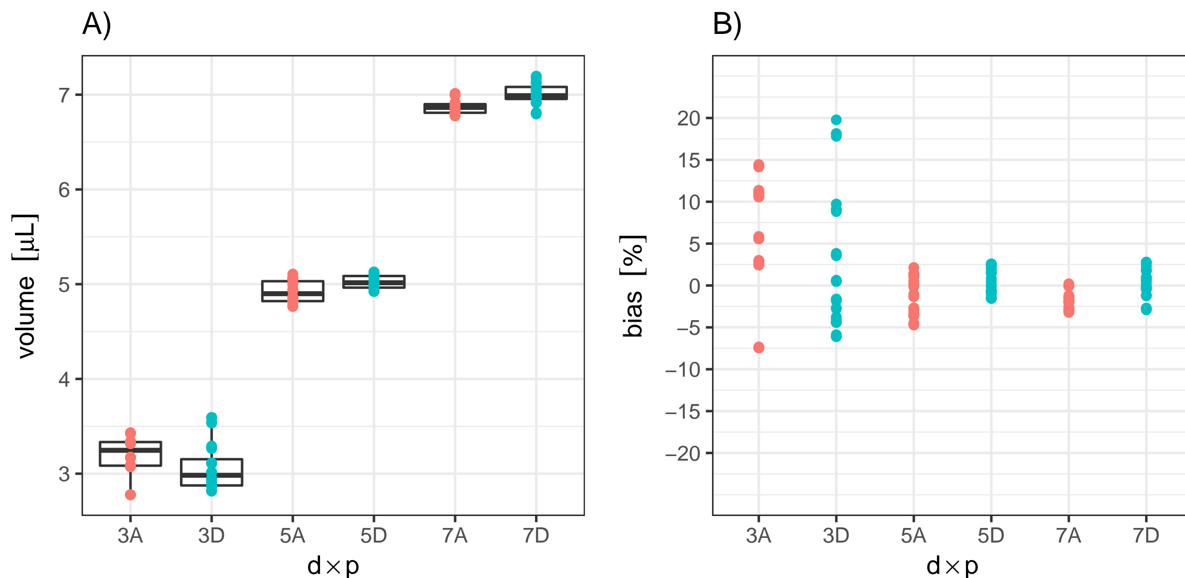


Figure 4.9: **Small-volume bias correction. A)** Graph of the volumes after correction. The x-axis is the desired dispense volume and liquid class (Air = red, Dry = green); the y-axis is the measured volume; the points are the individual measurements; box-plots are displayed behind the points. **B)** Graph of the biases after correction; The x-axis is the desired dispense volume and liquid class (Air = red, Dry = green); The y-axis is the bias %; the points are the individual measurements.

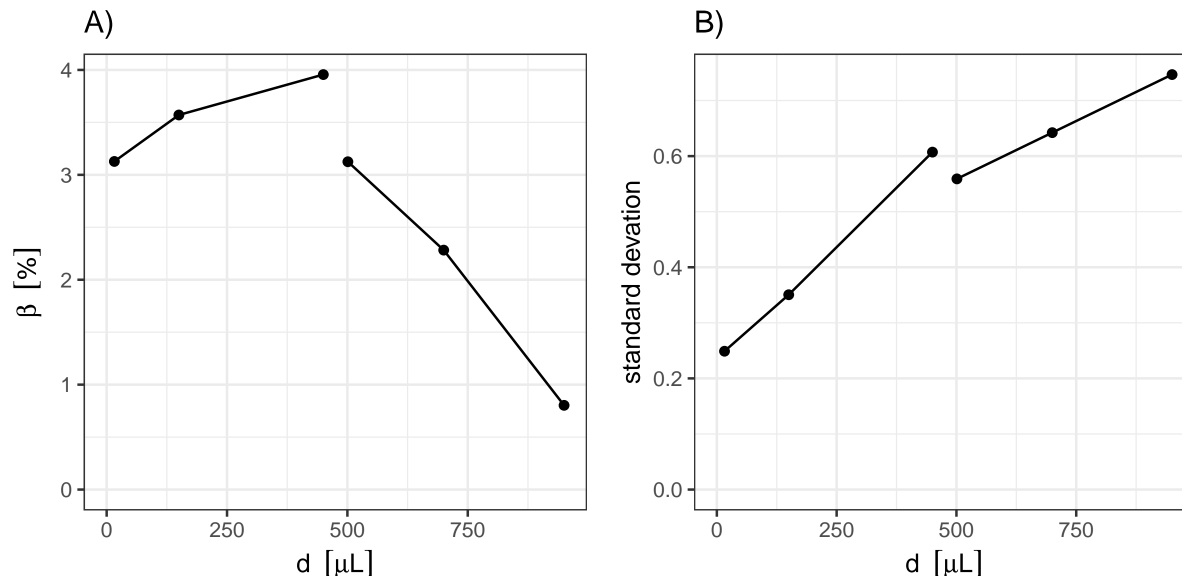


Figure 4.10: **Bias and spread of larger dispense volumes.** These graphs show the bias and spread estimates for subclasses 2 and 3 of the "Air" liquid class. The x-axis shows the dispense volume; The y-axis shows bias in Panel A and the standard error in Panel B; the dots show the estimates values; the lines connect the volumes belonging to the same subclass.

Absorbance Diagnostics. As illustrated previously in Figure 4.6C, the shape of the liquid surface changes as the well volume changes, and this impacts the percentage of liquid that is in the absorbance beam. We generally expect the surface curvature to be more concave at lower volumes, so low well volumes should results in lower absorbance measurements than higher volumes for the same amount of dye. Indeed, this is what we see in Figure 4.11. When possible, wells should all be filled to the same volume before absorbance measurements are taken, and this chart helps researchers choose that volume. By choosing a volume where the slope of the curve is fairly flat, one can minimize the measurement variance due to random differences in total well volume. Alternatively, if an experiment does not permit a uniform volume for absorbance measurement, the chart provides the correction factors — for example, according to the figure data, 50 µL well volumes have about 75% of the absorbance of 300 µL well volumes, so dividing the 50 µL measurements by 75% should allow one to compare them to the 300 µL measurements.

Z-level Diagnostics The z-level diagnostics help us determine the reliability and feasibility of z-level detection for different labware models and at different bench positions. The measurements for our 96-well reader plates are shown in Figure 4.12. Performing linear regression on the data results in an adjusted R^2 value of 0.9937, quantitatively verifying the good linear fit. Z-level detection starts around $d = 50\,\mu\text{L}$, the residual standard error is 5.8 µL, and the scale factor is 1.04 with standard error 0.0039. These results indicate that z-level measurements could be used for run-time error detection of well volumes in cases where the error is sufficiently large (e.g., $> 11\,\mu\text{L}$).

4.4 Materials and methods

4.4.1 Methods Overview

The pipetting model incorporates 1) the bias and variance of dispensing operations, 2) the unintended dilution occurring within fixed tips, 3) weight measurements, and 4) absorbance measurements. Based on the model, several protocols were designed to estimate the model parameters. Roboliq produces a script to run on the

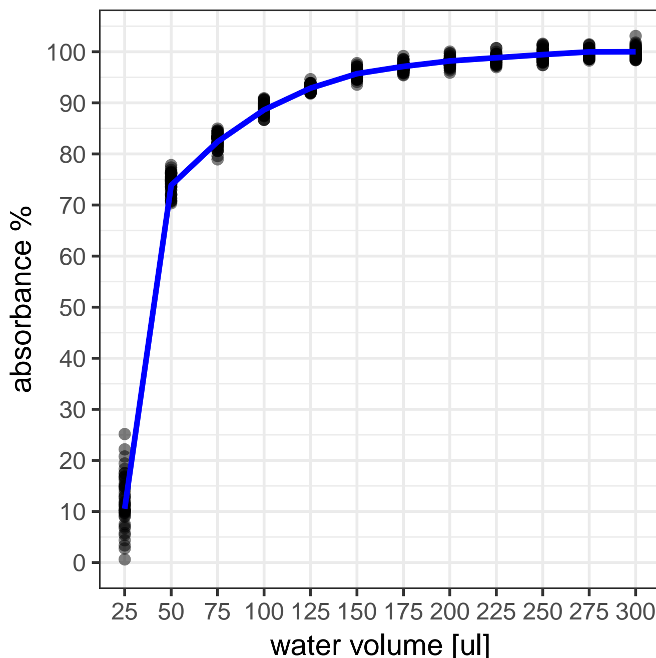


Figure 4.11: **Change in absorbance when water is added to** 16 µL **dye aliquot.** The y-axis is the percentage of absorbance relative to the average maximum. The x-axis is total well volume, whereby water is added to an initial dispense of 16 µL dye. The blue line represents the average absorbance changes, and the black dots show the normalized measurements. About 75% of full absorbance is reached for a well volume of 50ul, and 95% at 150 µL.

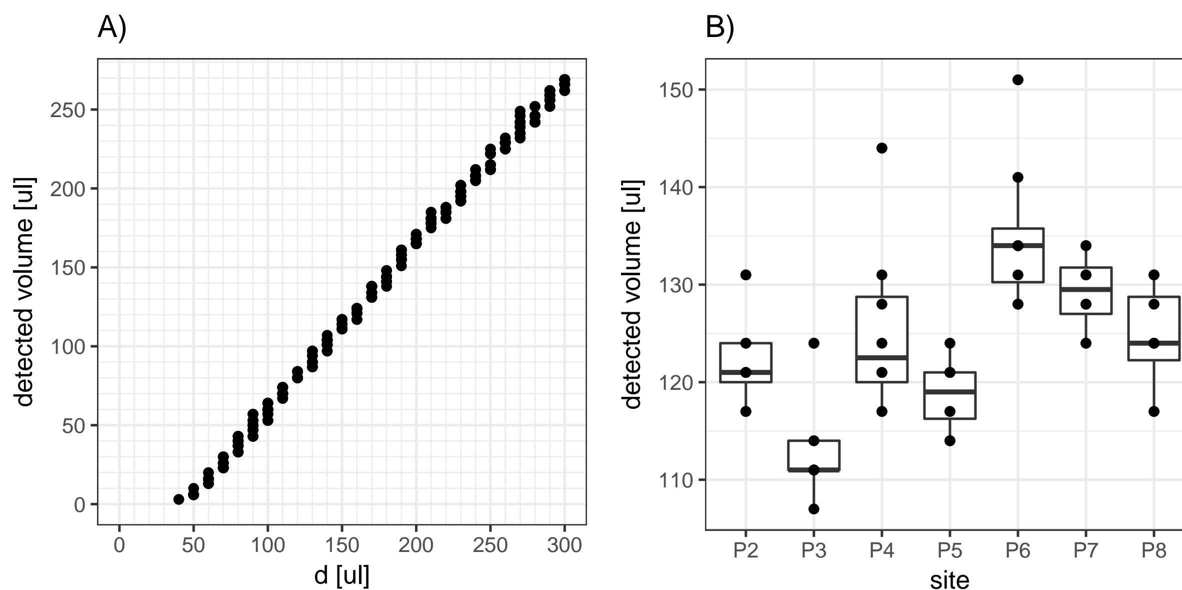


Figure 4.12: **Z-level measurements.** A) This graph shows the z-level measurements on a site named `P3`. The x-axis is the dispense volume and the y-axis is the volume Evoware estimates with z-level detection. The circles represent individual measurements. B) This graph shows the z-level measurements at different sites with a constant dispense volume of 150 µL. This data lets us estimate the relative offset of the z-level measurements between bench positions.

Tecan Evoware robot, and it aggregates the measurements for analysis (see Section 2.2.6 about designing experiments and tidy data). Finally, the analysis procedure is run in R and exports the parameter estimates to be used in future experiments and simulations.

For the correction experiment in Section 4.3.3 we passed the bias parameters to Roboliq to automatically correct the dispense bias by adjusting the volumes up or down accordingly.

Supplementary diagnostic procedures also evaluate meniscus effects (how the shape of the liquid surface impacts absorbance readouts) and z-level detection (detecting when the pipetting tips contact liquid). Appendix G additionally describes evaporation diagnostics.

An integrated absorbance reader is used in most of the quality control protocols to estimate the bias and variance of dispense volumes, as well as to determine the meniscus effects and evaporation rates. A precision balance is used for weight measurements in two protocols, and z-level measurements let us calculate the bias and variance of z-level detection.

The experiments were run on our Tecan Freedom 200 robot (Appendix A) using a pipetting arm with four fixed teflon coated tips for pipetting 3 − 1000 µL. As liquid sources, the robot had a 100 mL trough of filtered water and various dilutions of OrgangeG dye. The stock source of OrangeG had a concentration of $8\,\mathrm{g\,L^{-1}}$ — absorbance measurements of ≈ 1 were obtained by 13.7x and 137x dilutions resulted in 15 µL and 150 µL respectively. Absorbance readouts were performed on a Tecan M200 reader at a wavelength of 480 nm using 96-well transparent plates (Nunc Nunclon F). Weight measurements were performed on a Mettler Toledo precision balance that measures in 1 g steps. Plate shaking was performed on a Variomag Teleshake single-plate shaker or in the Tecan M200 reader to ensure proper mixing of dye mixtures. On the software side, the analysis was run with R v3.4, RStudio v1.0, and RStan v2.15 for MCMC simulations.

4.4.2 Protocol 1: characterization of "standard" pipetting configuration

We chose a specific volume and liquid class to use as a standard reference: 150 µL using our *air* dispense liquid class. Air dispense was chosen because it can be used regardless of whether the well already holds liquid (in contrast to dry and wet dispense). The standard volume was chosen so that 2x dilutions can be easily performed to determine the unintended dilution parameters − 150 µL was chosen because the wells of the reader plate can hold slightly more than 300 µL. If we had used 384-well plates instead, we would have had to choose a smaller reference volume.

This protocol utilizes both the absorbance reader and the balance to determine the parameter for the 150 µL air dispenses volume $\mathbf{VD}_{A,150}$ and unintended dilution $\mathbf{VU}_{A,150}$ (Table 4.1). The protocol steps are:

1. read empty plate
2. weigh empty plate
3. dispense 150 µL twice into each dye well (4 tips, 6 replicates, so 24 wells)
    - to minimize total evaporation, first dispense one layer in all wells, then the second layer
4. weigh plate
5. read filled wells
6. for each dye well, transfer 150 µL to an empty well
7. fill 24 empty wells with 300 µL water
8. fill all dye wells to 300 µL with water
9. read dye and water wells

After measuring the empty plate, the robot dispenses 2x 150 µL into a set of wells (Group A); It then transfers 150 µL from those wells to a new set of wells (Group B) and fills all wells with another 150 µL water; finally, the readouts are performed. If unintended dilution is taking place, we expect Group B to have lower absorbance values, since they will have experienced one more dilution than the Group A wells. The changes in absorbance from Group A to B are graphed in Figure 4.7B, where we see an approximately 1% dilution.

Due to the complex interactions that arise from using two kinds of measurements and creating a dilution series, this protocol cannot be properly analyzed with linear methods. Instead, we used Roboliq to generate

MCMC models in Stan format and performed the simulations through the RStan interface [122].

The electronic supplement has a folder named `qc41-150ul` for this experiment that contains the Roboliq protocol, the Stan model, the RStan analysis script, and the measurements.

4.4.3 Protocol 2: General test of dispense volumes ($\leq 150\,\mu L$) and variances

This protocol uses absorbance measurements to find the dispense bias and variance relative to the reference volume of 150 µL (Table 4.1 and Figure 4.7). It also lets us characterize the absorbance reader with regard to measurement error, the absorbance of the empty labware, and the variance of absorbance between wells (Table 4.2 and Figure 4.8). For volumes 3 to 16 µL, we fixed $\mathbf{VU} = 0$, because the unintended dilution of such small volumes is not practically identifiable with our total well volume of 300 µL.

The protocol uses the same transparent 96-well plates as the first experiment. Due to its well capacity, we are restricted to dispense volumes less than 300 µL. As discussed above, we want to estimate the parameters for each liquid subclass separately. If we assume that each subclass exhibits linear behavior over its volume range (e.g. true volume = desired volume * scale + offset), then the optimal choices of test volumes would be the extreme ends of the range. For well-behaved liquid classes, this is a safe choice, but to verify the model, we also include a central volume.

The smallest liquid subclass spans the range from 3 µL to 15 µL, so we selected the extremes 3 and 15 µL and a central volume of 7ul. The second subclass spans the range from >15 to 500 µL. On the lower end, we selected 16 µL, and in the middle, we selected 150 µL because it is the reference volume.

Since measurements will be performed by the absorbance reader, we need to ensure that our dye concentrations remain within the linear range of the reader, and our supplementary protocols found that the linear range extends to absorbance values around 3.6 (Figure 4.13) If we chose a single dye such that 150 µL had a readout of 2.6, then the 3 µL dispenses would be expected to have a readout of 0.05. But since the measurements have more uncertainty when they are that low, we instead selected two source concentrations: one such that 150 µL has a readout around 1, and one such that 15 µL has a readout around 1. We label them dye0150 and dye0015, respectively. The dispense volume 150 µL uses dye0150, whereas the dispense volumes ≤ 16ul use dye0015 (with one exception described below).

The protocol requires one plate per liquid class, though the liquid classes are distributed among plates for better randomization. We test three liquid classes: air dispense (dispense from above the well), wet dispense (dispense just below the well's liquid surface), and dry dispense (dispense at the bottom of an empty well). For each liquid class, we test the 5 chosen volumes with 4 replicates for each of our 4 tips, requiring 80 wells in total. Of the remaining 16 wells, 8 are filled with water only, and 8 are used to duplicate the 15 µL dispenses with dye0150. The duplicate dispenses of 15 µL with both dye0015 and dye0150 allow us to estimate the true concentration ratio between them. This is necessary to infer the true volumes dispensed relative to the standard 150 µL dispenses. To optimize the randomization, the volume and liquid class factors are distributed across plates as evenly as possible.

Wet dispenses require liquid to already be present in a well, so we dispense 60 µL water prior to wet dispenses. Later the well is filled to 300 µL using more water. Originally we only had two steps for such wells: 1) pre-fill the well with water (300 µL - dye volume), 2) dispense the dye aliquot. However, for small volumes, we found that the replicate absorbance measurements were exhibiting very high variances, even after 10 minutes of shaking. Presumably, the dye was not well-mixed. By mixing in a large volume of water after the dye dispense, variance had settled to a minimum after about 7 minutes of shaking. Therefore, after dispensing all the dye wells, we shake the plate for 7 minutes before measuring absorbance.

The protocol follows these steps:

- For each plate:
- Measure absorbance of empty plate
- Dispense 60 µL water first for wet dispenses
- Dispense specified amounts of dyes to all dye wells

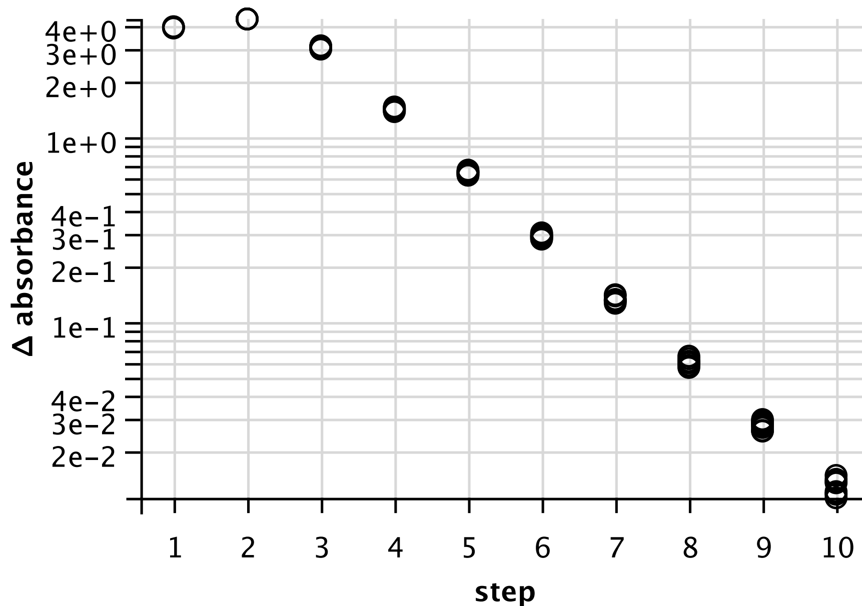


Figure 4.13: **Operating range of absorbance reader.** The data shown here come from a dilution series, where each step diluted the wells 2:1. The x-axis shows the dilution step; the y-axis shows the absorbance relative to the empty-well absorbance on a log scale; the circles show the data points for 8 replicates. In first two steps, the concentration of the dye saturates the reader; starting with the third step, however, the concentration appears to be within the linear range of the reader with absorbance around 3.6. The absorbance values at step 10 are around 0.01, and it is interesting to observe that the linearity has not broken down yet, but the relative variance has increased significantly.

- Fill all wells to 300 µL with water
- Shake for 7 minutes to ensure mixing of small dye volumes
- Measure absorbance of filled plate:
  - Shake for 30s in the reader and read twice
  - Again shake for 30s in the reader and read twice

For analysis, the model provides estimates of the dispense volumes and their variances, and the parameters are estimated using MCMC. The execution time of the MCMC simulation was 6.5 minutes for 4 chains of 4000 iterations.

The electronic supplement has a folder named `qc42-general` for this experiment that contains the Roboliq protocol, the Stan model, the RStan analysis script, and the measurements.

4.4.4 Correction and Validation Protocol

Using the previously discovered parameters, we aimed to correct the dispense biases. The protocol tests volumes 3, 5, and 7 µL relative to the standard volume of 150 µL (Figure 4.9). Note that we use one volume that was not used in previous tests, 5 µL.

This protocol uses one transparent 96-well plate, one 1.5ml eppendorf tube, and two troughs. One trough is filled with dye0015 and the other with water. A 10x dilution of dye0015 is made in the tube via 150 µL dispenses (1x dye0015, 9x water), which is then used to make eight standard wells with 150 µL air dispenses. The protocol steps are:

- Measure absorbance of empty plate
- Prepare calibration wells:
- Dispense 150 µL dye0015 to a tube
- Dispense 9x 150 µL water to the tube
- Shake the tube
- Transfer 150 µL to 8 wells
- Assign some wells as water wells
- Dispense 60 µL to all test wells
- Dispense corrected dispense volumes of dye0015 to all test wells
- Fill all wells to 300 µL
- Measure absorbance of filled plate:
  - Shake for 30 s in the reader and read twice
  - Again shake for 30 s in the reader and read twice

The distribution of wells was as follows:

- 8 wells for calibration
- 3 volumes $\times$ 4 tips $\times$ 3 liquidClasses $\times$ 2 replicates = 72 wells
- 16 wells for water
- total of 96 wells

The analysis of this experiment was simpler, so it was possible to use the linear mixed effect method and scaling for the graphs. The electronic supplement has a folder named `qc43-correction` for this experiment that contains the Roboliq protocol, the R analysis report, and the measurements.

4.4.5 Supplementary and Diagnostic Protocols

Dispenses in eppendorf tubes to estimate bias and variance of larger volumes. For these estimates, we rely solely on weight measurements to find the bias of the larger volumes 450, 501, 750, and 950 µL. Due to the larger volumes, we use eppendorf tubes instead of reader plates. By filling eppendorf tubes and using replicates, the data provides both bias estimates and rough variance estimates (Figure 4.10).

This experiment uses 1.5 mL eppendorf tubes, water, the balance, and a single syringe on the pipetting arm. Our balance is not integrated with the robot, so each weighing requires manual involvement. Our tube holder can hold up to 20 eppendorf tubes at once. Due to the manual interactions, I wanted to minimize the number of measurements in this protocol, and restricted the tests to using a single syringe on the pipetting arm (syringe 1), and only did 2 replicates of each volume. The tests were run for the volumes 16, 150, 450, 501, 750, and 950 µL. The protocol steps are:

1. Weigh empty tubes twice
2. For each volume and replicate:
    1. Dispense volume one or more times to fill the tube
    2. Weigh tube twice

The analysis uses the linear mixed-effects method and traditional statistics to estimate the bias and variance. The electronic supplement has a folder named `qc01-accuracy-tubes` for this experiment that contains the Roboliq protocol, the R analysis report, and the measurements.

Absorbance tests to find the impact of volume and meniscus on measurements. Here we want to measure the impact of well volume on the absorbance readout. We dispense a small dye aliquot to half of the wells on a plate and then measure the change in absorbance as both the dye and non-dye wells are gradually filled with water (Figures 4.8 and 4.11). We found that adding water to an empty well decreased the absorbance value by about 0.01 units on our 96-well reader plates. Although we did not investigate the physical cause of the decrease, we made use of this knowledge to improve the accuracy of our parameter estimates in all other experiments.

This protocol uses a single transparent 96-well reader plate, one trough for dye0015, the system water source, and the absorbance reader. The protocol steps are:

1. Measure absorbance of empty plate
2. Dispense 16 µL dye0015 to half of wells (randomized)
3. For d in 25 to 300 µL by 25 µL steps:
    1. Fill wells with water to volume d
    2. Measure absorbance of plate

The analysis is performed using linear statistical methods to create the chart and table showing the impact of well volumes on absorbance measurements. The electronic supplement has folders named `qc02-absorbance-B` for this experiment that contains the Roboliq protocol, the R report, and the measurements.

Z-level tests to calibrate Evoware's z-level measurements with true volumes. This experiment uses one 96-well reader plate (but any kind of plate can be used), a trough for water, and the z-level detection capabilities of the pipetting arm.

Evoware's z-level measurements are automatically performed before all aspirations and wet dispenses. The measurements can also be explicitly performed. We found a systematic difference between the explicit and automatic measurements, however, so we ended up only using the explicit measurements for this protocol.

The protocol should be run for each labware model of interest. For the flat-bottom square wells on our reader plates, the relationship between true volume and z-level measurements is highly linear (Figure 4.12A). However, for irregular shapes (such as in PCR wells), the relationship is no longer nicely linear due to Evoware's sub-optimal parameterization of the well shape.

There is also a lower bound on the volume that can be reliably detected by z-level measurements due to difficulties in properly specifying the true altitude of the plate bottom, and because low water volumes within wells tend to stick to the sides of the well where the tip cannot detect them. Different locations on the bench may vary in height, which causes further offsets in the detected levels (Figure 4.12B).

The protocol has two parts with these steps:

- Part 1: test different volumes

  - dispense water volumes ranging from 0-300 µL in steps of 10 µL; 2 replicates for each volume, so 62 wells. (wells and volumes should be randomly distributed)
  - measure each well with each tip twice (randomized order)

- Part 2: test different bench positions
  - for each tip, dispense 150 µL into one well
  - using the same tip/well pairs, measure z-level twice
  - move the plate to other bench positions and repeat the measurements

The analysis is performed using the linear mixed-effects method. The electronic supplement has a folder named `qc05-zlevel` for this experiment that contains the Roboliq protocol, the R analysis script, and the measurements.

Well-mixedness Diagnostics.

It is important to ensure that our dye aliquots are well-mixed because that minimizes the variance of the absorbance readouts. We found that if the small dye aliquot is dispensed first and then a large volume water is mixed in, then the well will be well-mixed. But this is not possible for wet dispenses, since water needs to already be in the well. An obvious solution would be to use the pipetter to mix the wells by aspirating and dispensing several times, but this would complicate analysis because we found such mixing to affect well volume and concentration. Instead, our solution was to dispense a small-to-medium volume of water first (e.g., 60 µL), then the small dye aliquot, then fill the well with a large water aliquot to 300 µL.

This script helps determine how long you need to shake to ensure well-mixed wells. It tests four different "prefill" volumes, meaning how much water is dispensed before the dye aliquot. "Maximum prefill" refers to the case where the well is first filled with $300 - d$ µL for a dye volume d. It then graphs the time course for the user to choose a sufficient shaking duration, as shown in Figure 4.14 for several dye volumes: The 150 µL dye dispenses are mostly unaffected by the pre-fill volume, but there is a significant impact on the small dye volumes that demonstrates the problem of poor mixing of small dye aliquots when added to a mostly full well. Even after 10 minutes of shaking, the spread is still very large for 7 µL in the maximum prefill condition.

We decided that 60 µL works reasonably well for low-volume Wet dispense operations. To determine the required shaking duration, the variance of 0 µL pre-fill wells (which are well mixed) can be compared to the variance of the 60 µL pre-fill wells — once they are basically indistinguishable, then we've done enough shaking. The most critical case here is the 3 µL dye volume, because it will have the smallest absolute absorbance values. In the 60 µL condition, its spread seems to settle to about 0.001 absorbance units within 5 minutes of shaking, which is sufficient for our analyses to obtain accurate parameter estimates.

This protocol uses a single 96-well reader plate, a trough of dye0015, a trough of dye0150, the system water source, and the absorbance reader. The protocol steps are:

1. Assign some control well to get 300 µL of dye0150
2. Assign some water-only wells
3. Assign various volumes D (3, 7, 15, 16, 150) to wells; some wells should be selected for "water-first".
4. Dispense 60 µL water to the "water-first" wells
5. Dispense D of the appropriate dye
6. Fill all wells to 300 µL with water
7. Loop N times:
   1. measure plate in absorbance reader twice (with no shaking in reader)
   2. shake on shaker for T duration

The analysis method is simply graphical - it is intended to help the user decide how long to shake before measuring absorbance. For example, we used the graph to choose a shaking time of 7 min for 3 µL dispenses. The electronic supplement has a folder named `qc51-shakeDuration` for this experiment that contains the Roboliq protocol, the R analysis script, and the measurements.

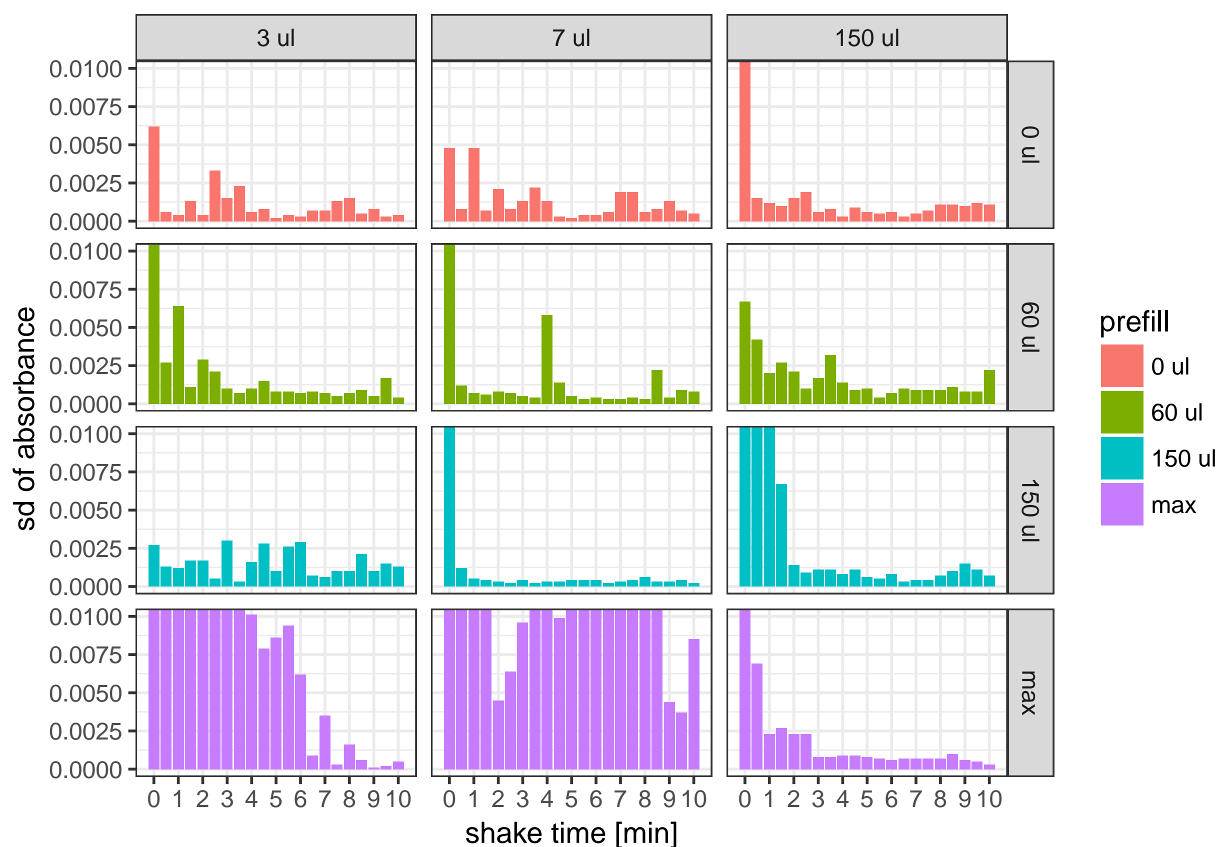


Figure 4.14: **Spread of absorbance measurements vs shaking time.** These graphs plot the variance of absorbance measurements as a function of the shaking duration after mixing dye and water. The facet columns represent the volume of the dye aliquots (3 µL, 7 µL and 150 µL); The facet rows represent the amount of "prefill" water that was dispensed in the well before the dye aliquot (they are also colored to help provide more visual distinction among the facets). The x-axis shows shaking time from 0 to 10 minutes; The y-axis shows the standard deviation of duplicate absorbance measurements, and it is capped at 0.01 absorbance units to zoom in on the smaller spreads.

4.5 Conclusion

In this chapter, we described quality control protocols for Tecan robots that we automated with Roboliq. The procedures provide practical knowledge about the system that would otherwise be difficult to obtain with much accuracy, including pipetting bias and spread, undesired dilution volumes, and measurement error, along with additional diagnostics. The proposed mathematical model is amenable to MCMC analysis, providing both parameter estimates and their standard errors. This information can be used for maintenance purposes, determining the limitations of a robot, troubleshooting experimental errors, detecting run-time errors, and correcting pipetting bias.

Future work could involve extending or generalizing the protocols. The full set of protocols was only run on one Tecan, and certain adaptations would be required to generalize the protocols to other robots (for example, one of the experiments uses tubes, but not all robots have tube holders). Ideally, robots should be optimized for every liquid they handle [123] — in this paper, we only consider water and dye mixtures that aspirate and dispense like water, so an extension to other liquids would be interesting. Developing an experiment to efficiently determine the undesired dilution of a range of volumes would be valuable — we did not pursue this after ascertaining that the undesired dilution at 150 µL is only about 1 µL, but it may be worse for other volumes. The software would also benefit from a graphical user interface. Currently, the software is controlled from the command line and RStudio, and entry of weight values is inconvenient because the balance is not connected to the robot. A colleague created a simple user interface to Roboliq for yeast transformation experiments, and a similar effort would be beneficial for controlling and coordinating the quality control experiments as well.

Chapter 5

Conclusion

5.1 Conclusions

Overall, we can evaluate Roboliq in terms of the aims discussed in Section 1.2, repeated here for convenience:

- Execution: we aim to simplify the programming of reproducible experiments.
- Sharing: we aim to provide a portable way to specify protocols that can be shared between labs.
- Design → Execution: we aim to import let users build their protocols upon design specifications.
- Execution → Analysis: we aim to output measured data that includes all design factors to facilitate analysis.
- Execution → Design: we aim to quantify the robot's pipetting performance and let this knowledge inform design decisions.
- Analysis → Design: we aim to simulate execution measurements so that analysis can validate the design's statistical power prior to execution.

Execution. On the one hand, Roboliq makes programming more difficult rather than easier, because it is an additional piece of software to learn and it requires some familiarity with the command line. Both of these mental burdens can be addressed by creating application-specific GUIs, which we have successfully done but do not discuss here because the applications were not within the scope of the thesis. In our view, one of the best uses of Roboliq is embedding it in such application-specific software.

On the other hand, Roboliq makes it much easier to create reproducible experiments as defined in Section 1.2.1, because it supports data tables and outputs "tidy" measurement data. Its use of AI also simplifies a lot of commands and allows most details to be omitted relative to Tecan Evoware scripts. And indeed, most of the errors that can easily occur while writing Tecan scripts are impossible by design in Roboliq, useless you use low-level commands directly. So even without GUIs, Roboliq scripts are faster to write, easier to maintain, and safer to execute than the equivalent Tecan scripts.

Sharing. My initial goals for portability were naively ambitious, having not yet understood how drastically different certain experimental steps need to be performed on different robots. What we actually achieved was more modest but still valuable: to be portable between robots with reasonably similar setups, and to ease adaptation in cases of divergent setups.

To this end, we developed data structure to represent protocols and robot configurations. Its special property is that it can be arbitrarily split between multiple files, and then merged together by a simple algorithm (see Section 2.2.8). This allows for robot configurations to be in one file, the portable aspects of a protocol in a second file, and lab-specific adaptations in a third file.

The Automated Planning approach allows Roboliq to discover many of the low-level execution details on its own, so that they can be omitted from the protocol definition, thereby making the protocols more portable.

Design → Execution and Execution → Analysis. These link is made possible by data tables (Section 2.2.6). You can properly design the experiments by hand or in a program such as R, and by formulated the design as a table (which is standard practice anyway), Roboliq can use tables to guide the execution with its `data` property, functions, and substitutions. In the measurement commands, the design factors in the active data table are automatically associated with the wells being measured, letting Roboliq save measurements together with metadata so that the user has a "tidy" dataset to analyze immediately.

Execution → Design and Analysis → Design. These are the most abstract links in the design-execution-analysis stages (Figure 1.1, Sections 1.2 and 2.2.6), and the possibility of using Roboliq to tightly couple the stages is its most valuable quality for scientific research, in my opinion. The feature relies on both data tables and measurement simulation. By following the workflow described in Section 2.2.7, it is possible to refine the design, execution, and analysis before any actual experimental run. This has spared us countless experiments that would have failed, had we simply followed our intuition about what would work.

Laboratory automation systems, especially decentralized ones without high standardization, suffer from numerous practical limitations compared to automation solutions in other engineering domains [25]. In addition to general limitations on portability, ease-of-use, and flexibility, problems include software crashes, communication errors between devices, unpredictable failures in liquid handling, labware misplacement during transportation, and predominantly open-loop operation (for example, a robot can continue through the motions of pipetting even though a reagent is missing). We argue that Roboliq will contribute to reducing such limitations because it enables (i) platform-independent high-level programming with complex constructs; (ii) predominantly automatic generation of lower-level commands with information about robot configuration, protocols, and reagents, and (iii) automatic generation of scripts for execution on the robot and its associated devices. Our proof-of-principle applications in Chapter 3 provided practical evidence, and the quality control protocols in Chapter 4 can help ensure reproducibility. Furthermore, Roboliq's data tables in Chapter 2 help users integrate the software into a scientific workflow where design, execution, and analysis are tightly linked.

5.2 Future Work

In terms of future extensions of Roboliq, the modular software architecture makes it possible to add currently missing components such as compilers for other types of robots. The central planner currently reasons using first-order logic, which could be extended to a high-order logic to allow for more expressive power. Also, the constraint-based planning only uses set-based constraints (for example, a bench site can be one of the configured bench sites), but one could include quantitative and programmed constraints. The software should also be integrated with other related systems. For example, it could be integrated with systems that try to create reproducible protocols (e.g., Biocoder [34] or Antha [35]), reproducible laboratory workflows (e.g., Galaxy [124]), or eventually accept protocol ontologies such as OBI [125] and EXACT [33, 56]. We also argue that Roboliq will provide the technical basis for establishing the advanced quality control mechanisms that are urgently needed in the field [29]. Fault detection and error correction could be enabled by protocols that automatically integrate controls for liquid handling (e.g., control wells with easily quantifiable colored solutions) and by estimating process states using the robot's sensor information. Such developments critically rely on having an 'intelligent' AI-based system that generalizes to different protocol and device types, visual programming, or integrated data analysis.

Appendix A

Hardware Setup in this Thesis

Labware. Microwell plates were first created in the early 1950's [126] in a 8 rows × 12 columns configuration of 96 wells. The distance between well centers was standardized at 9 mm for 96-well plates. Configures with 6, 12, 24, 384, 1536, and 3456 wells are also common nowadays. Further variations are possible on well shapes (square or round), well-bottom shapes (flat, conical, spherical), coloration (black, white, transparent, translucent), material, and other special features. Although the plate geometries tend to be similar (e.g. base area of about 86 × 128 mm^2), the differences are still big enough that, for each model, the technician may need to configure the following characteristics on the robot:

- Pipetter: aligning the tips to the middle of the well; finding the altitude of the empty wells; configuring the maximum volume of the well.
- Transporter: configuring the proper grip; ensuring proper placement on the holders.
- Equipment: configuring any other equipment that needs to know the labware and well dimensions (such as sealers and fluorescence readers).
- Software: indicating which labware can be used at which bench site and in which equipment.

It is therefore advisable to minimize the assortment of models used; this will tend to contribute to the robustness of experiments by allowing the technician to better focus on maintaining and optimizing fewer labware configurations over time. For the experiments described in this thesis, we employed seven main labware models (Figure A.1).

Robotic Arms. The robotic arms on the Tecan are used for moving plates around the workspace. They have 5 degrees of freedom (Figure A.2). When moving a plate, the arm starts at the top-back of the bench (maximum y and z coordinates); the arm then moves along the x axis to the x coordinate of the plate; then it moves to the appropriate y coordinate, adjust the gripper distance d if neccesary, lowers itself along the z axis to a pre-programed height for the given labware model and bench position, and closes the gripper distance d in order to grab the plate. It then reverses the z and y movements before moving the new bench position, lowering the plate, and releasing the grippers. This process tends to be fairly reliable, but its infrequent failures can still be very problematic, resulting in plate spills and other execution failures. This can be minimized by proper maintenance and performing quality control of the arm movements prior to running the real experiment.

Equipment and Integration. Integration of robots and equipment is an extremely important issue for lab automation, and the standardization of interfaces is being pursued by SiLA [127]. This effort should help increase the modularity and compatibility of automation hardware and software, while simultaneously allowing greater choice and bringing down the costs due to greater competition. For companies which benefit from lockin, the incentive to pursue integration with other manufacturers is low. Unforturnately, this applies to our robots as well, and there are significant hurdles to using Tecan robots with equipment that does not come from Tecan itself.

Measurement Equipment. In this thesis, we are be concerned with measuring absorbance, weight, and

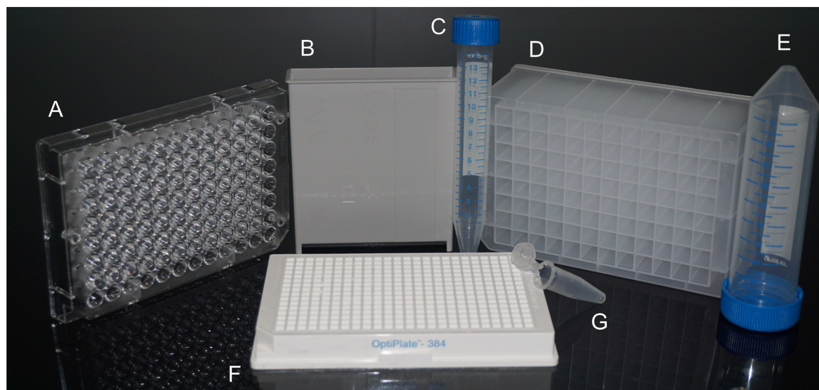


Figure A.1: **Labware frequently used in this thesis.** (A) Nunc Nunclone F 96 square-well transparent flat bottom plate used for absorbance measurements. Each well holds slightly more than 300 µL. (B) Trough holding 100 mL. (C) Falcon tube, holding 15 mL. (D) Deep well plate with 96 wells. Each well holds 2500 µL. (E) Falcon tube, holding 50 mL. (F) OptiPlate White 384 square-well flat bottom plate. Each well holds slightly more than 75 µL. (G) Eppendorf tubes, holding 1.5 mL.

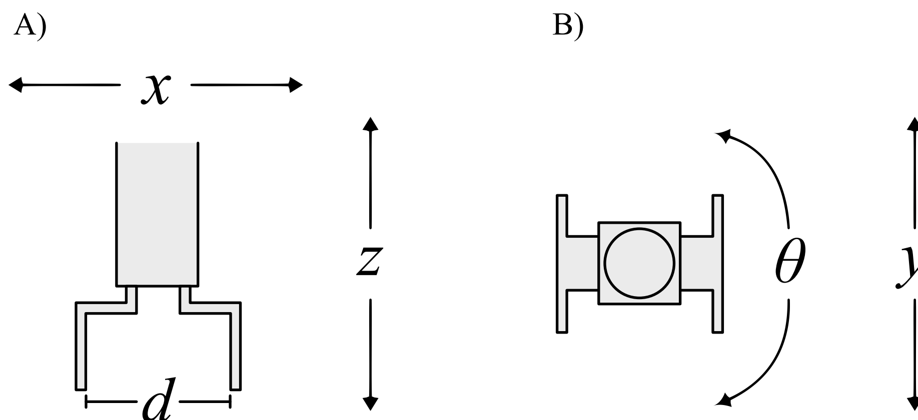


Figure A.2: **Tecan robot arm, degrees of freedom.** Tecan's robotic arms have 5 degrees of freedom in their movements – x: left/right, y: forward/backware, z: up/down, θ: rotation about the z-axis, and d: distance between the two grippers. A) Front view of an arm and gripper. B) Top view of an arm and gripper.

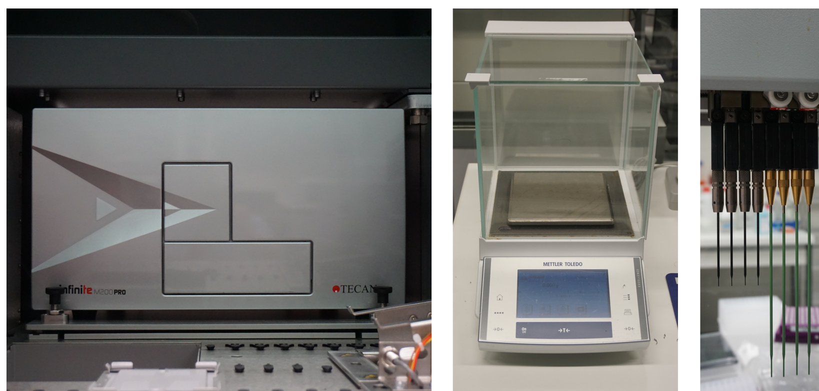


Figure A.3: **Measurement equipment used in this thesis.** (A) Tecan Infinite M200 Pro optical reader (B) Mettler Toledo precision balance (C) Tecan pipetter tips with z-level detection

liquid level (Figure A.3).

Dangers. These robots can be quite dangerous to humans, so it is necessary to keep the robot's workarea locked during operation to avoid human proximity to the moving parts. This causes some inconvenience, so many operators disable the lock. We were warned during training, however, that the pipetter have been known to pierce all the way through a human hand. Furthermore, the robot ignores its emergency stop button until *after* the current action has completed.

Preventing evaporation. For long-running experiments, it was sometimes necessary to minimize evaporation. Sometimes it is sufficient to simply have the robot place a lid on the plate after pipetting operations. Otherwise we used the integrated sealers to apply an adhesive seal to the plate. The advantage of lids vs seals is that they are more easily removed and do not require extra sealing equipment. The advantage of seals is that they are more effective and can prevent inter-well contamination in the case of vigorous shaking, and they do not require that plates be purchased with lids. We used two different sealing materials: one is air-tight, and one is breathable for cultivating yeast cultures. The sealing materials cannot be easily swapped, however, so each of our sealers was permanently assigned a specific material; on the main robot described here, we used the air-tight material.

If you need to pipette from a plate after placing a lid on it, the lid can simply be removed. If instead the plate had been sealed, then there are two options: peel the seal off the entire plate, or pierce the seal. Peeling off the seal may not be an option if some wells need to stay seeled, and indeed we avoided peeling whenever possible, because the peeling equipment occasional failed to properly remove the seal, leading to unpredictable errors. Such failures appeared to be more likely when the seal had been on the plate for a long time. If peeling was necessary, we could alternatively program the robot to pause and ask the user to manually remove the seal.

Piercing the seal is the other option, with its own set of tradeoffs. First of all, we found that piercing gradually damaged the teflon coating on our tips, so we could also use non-coated steel tips. Secondly, piercing also left adhesive residue on the tips which had to be manually cleaned. Thirdly, the plate needs to be held down, lest it stick to the tip when the pipetter is raised; this requires a special "downholder" construction on the bench (Figure A.4). Lastly, Tecan does not provide a reliable downholder for deep-well plates, so the plates may occasionally pop out of place when the pipetter is retracted.

Handling condensation on sealed plates. Our robot offers two methods to remove condensation from the seal of a sealed plate: brief centrifugation and top-side heating in the thermocycler. Centrifugation is only viable if it do not damage the well contents (e.g. cells) or cause the liquid components to separate. Using our thermocycler is only possible with PCR labware. If neither method could be applied (such as when growing cell cultures in a deep-well plate), we had to forego the removal and instead extract an aliquot to another plate for measurements.

Temperature control. Due to the temperature-dependence of biochemical reactions, it is sometimes

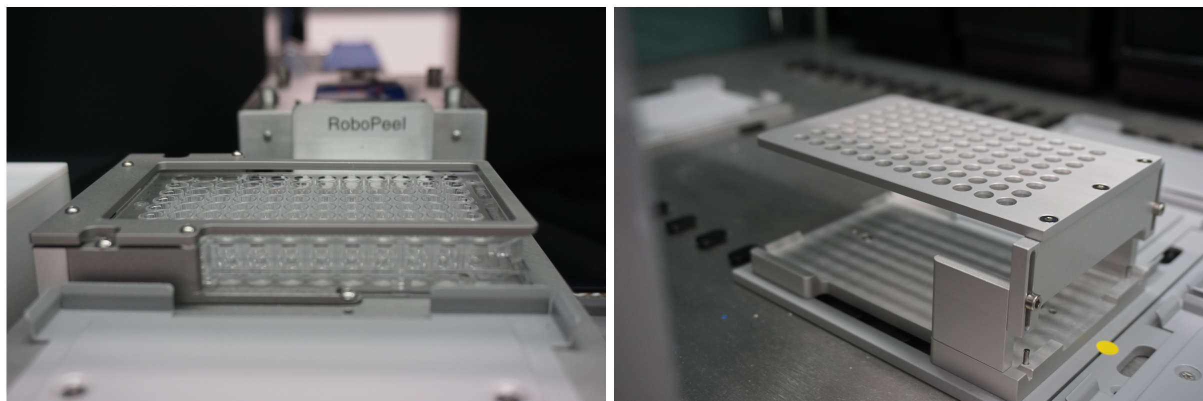


Figure A.4: **Downholders.** These are two of the downholders we have for holding plates down when pipetting from sealed plates. The downholder on the left is for microwell plates, and the the one on the right is for deep well plates.

necessary to cool or heat labware. To suppress reactions, we have a coolant system on the main robot to cool tubes and PCR plates. The coolant is not software controlled, however, so it is necessary to turn it on manually prior to experiments that require cooling.

Our robots have five possibilities for heating: the termocyclers, the centrifuge, the optical readers, the dedicated incubators, and a powerful shaker for growing yeast colonies. The experiments described in this thesis only use the heating capabilities of the centrifuge and the optical readers. For example, when monitoring long-term changes using fluorescent measurements, we incubated a plate in the reader at 25 °C between readouts.

Other techniques. Besides the techniques used in this thesis, many others are available for automation using robots from Tecan and other manifacturers. Other techniques our robots perform include vacuum filtering, separation by centrifuging a filtered plate on top of waste plate, removal of liquid contents by plate flipping, PCR reactions, cultivation of cultures in temperature controlled aggitators, and colony picking. Most of these have been automated with our Roboliq software, but they do not fit within the context of this thesis.

Appendix B

Robot configuration

In order for Roboliq to compile your protocol for your robot setup, it needs to know how the robot is configured. This is the most complicated part of Roboliq – it requires advanced technical knowledge or your robot and some programming skill. Once the configuration has been specified, future users do not need to be concerned with it.

So if someone else has configured Roboliq for your robot already, then you should probably just proceed to the next chapter. However, if you need to write the configuration, this chapter is for you.

B.1 Evoware configuration

(If you are not using an Evoware robot, you can skip this section.)

Roboliq supplies a simplified method for configuring Evoware robots. You will need to create a JavaScript file in which you define an `EvowareConfigSpec` object, and then have it converted to the Roboliq protocol format. Your config file will have the following structure:

```
const evowareConfigSpec = {
  // Lab name and robot name
  namespace: "YOUR LAB ID",
  name: "YOUR ROBOT ID",
  // Compiler settings
  config: {
    TEMPDIR: "TEMPORARY DIRECTORY FOR MEASUREMENT FILES",
    ROBOLIQ: "COMMAND TO CALL ROBOLIQ'S RUNTIME",
    BROWSER: "PATH TO WEB BROWSER"
  },
  // Bench sites on the robot
  sites: {
    MYSITE1: {evowareCarrier: "CARRIER ID", evowareGrid: MYGRID, evowareSite: MYSITE},
    ...
  },
  // Labware models
  models: {
    MYPLATEMODEL1: {type: "PlateModel", rows: 8, columns: 12, evowareName: "EVOWARE LABWARE NAME"},
    ...
  },
  // List of which sites and labware models can be used together
  siteModelCompatibilities: [
```

```
  {
    sites: ["MYSITE1", ...],
    models: ["MYPLATEMODEL1", ...]
  },
  ...
],
// List of the robot's equipment
equipment: {
  MYEQUIPMENT1: {
    module: "EQUIPMENT1.js",
    params: {
      ...
    }
  }
},
// List of which lid types can be stacked on which labware models
lidStacking: [
  {
    lids: ["lidModel_standard"],
    models: ["MYPLATEMODEL1"]
  }
],
// List of the robot's robotic arms
romas: [
  {
    description: "roma1",
    // List of sites this ROMA can safely access with which vectors
    safeVectorCliques: [
            { vector: "Narrow", clique: ["MYSITE1", ...] },
            ...
        ]
  },
  ...
],
// Liquid handing arm
liha: {
  // Available tip models
  tipModels: {
    MYTIPMODEL1000: {programCode: "1000", min: "3ul", max: "950ul", canHandleSeal: false, canHandleCel
    ...
  },
  // List of LIHA syringes (e.g. 8 entries if it has 8 syringes)
  syringes: [
    { tipModelPermanent: "MYTIPMODEL1000" },
    ...
  ],
  // Sites that the LIHA can access
  sites: ["MYSITE1", ...],
  // Specifications for how to wash the tips
  washPrograms: {
    // For Example: Specification for flushing the tips with `programCode == 1000`
    flush_1000: { ... },
    ...
```

```
    }
  },
  // Additional user-defined command handlers
  commandHandlers: {
    "MYCOMMAND1": function(params, parsed, data) { ... },
    ...
  },
  // Optional functions to choose among planning alternatives
  planAlternativeChoosers: {
    // For Example: when the `shaker.shakePlate` command has
    // multiple shakers available, you might want to use
    // the one name `MYEQUIPMENT1`
    "shaker.canAgentEquipmentSite": (alternatives) => {
            const l = alternatives.filter(x => x.equipment.endsWith("MYEQUIPMENT1"));
            if (l.length > 0)
                return l[0];
        }
  }
};

const EvowareConfigSpec = require('roboliq-evoware/dist/EvowareConfigSpec.js');
module.exports = EvowareConfigSpec.makeProtocol(evowareConfigSpec);
```

The details about `evowareConfigSpec` are in the Evoware API documenation, and an extensive example can be found in `roboliq-processor/tests/ourlab.js`. This information is probably enough to configure your Evoware robot, and you can skip the rest of this chapter unless you need to setup a different kind of robot.

B.2 Components

A *configuration* is a JavaScript object with the following properties:

- `roboliq`: specifies the version of Roboliq.
- `schemas`: JSON Schema defintitions for all object and commands.
- `objects`: specifies the objects provided by the robot.
- `commandHandlers`: specifies the functions that handle protocol commands, for example by generating low-level commands from high-level commands.
- `predicates`: specifies the *logical predicates* used by Roboliq's A.I. to figure out how to compile some high-level commands to low-level commands for this configuration.
- `objectToPredicateConverters`: specifies functions that generate logical predicates based from `objects`.
- `planAlternativeChoosers`: specifies functions that can choose a specific plan among various alternative plans
- `planHandlers`: specifies the functions that transform logical tasks into commands.

The first four properties are easy for the average JavaScript programmer to understand: they are straight-forward JSON values or JavaScript functions. In contrast, the last four properties (`predicates`, `objectToPredicateConverters`, `planAlternativeChoosers`, and `planHandlers`) rely on concepts from Artificial Intelligence, which will require more effort to grasp.

B.2.1 `roboliq`

As with protocols, the first property of a configuration should be `roboliq: v1`.

B.2.2 `schemas`

For every command and object type, Roboliq requires a schema that defines its properties. By convention, object types begin with an upper-case letter, and commands begin with a lower-case letter. The schemas are written in a standardized format, which you can learn more about at JSON Schema.

Normally, the schemas required for your robot should be automatically provided by your chosen backend and the equipment you select. The `schemas` property is a hash map: its keys are the command names and object type names; its values are the respective JSON schemas. You will not need to write schemas unless you are creating new commands or objects types for Roboliq. An example schema is provided below in the `commandHandlers` section.

B.2.3 `objects`

The `objects` property of a configuration is the same as describe for protocols in the Quick Start chapter. However, robot configurations usually contain complex objects, such as measurement devices, whereas protocols usually only define fairly simple labware. Furthermore, the configuration may contain information that's required for the backend compiler. The object schemas are described in the documentation for standard Roboliq objects and for Evoware objects.

Namespaces. Because a robot "contains" its devices, sites, and permanent labware, we use `Namespace` types to build nested objects. For example, if our lab is named "bsse", we have a robot named "bert", and it has a site named "P1", this could be encoded as follows:

```
objects:
  bsse:
    type: Namespace
    bert:
      type: Namespace
      site:
        type: Namespace
        P1:
          type: Site
          ...
```

We can then reference the site in the protocol as `bsse.bert.site.P1`.

B.2.4 `commandHandlers`

A command handler is a JavaScript function that processes command parameters and returns information to Roboliq about the command's effects, sub-commands, and user messages.

Command handlers are supplied by various modules. Roboliq's command modules are in the subdirectory `roboliq-processor/src/commands`, and Evoware's command modules are in the subdirectory `roboliq-evoware/src/commands`. The API documentation contains information about the available commands.

If you want to write your own command handler, you can add it to `commandHandlers` as a property. The property name should be the name of the command, and the value should be the command handler function. For example, let us make a simplistic function called `my.hello` that tells an Evoware robot to say "Hello, YOURNAME!". First we will defined the schema for the new command:

```
schemas: {
  "my.hello": {
    description: "Say hello to someone",
    properties: {
```

```
      name: {
        description: "Name to say hello to",
        type: "string"
      }
    },
    required: ["name"]
  }
}
```

This schema lets Roboliq know that there's a command named `my.hello`.

```
commandHandlers: {
  "my.hello": function(params, parsed, data) {
    return {
      expansion: [
                {
                    command: "evoware._userPrompt",
                    text: "Hello, "+parsed.value.name",
                }
            ]
    };
  }
}
```

B.3 Logic components

Roboliq uses an Artificial Intelligence method called Hierarchical Task Network (HTN) Planning. In particular, it uses a SHOP2 implementation written by Warren Sack in JavaScript.

B.3.1 `predicates`

A predicate defines a "true statement". For example, the following predicate is named `sealer.canAgentEquipmentProgramMode` and it lets Roboliq know that the robot agent `ourlab.mario.evoware` can use the sealer `ourlab.mario.sealer` with the labware model `ourlab.model.plateModel_96_round_transparent_nunc` at site `ourlab.mario.site.ROBOSEAL` with the internal program file `PerkinElmer_weiss.bcf`.

`yaml "sealer.canAgentEquipmentProgramModelSite":     "agent": "ourlab.mario.evoware" "equipment": "ourlab.mario.sealer"     "program": "C:\\HJBioanalytikGmbH\\...\\PerkinElmer_weiss.bcf", "model": "ourlab.model.plateModel_96_round_transparent_nunc",     "site": "ourlab.mario.site.ROBOSEAL"ya`

The general form for predicates is:

```
predicateName:
  object1: value1
  object2: value2
  ...
```

The values are usually object names, and predicates often define true relationships among objects. All of the predicates together form a database of true statements about the "world" in which the protocol will run. They are used by certain command handlers to automatically figure out valid operations without the user needing to specify the low-level details. (Currently, the end-user documentation does not contain details about which predicates are used by which commands, but it can be found in the command handler source code.)

Here is an example excerpt of using the predicate database to find all valid sealers from the `sealer.sealPlate` command handler:

```
        const predicates = [
            {"sealer.canAgentEquipmentProgramModelSite": {
                "agent": parsed.objectName.agent,
                "equipment": parsed.objectName.equipment,
                "program": parsed.objectName.program,
                "model": model,
                "site": parsed.objectName.site
            }}
        ];
        const [chosen, alternatives] = commandHelper.queryLogic(data, predicates, "sealer.canAgentEquipm
```

The function `commandHelper.queryLogic()` will find all solutions the `predicates` array, filling in the missing values as necessary. If more than one alternative solution is present, it will choose one of them. The solution can either by chosen by a appropriate function `planAlternativeChoosers`, or else Roboliq simply picks the first item in the alternatives list.

B.3.2 `objectToPredicateConverters`

Roboliq's AI needs predicates describing the available objects. These are generated dynamically by the functions supplied in `objectToPredicateConverters`. It is a map from an object type to a function that accepts a named object and returns an array of predicates. For example, here is Roboliq's converter for `Plate` objects:

```
objectToPredicateConverters: {
  Plate: function(name, object) {
        const predicates = [
            { "isLabware": { "labware": name } },
            { "isPlate": { "labware": name } },
            { "model": { "labware": name, "model": object.model } },
            { "location": { "labware": name, "site": object.location } }
        ];
        if (object.sealed) {
            predicates.push({ "plateIsSealed": { "labware": name } });
        }
        return predicates;
    },
    ...
}
```

This converter creates between 4 and 6 predicates for every plate object: `isLabware` lets the AI know that the plate is a kind of labware, `isPlate` says that the plate is a plate, `model` says which labware model the plate has, and `location` says where the plate is located. Furthermore, `plateIsSealed` will be present if-and-only-if the plate is sealed.

You will only need to create additional object-to-predicate converters if you want to extend Roboliq's object types, or perhaps if you create an advanced command that required additional logic.

B.3.3 `planHandlers`

Plan handlers are functions that convert from a planning action to a Roboliq command (usually a low-level command). It is a map from an action name to a function that accepts that action parameters and returns

an array of commands. The function also accepts the parameters of the parent command (the one that generated the plan), and the protocol data, in case those are needed to compute the new command.

An example use-case is using the `transporter.movePlate` command to move a plate into a closed centrifuge: the planning algorithm will include an action to open the centrifuge first, and then the `transporter.movePlate` command will call the appropriate function in `planHandlers` to create the required sub-command.

B.4 Conclusion

Configuring a robot can be complicated. First of all, it requires a lot of detailed knowledge about your robot. Secondly, it involves a lot of interdependencies. For example, in order to support a new labware model, you will need to add the model to the `models` list (easy), but you also need to update the list of site/model compatibilities, the list of safe transport vectors, and perhaps the list of which models accept lids or can be stacked on top of each other. It is certainly possible to achieve, but you are likely to encounter some frustrations if you need to trouble-shoot why you get compiler errors when trying to use a new model, for example.

Appendix C

Advanced Protocols

Advanced protocols can contain many more elements than the simple protocols described in Section 2.2.1. Here we will discuss `imports`, `parameters`, `objects`, `data` properties, variable scope, substitution, directives, and template functions.

C.1 Imports

An `imports` section holds an array that lists the external modules required by a protocol. Roboliq loads the external JSON, YAML, or JavaScript modules before processing the protocol's remaining fields to allow for modularization and re-use of protocols. For example, the import can contain parameters, variables, and objects that are common among several protocols.

C.2 Parameters

Parameters are named values that you define at the beginning of a protocol. You can then use the parameter name instead of the value later in the protocol. In this partial example, a `VOLUME` parameter is defined which is used in the `pipetter.pipette` step:

```yaml
parameters:
  VOLUME:
    description: "amount of water to dispense"
    value: 200 ul
objects:
  ...
steps:
  1:
    command: pipetter.pipette
    sources: water
    destinations: plate1(all)
    volumes: $#VOLUME
```

A parameter should be given a `description` and a `value`. Notice that `$#` prefixes `VOLUME` in the step; `$#` tells Roboliq to substitute in a parameter value. Substitution is discussed in more detail later in this chapter.

C.3 Objects

There are several more object types you might want to use in advanced protocols:

`Data`: for defining a data table. The `Data` type facilitates complex experimental designs, and you can read more about it in the next section and in the chapter on Design Tables.

`Variable`: For defining references to other variables. Variables are not particularly useful in Roboliq, but they could potentially be used to easily switch between objects in case you have, for example, two water sources `water1` and `water2`. In that case you could have a `water` variable whose value to set to the source you want to use:

```
objects:
  water1: ...
  water2: ...
  water:
    description: "water source"
    type: Variable
    value: water1
steps:
  1:
    command: pipetter.pipette
    sources: water
    destinations: plate1(all)
    volumes: 50 ul
```

`Template`: You can use a template to define re-usable steps. Here is a toy example for a template named `dispenseToPlate1` which creates a `pipetter.pipette` command that transfers a volume of water to all wells on `plate1`:

```
objects:
  plate1:
    type: Plate
    ...
  dispenseToPlate1:
    description: "template to dispense `volume` ul of water to all wells on plate1"
    type: Template
    template:
      command: pipetter.pipette
      sources: water
      destinations: plate1(all)
      volumes: "{{volume}}"
steps:
  1:
    command: system.call
    name: dispenseToPlate1
    params:
      volume: 10 ul
  2:
    command: system.call
    name: dispenseToPlate1
    params:
      volume: 50 ul
```

This protocol will first dispense 10ul to all wells, then another 50ul to all wells – not actually a useful protocol, but it illustrates the point. In the `system.call` command, the name of the template is specified along with the parameters template parameters. Templates are expanded using the Handlebars template engine, which

is the reason for the “{{” and “}}” delimiters in the line `volumes: "{{volume}}"`.

C.4 Data

Roboliq supports data tables to enable complex experiments. Conceptually, a data table is like a spread sheet of rows and named columns, where each row represents some “thing”, a each column represents a property. In Roboliq, a data table is an array of JSON objects: each object is a row, and each property is a column. Normally all the objects will have the same set of properties (but this is not required).

Data tables are supported by a `Data` type, a `data` property, a `data()` directive, and a set of `data.*` commands.

C.4.1 `Data` type

You can define a data table using the `Data` object type. Here’s an example where each row has a well, a liquid source, and a volume:

```
objects:
  data1:
    type: Data
    value:
      - {well: A01, volume: 10 ul, source: liquid1}
      - {well: B01, volume: 10 ul, source: liquid2}
      - {well: A02, volume: 20 ul, source: liquid1}
      - {well: B02, volume: 20 ul, source: liquid2}
```

You can define as many data tables as you want in a protocol.

C.4.2 `data` property

After defining a data table, you need to “activate” it for usage. This is done using the `data` property, which understands the following parameters:

- `source`: this is the name of a `Data` object.
- `where`: this is an optional boolean mathjs expression that is evaluated for each data row – only rows for which the expression evaluates to true are activated.
- `orderBy`: an optional array of column names for ordering rows. The ordering behavior is the same as the `_.sortBy` function in lodash.

Any step can be given a `data` property to make a table available in that step and its sub-steps. Here’s an example application:

```
steps:
  1:
    data: {source: data1}
    command: pipetter.pipette
    sources: $$source
    destinationPlate: plate1
    destinations: $$well
    volumes: $$volume
```

The `data` property activates our data table `data1`. The command `pipetter.pipette` can now access the data columns by using the `$$`-prefix along with the column name. So the above example is essentially equivalent to this:

```
steps:
  1:
    command: pipetter.pipette
    sources: [liquid1, liquid2, liquid1, liquid2]
    destinationPlate: plate1
    destinations: [A01, B01, A02, B02]
    volumes: [10 ul, 10 ul, 20 ul, 20 ul]
```

C.4.3 `data.*` commands

Roboliq's `data.*` commands provide two commands for more sophisticated handling of data tables: `data.forEachRow` and `data.forEachGroup`.

`data.forEachRow` lets you run a series of steps on each row of the data table. For each row, the command activates a new data table containing only that row, and it runs its sub-`steps` using that new table. Here's a toy example:

```
steps:
  1:
    data: {source: data1}
    command: data.forEachRow
    steps:
      1:
        command: pipetter.pipette
        sources: $source
        destinationPlate: plate1
        destinations: $well
        volumes: $volume
      2:
        command: fluorescenceReader.measurePlate
        object: plate1
        output:
          joinKey: well
```

In this case, the sub-steps will be repeated 4 times, once for each row. That mean each well will be dispensed into and measured before moving onto the next well. Notice that here only a single \$-prefix was used rather than the double \$\$-prefix for the column variables. When a data table is activated, Roboliq will check if any of the columns have all the same value; if so, that property and value will be automatically added to the current scope (see the next section about Scope). Scope variables are accessible via the \$-prefix. Since the `data.forEachRow` command activates each row individually, all of its columns will be added to the scope.

`data.forEachGroup` lets you operate on groups of rows at a time. You provide a `groupBy` property for it to group by, and then for each group it activates a new data table with those rows and runs its sub-`steps` using that new table. Here's another toy example:

```
steps:
  1:
    data: {source: data1}
    command: data.forEachGroup
    groupBy: source
    steps:
      1:
        command: pipetter.pipette
        sources: $source
        destinationPlate: plate1
```

```
        destinations: $$well
        volumes: $$volume
      2:
        command: fluorescenceReader.measurePlate
        object: plate1
        output:
          joinKey: well
```

Since there are two unique `source` values in `data1`, the `data.forEachGroup` command will create two new data tables for it sub-steps:

First table:

```
- {well: A01, volume: 10 ul, source: liquid1}
- {well: A02, volume: 20 ul, source: liquid1}
```

Second table:

```
- {well: B01, volume: 10 ul, source: liquid2}
- {well: B02, volume: 20 ul, source: liquid2}
```

So the sub-steps will be repeated twice, once for new data table. Notice that here we used the single $-prefix for `source`, but the double $$-prefix for `well` and `volume`. Because the values of the well and volume columns are not the same in all rows, they are not automatically added to the scape, and we canot use the single $-prefix.

C.4.4 `data()` directive

The `data()` directive lets you assign a modified version of your data table to a property value. It will be easier to explain this after the topic of substitution has been covered, so we will postpone the discussion till Section C.6.3.

C.5 Scope

Scope is the set of currently active variables in a step. These usually come from one of two sources: 1) the `data` directive and commands, as discussed above, or 2) a loop command like `system.repeat`, which lets you add an index variable to the scope of the sub-steps.

The scope is a kind of stacked-tower structure. When a step pushes variables into scope, they are available to that step's command and all substeps; however, they are not available to sibling or parent steps.

C.6 Substitution

Substitution lets you work with parameters and data tables by inserting their values into the protocol. Roboliq supports three forms: *template* substitution, *scope* substitution, and *directive* substitution.

C.6.1 Scope $-substitution

In scope substition, an expression starting with $ is replaced with a value from the scope. There are various forms of replacement, which we will dive into now.

`$#...`: pre-scope substitution for parameter values

Parameters are not actually part of the scope, and they are accessible outside of steps as well. This means that they can be used in other parameter values and in object definitions, which is not the case for normal scope variables. You can substitute the value of a parameter named `MYPARAM` by with `$#MYPARAM`.

`${...}`: javascript expression

Roboliq will substitute in the result of a JavaScript expression. The JavaScript expression has access to:

- The scope variables
- `_`: the lodash module and its many functions.
- `math`: the mathjs module and its many functions.

Note that any value JSON value may be returned, whether it is a string, number, boolean, array, or object.

`$(...)`: mathjs calculation

The mathjs module provides a fairly broad range of math operations and is able to handle of units, such as volume. The mathjs expression has access to the current scope variables.

`$...`: scope value substitution

Here you just name the scope variable, and Roboliq will substitute in its value.

If you have activated a data table using the `data` property, then you can use `$colName` to get an array of all the values in the column named `colName`. Furthermore, if any of the data columns are filled with the same value, then that value is added to the scope as `$colName_ONE`, where `colName` is the actual name of the column. For example, if the active data table has a column named `plate` whose entries are all `plate1`, then `$plate1_ONE = "plate1"`.

NOTE: Scope substitution can only be used as a parameter value, but not as a parameter name or part of a longer string. The following uses are invalid:

- `text: "Hello, $name"`: Roboliq only supports scope substitution for an entire value, so the `name` value will not be substituted into this text.
  You can use template substitution for this purpose instead.
- `$myparam: 4`: Roboliq does not support scope substitution for property names. You can use template substitution for this purpose instead.

Examples

Let us look at examples of $-substitutions. Consider this protocol:

```yaml
roboliq: v1
parameters:
  TEXT: { value: "Hello, World" }
objects:
  data1:
    type: Data
    value:
      - {a: 1, b: 1}
      - {a: 1, b: 2}
steps:
  1:
    data: data1
    command: system.echo
    value:
      javascript: "${`${TEXT} ${a} ${__step.command}`}"
      math: "$(a * 10)"
      scopeParameter: $TEXT
      scopeColumn: $b
      scopeOne: $a_ONE
```

```
      scopeData: $__data[0].b
      scopeObjects: $__objects.data1.type
      scopeParameters: $__parameters.TEXT.value
      scopeStep: $__step.command
```

The `system.echo` command will output the object described in its `value` parameter. The resulting value is this:

```yaml
javascript: "Hello, World 1 system.echo"
math: 10
scopeParameter: "Hello, World"
scopeColumn: [1, 2]
scopeOne: 1
scopeData: 1
scopeObjects: "Data"
scopeParameters: "Hello, World"
scopeStep: "system.echo"
```

C.6.2 Template substitution

Template substitution uses the Handlebars template engine to manipulate text. Template substitution occurs on strings that start and end with a tick (‘). Here’s a simple example that produces a new string:

```yaml
text: "`Hello, {{name}}`"
```

If the `name` in the current scope is “John”, then this will set `text: "Hello, John"`.

If the template substitution result is enclosed by braces or brackets, Roboliq will attempt to parse it as a JSON object. Here’s a trivial example that turns a template substitution into a command, assuming that `name` is currently in scope:

```yaml
1: `{command: "system._echo", text: "Hello, {{name}}"}`
```

C.6.3 Directive ()-substitution

Directives are substitution functions. The main one is the `data()` directive, which was briefly mentioned above in the Data section. The other directives are also closely related to `Data` objects, and they are discussed more in Appendix E on Design Tables.

The `data()` directive lets you assign a modified version of your data table to a property value. The directive can take several properties:

- `where`: same as for the `data` property, this lets you select a subset of rows in the active data table.
- `map`: each row in the data table will be mapped to this value. This is how you can transform your rows.
- `summarize`: like `map`, but for summarizing all the rows into a single row. Summarize has a particularity: all column names are pushed into the current scope as arrays, and they are not overwritten by a single common value even if the column only contains a single value.
- `join`: a string separator that will be used to join all elements of the array (see `Array#join` in some JavaScript documentation).
- `head`: if set to `true`,

Let us consider some examples using the `data1` table from above. Here is the table:

```yaml
- {well: A01, volume: 10 ul, source: liquid1}
- {well: B01, volume: 10 ul, source: liquid2}
```

```yaml
- {well: A02, volume: 20 ul, source: liquid1}
- {well: B02, volume: 20 ul, source: liquid2}
```

and here are the examples:

Directive:
`data(): {where: 'source == "liquid1"'}}`
Result:

```yaml
- {well: "A01", volume: "10 ul", source: "liquid1"}
- {well: "A02", volume: "20 ul", source: "liquid1"}
```

Directive:
`data(): {map: '$volume'}`
Result:

```yaml
["10 ul", "10 ul", "20 ul", "20 ul"]
```

Directive:
`data(): {where: 'source == "liquid1"', map: '$(volume * 2)'}`
Result:

```yaml
["20 ul", "40 ul"]
```

Directive:
`data(): {map: {well: "$well"}}`
Result:

```yaml
- {well: "A01"}
- {well: "B01"}
- {well: "A02"}
- {well: "B02"}
```

Directive:
`data(): {map: "$well", join: ","}`
Result:

```yaml
"A01,B01,A02,B02"
```

Directive:
`data(): {summarize: {totalVolume: '$(sum(volume))'}}`
Result:

```yaml
- {totalVolume: "60 ul"}
```

Directive:
`data(): {groupBy: "source", summarize: {source: '${source[0]}', totalVolume: '$(sum(volume))'}}`
Result:

```yaml
- {source: "liquid1", totalVolume: "30 ul"}
- {source: "liquid2", totalVolume: "30 ul"}
```

Appendix D

Software Implementation Details

This appendix chapter goes into the most important details of the software implementation that were not described in Chapter 2.

D.1 Well and Liquid Selection

This section contains addition information about well and liquid selection from Section 2.2.3.

Examples

Here are further examples for a plate `p` with 96 wells and a liquid source `q` in a tube:

1. `p(A1)` or `p(A01)` or `p(A001)`: well `A1` on plate `p`
2. `p(all)`: all wells on plate `p`
3. `q`: all wells with liquid `q` (in this case the single well in the tube containing `q`)
4. `p(A1) + q`: well `A1` on plate `p` and liquid `q`
5. `p(A1, A2)` or `p(A1) + p(A2)`: wells `A1` and `A2` on plate `p`
6. `p(A1 down to D1)`: select wells starting from `A1` and moving down the plate until `D1` is reached – so wells `A1`, `B1`, `C1`, `D1` on plate `p`
7. `p(A1 down take 4)`: select wells starting from `A1` and moving down the plate until 4 wells have been selected – so wells `A1`, `B1`, `C1`, `D1` on plate `p`
8. `p(A1 right to A4)` or `p(A1 right take 4)`: analogous to above but moving right instead of down – so wells `A1`, `A2`, `A3`, `A4` on plate `p`
9. `p(A1 down to C3)` or `p(A1 down take 20)`: select wells starting from `A1` and moving down the plate until `C3` or 20 wells have been selected
10. `p(A1 down block B2)`: select wells in a rectangle block between `A1` and `B2` starting from `A1` and moving down until `B2` is reached – so wells `A1`, `B1`, `A2`, `B2` on plate `p`
11. `p(A1 right block B2)`: wells `A1`, `A2`, `B1`, `B2` on plate `p`
12. `p(all random(0) take 4)`: select all wells, then randomize their order with random seed 0, and take the first 4 – provides 4 randomly selected wells, which happen to be `C09`, `A06`, `B04`, `E10`
13. `p(A1 down take 8 row-jump(1))`: like `p(A1 down take 8)`, but the ordering is altered to allow for spacing between rows – so wells `A1`, `C1`, `E1`, `G1` then `B1`, `D1`, `F1`, `H1`

Grammar

Roboliq's parser for well selections is based on a *parsing expression grammar* (PEG) [128] and the `pegjs` library [129]. Below we list the PEG grammar as processed by `pegjs`:

```
start
  = init:(x:entity ws '+' ws { return x; })* last:entity
```

```
    { return init.concat.apply([], init).concat(last); }

ws = [ \t]*

spaces = [ \t]+

entity
  = labwareClause
  / wellClause
  / source

labwareClause
  = labware:ident ws '(' ws clauses:wellClauses ws ')'
    { return clauses.map(function (clause) {
        return {labware: labware, subject: clause.subject, phrases: clause.phrases}; }); }

source
  = source:ident
    { return {source: source}; }

ident
  = init:(x:identPart ws '.' ws { return x; })* last:identPart
    { return init.concat([last]).join('.'); }

identPart
  = first:[A-Za-z_] rest:[0-9A-Za-z_]*
    { return first.toString() + rest.join('').toString(); }
  / [A-Za-z_]

wellClauses
  = init:(x:wellClause ws ',' ws { return x; })* last:wellClause { return init.concat([last]); }

wellClause
  = subject:wellSubject phrases:wellPhrases?
  { return {subject: subject, phrases: phrases}; }

wellSubject
  = well
  / "all"

  // well on labware; matches strings such as "A01"
well
  = row:[A-Z] col:integer
  {
    var columnText = col.toString();
    if (columnText.length < 2) columnText = "0" + columnText;
    return row.toString()+columnText;
  }

integer
  = digits:[0-9]+ { return parseInt(digits.join(""), 10); }

wellPhrases
  = phrases:(spaces x:wellPhrase { return x; })*
```

```
wellPhrase
  = 'down' spaces 'block' spaces ('to' spaces)? to:well { return ["down-block", to]; }
  / 'down' spaces ('take' spaces)? n:integer { return ["down", n]; }
  / 'down' spaces ('to' spaces)? to:well { return ["down-to", to]; }
  / 'right' spaces 'block' spaces ('to' spaces)? to:well { return ["right-block", to]; }
  / 'right' spaces ('take' spaces)? n:integer { return ["right", n]; }
  / 'right' spaces ('to' spaces)? to:well { return ["right-to", to]; }
  / 'random' ws '(' ws seed:integer ws ')' { return ['random', seed]; }
  / 'random' ws '(' ws ')' { return ['random']; }
  / 'take' spaces n:integer { return ['take', n]; }
  / 'row-jump' ws '(' ws n:integer ws ')' { return ['row-jump', n]; }
```

D.2 Commands

Command names are composed of two identifiers separated by a period: a module and a function within the module. For example, the command `transporter.movePlate` refers to the `transporter` module and its `movePlate` function. Low-level functions begin with an underscore `_`, such as the command `transporter._movePlate`. In Table D.1 we list Roboliq's standard commands – each module name is printed in bold font, and its functions follow with a short description. The detailed documentation is available online [87, 130].

Although Roboliq defines the low-level commands in Table D.1, the compiler backend needs to actually handle them. In addition to those low-level commands, our compiler backend for the Tecan Evoware provides several Evoware-specific low-level commands shown in Table D.2 and documented in more detail online [130].

Table D.1: List of Roboliq's standard commands

Command	Short description
absorbanceReader	
`measurePlate`	Measure the absorbance of a plate.
centrifuge	
`centrifuge2`	Centrifuge two plate.
`insertPlates2`	Insert up to two plates into the centrifuge.
data	
`forEachGroup`	Perform sub-steps for every grouping of rows in the active data table.
`forEachRow`	Perform sub-steps for every row in the active data table.
equipment	
`_run`	Run the given equipment.
`close`	Close the given equipment.
`open`	Open the given equipment.
`openSite`	Open an equipment site.
`start`	Start the given equipment.
`stop`	Stop the given equipment.
fluorescenceReader	
`measurePlate`	Measure the fluorescence of wells on a plate.
incubator	

Command	Short description
incubatePlates	Incubate the given plates
insertPlates	Insert up to two plates into the incubator.
run	Run the incubator with the given program
pipetter	
_aspirate	Aspirate liquids from sources into syringes.
_dispense	Dispense liquids from sryinges into destinations.
_measureVolume	Measure well volume using pipetter tips.
_mix	Mix liquids by aspirating and re-dispensing.
_pipette	Pipette liquids from sources to destinations.
_punctureSeal	Puncture the seal on a plate using pipetter tips.
_washTips	Clean the pipetter tips by washing.
cleanTips	Clean the pipetter tips.
measureVolume	Measure well volume using pipetter tips.
mix	Mix well contents by aspirating and re-dispensing.
pipette	Pipette liquids from sources to destinations.
pipetteDilutionSeries	Pipette a dilution series.
pipetteMixtures	Pipette the given mixtures into the given destinations.
punctureSeal	Puncture the seal on a plate using pipetter tips.
scale	
weigh	Weigh an object
sealer	
sealPlate	Seal a plate.
shaker	
run	Run the shaker. This will simply run the given shaker program, regardless of
shakePlate	Shake a plate.
system	
_description	Include the value as a description in the generated script.
_echo	Include the value as an echo statement in the generated script for trouble-shooting.
call	Call a template function.
description	Include the value as a description in the generated script.
echo	Include the value as an echo statement in the generated script for trouble-shooting.
if	Conditionally execute steps depending on a conditional test.
repeat	Repeat sub-steps a given number of times.
runtimeExitLoop	Perform a run-time check to test whether execution should exit the current loop.
runtimeLoadVariables	Load the runtime values into variables.
runtimeSteps	Handle steps that require runtime variables.
timer	
_sleep	Sleep for a given duration using a specific timer.
_start	Start the given timer.
_stop	Stop the given timer.
_wait	Wait until the given timer has reacher the given elapsed time.

Command	Short description
`doAndWait`	Start a timer, perform sub-steps, then wait till the given duration has elapsed.
`sleep`	Sleep for a given duration.
`start`	Start a timer.
`stop`	Stop a running a timer.
`wait`	Wait until the given timer has reacher the given elapsed time.
transporter	
`_moveLidFromContainerToSite`	Transport a lid from a container to a destination site.
`_moveLidFromSiteToContainer`	Transport a lid from an origin site to a labware container.
`_movePlate`	Transport a plate to a destination.
`doThenRestoreLocation`	Perform steps, then return the given labwares to their prior locations.
`moveLidFromContainerToSite`	Transport a lid from a container to a destination site.
`moveLidFromSiteToContainer`	Transport a lid from an origin site to a labware container.
`movePlate`	Transport a plate to a destination.

Table D.2: List of low-level commands for Tecan Evoware

Command	Short description
evoware	
`_execute`	An Evoware Execute command
`_facts`	An Evoware FACTS command
`_raw`	An Evoware direct command
`_subroutine`	An Evoware ‘Subroutine’ command
`_userPrompt`	An Evoware UserPrompt command
`_variable`	Set an Evoware variable

D.3 Core Algorithms

Roboliq’s core algorithms are listed in Figures D.1 and D.2.

D.4 Software Dependencies

Most of Roboliq’s source code is written in JavaScript and runs on the Node.js engine. A popular library system called `npm` has grown up around Node.js – as of this writing, it holds almost 500,000 libraries conveniently available for use in JavaScript projects. Among the many libraries that Roboliq employs, here are the most noteworthy ones:

- `lodash`: a large functional library of generally utilities; it helps makes JavaScript a palatable language.
- `mathjs`: a library of math functions and parsers that Roboliq uses to handle math expressions in protocols.
- `handlebars`: the template library Roboliq uses for template substitution in protocols.
- `jsonschema`: a library for validating JavaScript objects based on JSON schema definitions.
- `yamljs`: a library for reading and writing YAML files.
- `randomjs`: a library for random number generation; it allows us to ensure consistent results across platforms.
- `markdown-it`: a library to allow for rich-text formatting in `description` properties.

```
mergeObjects(o1, o2):
  result = empty object
  keys = union of keys in o1 and o2
  for each key:
    if o1[key] and o2[key] are in both objects:
      result[key] = mergeObjects(o1[key], o2[key])
    else if o2 has key:
      result[key] = o2[key]
    else:
      result[key] = o1[key]

loadProtocol(url, params):
  protocol = load url as JSON, YAML, or JavaScript with params
  if protocol has 'import' key:
    module = empty object
    for each requirement in protocol.requires:
      protocol2 = loadProtocol(requirement url, requirement params)
      module = mergeObjects(module, protocol2)
    protocol = mergeObjects(module, protocol)
    remove 'import' key from protocol
  return protocol
```

Figure D.1: **Pseudocode for merging objects and loading protocols.** JSON data consists of several types of values: basic values such as numbers and strings, arrays, and objects. A JSON object is a collection of key/value pairs. `mergeObjects` inspects two objects, whereby the fields of the second object have priority – if they both have a particular key whose values are also objects, those values are recursively merged; if instead the second object has the key, take its value; otherwise take the value from the first object. `loadProtocol` loads the given URL as a JSON object, a YAML object, or a JavaScript function (to which it passes extra parameters if supplied). If the loaded JSON object has an `import` key, the algorithm will recursively load the required modules and merge them.

```
expandProtocol(protocol):
  objects = clone a copy of protocol.objects
  expandStep(protocol, "", objects);

expandStep(protocol, id, objects):
  step = lookup step with id in protocol
  if step has 'command' key:
    predicates = protocol.predicates ++ objectPredicates(objects)
    handler = protocol.commandHandlers[step.command]
    result = handler(step, objects, predicates, protocol.planHandlers)
    protocol.cache[id] = result
    protocol.errors[id] = result.errors
    abort if there were errors
    if result has 'expansion' key:
      merge result.expansion into step (mutates protocol too)
    protocol.effects[id] = result.effects
    for effect in result.effects:
      merge effect into objects
  for each substep in step:
    substepId = id + '.' + substep index
    expandStep(protocol, substepId, objects)
```

Figure D.2: **Pseudocode for expanding the steps of a protocol.** `expandProtocol()` starts the expansion process by cloning a mutable copy of the protocol's objects and calling `expandStep()`. In `expandStep()`, we first check whether the current step contains a command. If so, the original predicates are merged with the dynamic object predicates, the command handler is invoked, its results are stored, and errors, expansions, and effects are handled. Finally, if the step has sub-steps, each of them is expanded in turn.

Another significant dependency is the `babel` transpiler system to support some advanced language constructs that node.js does not yet support natively. However, this turned out to be an unfortunate choice, because it complicates the deployment process significantly and should be removed.

D.5 JSON Schemas

In Roboliq protocols, the objects and steps are declared as JSON objects — and their structures are defined according to JSON Schema [84], with a couple extensions to accommodate the requirements of Roboliq. JSON Schema lets you define which properties a JSON object must or can have, and what kinds of values those properties should have. Internally, Roboliq uses the schemas in several ways:

- Validation: each object and command gets validated against its JSON Schema. If any required properties are missing, the user is shown an error message.
- Pre-processing for command handlers: some values require special processing, such as when specifying wells (see Section 2.2.3). If a property's type is "Wells", then the string value is parsed into the corresponding set of wells, and that array is passed to the command handler instead of the raw string.
- Documentation: the documentation for Roboliq's objects and commands generated from the schema definitions. This ensures that the source code and the documentation are really in sync.

Schemas can also be vital for programming GUIs that should work with protocols. Though not part of this thesis, I programmed a specialized GUI for a couple use-cases, and the schemas played an important role: since the schemas tell us about an object's structure, the GUI can help a user build a protocol and validate protocols.

All schemas needed for a protocol should be in the `schemas` property, which would normally be in the configuration file rather than in the protocol file. The `schemas` property is a key/value dictionary whose keys are the names of commands or object types, and whose values are JSON schemas with Roboliq extensions. Figure D.3 shows an example of two schemas in Roboliq. By convention, object types begin with a capital letter, and commands begin with a lower-case letter.

Every object type has a `type` property whose value must be the name of the object type. This is how Roboliq identifies which schema is used to validate an object. Many commands have the properties `agent`, `equipment`, and `program`. The `agent` property identifies who or what will be handling the command – in this thesis, the agent is always one of our Tecan Evoware robots, but in principle it could also be a person, another type of robot, or a piece of software. The `equipment` property tells which equipment should be used when running the command – for example, if the robot has two shakers, you can explictly choose which one to use. The `program` property allows you to select or define an equipment-specific program or specify execution details. These properties are typically required for low-level commands, but they are optional for high-level commands because the automated planning algorithm finds valid values.

D.6 Input Formats

Roboliq accepts protocols in four input formats:

- JSON
- YAML
- JavaScript Node.js module that outputs an object
- JavaScript Node.js module that outputs a function

JSON stands for JavaScript Object Notation. It is a simple format for encoding software data, and it has gradually become the defacto format of choice for transfering data on the web.

```
Sealer:
  description: Sealing equipment.
  properties:
    type: {enum: [Sealer]}
    description: {description: "Description of this sealing equipment", type: markdown}
  required: [type]

sealer.sealPlate:
  description: "Seal a plate."
  properties:
    agent: {description: "Agent identifier", type: "Agent"}
    equipment: {description: "Equipment identifier", type: "Equipment"}
    program: {description: "Program identifier for sealing", type: "string"}
    object: {description: "Plate identifier", type: "Plate"}
    site: {description: "Site identifier of sealer", type: "Site"}
    count: {description: "Number of times to seal (defaults to 1)", type: number, default: 1}
    destinationAfter: {description: "Site to move the plate to after this command", "type": "SiteOrStay"}
  required: ["object"]
```

Figure D.3: **Example of a JSON Schema definition in Roboliq**. These are excerpts from Roboliq's definitions of the `Sealer` object type and the `sealer.sealPlate` command. The `Sealer` object type is used to define sealing equipment. Object of type `Sealer` are declared for sealing equipment; they require a property `type: Sealer`, they can have an optional description in the popular Markdown text format. The only required property is `type`. The `sealer.sealPlate` command has seven properties; the optional `agent`, `equipment`, and `program` properties are common to most commands; the `object` property tells which plate to seal; the remaining properties are optional.

YAML is a JSON-like format that is more legible. Due to its superior legibility, this manual usually displays protocols in YAML.

JavaScript is the lingua-franca of the web, and Node.js is the most popular platform for running JavaScript applications outside of browsers. You probably will not need JavaScript for Roboliq, unless you get into more advanced applications. If you do use JavaScript, your module may either 1) output a JavaScript object (i.e. `module.exports = myOjbect;`) or 2) a function that accepts configuration parameters and returns an object (i.e. `module.exports = function(options) { ... return myOjbect; };`).

D.6.1 Data Structure for Well Contents

Roboliq tracks the contents of wells with each pipetting operation. The data structure for well contents is a recursive nested array that is constructed as follows:

- `[]`: empty well
- `[VOLUME, LIQUIDNAME]`: a certain volume of a single liquid
- `[TOTALVOLUME, [RELVOLUME1, CONTENTS1], [RELVOLUME2, CONTENTS2], ...]`: a total volume of a mixtures of other liquids

The first two cases are simple. In the third case, the `TOTALVOLUME` tells the total well volume. The other volumes are *relative* volumes – to find the absolute volume of the ith contents, you need to calculate

$$VOLUME_i = TOTALVOLUME \cdot \frac{RELVOLUME_i}{\sum_i RELVOLUME_i}$$

It is useful to keep the relative volumes rather than recalculating the absolute volumes in two scenarios: 1) when removing liquid from a well, we only need to update the total volume, and 2) when transferring an aliquot of a mixture (call it `CONTENTSX`) to another well, we can just append that to the content list of the destination well along with the volume transfered (e.g. `[10 ul, CONTENTSX]`), so it is not necessary to recalculate the absolute values of the components. This helps avoid rounding errors and unnecessary

calculations. The structure is recursive, because the components above labeled `CONTENTS1` and `CONTENTS2` have the same structure. Here are some examples:

- `[100 ul, water]`: 100 µL water
- `[30ul, [10ul, reagent1], [20ul, reagent2]]`: 30 µL mixture of 10 µL reagent1 and 20 µL reagent2
- `[30ul, [20ul, reagent1], [40ul, reagent2]]`: same as above, but the higher relative volumes might be because 30 µL was already removed from an original volume of 60 µL
- `[30ul, [10ul, reagent1], [20ul, [10ul, reagent1], [10ul, reagent2], [20ul, water]]]`: 30 µL mixture of 15 µL reagent1, 5 µL reagent2, and 10 µL water

D.6.2 Backend Compilers and the `agent` Property

The Roboliq processor outputs a detailed instruction list, whose low-level instructions have fairly detailed parameters describing what needs to be done. The backend compiler (that generates the actual robot program) therefore requires relatively little sophistication and can be completely decoupled from the rest of Roboliq and written in any language.

A backend is configured to be responsible for certain `agents` – for example, if you have Tecan robots name `mario` and `luigi` like we do, then the Tecan backend should compile the instructions whose `agent` property is set to one of those names. In this way, its possible to assign instructions to multiple robots and to multiple backends within a single Roboliq protocol.

D.6.3 Execution

D.6.3.1 Execution on a Tecan Evoware Robot

During my PhD research, I only used Tecan Evoware robots and did not program backends for other models, but the software structure is modular, so other backends can also be added. The typical steps in running a Roboliq protocol on a Tecan Evoware robot are:

1. Write a configuration for your lab or use one which has been written by someone else.
2. Write a script for your protocol using the Roboliq syntax and commands.
3. Compile your script and configuration to create a program (.ESC) to be executed by your robot.
4. Open the Tecan Evoware script (.ESC) in Evoware, and run it.
5. Load Roboliq's measurement data into R or another statics program to analyze the data.

Roboliq's online user manual [86] contains a walk-through tutorial for the details of how to install and run the software.

D.6.3.2 Roboliq's Output Files

Roboliq generates a directory structure with several files for each experiment. When you invoke Roboliq, you pass it the path to a root directory for your experiments. By default, Roboliq creates a new directory with the same name as your protocol (more specifically, with the name of the last protocol file you pass to it, minus file extensions such as `.yaml` or `.json`). The follwoing files will be saved in that directory:

- The instruction list with the extension `.out.json` – this file contains the merged input files along with the low-level instructions generated by the command handlers.
- The robot program(s) created by the backend compiler (for Tecan Evoware, the programs have an `.ESC` extension).
- An `index.html` file that tells the user how to setup the bench, how much of the source liquids will be used, and what the final well contents should be.

D.6.3.3 Runtime Files

While executing a Roboliq experiment, more files and directories are generated. First a “run” directory is created that holds measurements made during the current experimental run; The directory is named by the date and time that execution started. Additionally, a `.run` file is created with the name of the run directory for easy lookup. When measurement are made, they may initially be saved in a temporary directory specified in the configuration file. Afte the measurement is done, the files are moved to the run directory and processed to extract the values to a JSON file. The JSON measurement files may be further concatenated into a larger new-line delimited JSON file for convenient loading during analysis (depending on the measurement command and the parameters specified in the experiment).

We tested several approaches before settling on the current one. One approach that seemed promising was using HDF5 files [131] for storing all the protocol information and measurement data. HDF5 was designed for sharing large amounts of diverse scientific data in a single file. We eventually abandoned this approach, however, because the libraries did not offer convenient cross-platform and cross-language support, but we want Roboliq to be easily used by third-party software.

For that reason, we instead ended up using JSON files and a simple directory structure. This approach keeps a clean separation between experiments and their various runs.

D.6.3.4 Runtime Callbacks

For Tecan Evoware, the generated ESC files usually make several calls back to Roboliq:

- At the very beginning of the script, the file `index.html` is opened in a browser (this behavior can be suppressed) for the user to confirm that everything is setup correctly before the robot performs any actions.
- After the user confirms the bench setup, the script calls a Roboliq command-line utility to create the run directory.
- After each measurement, the script calls Roboliq to process the measurement files.

These callbacks allow Roboliq to augment the run-time behavior that is possible to achieve with Tecan Evoware alone.

D.6.3.5 Runtime Control

The Roboliq protocols described in this thesis were designed to execute a fixed sequence of commands from start to finish. However, we also ran several experiments that required dynamic run-time control in response to measurements. One example was diluting a well until its absorbance measurement passed a certain threshold.

Roboliq’s automated planning methods need to know the state of all wells, labwares, bench sites, and equipment – otherwise they ca not generate plans. So the processor needs to suspend when it reaches a point where the next step is not know at compile time because it depends on the value of a measurement. The output is then a partial instruction list which can be compiled, but it has two special instructions at the end. The second-to-last instruction is to resume processing with the measured data, allowing Roboliq to emulate the following steps. The last instruction is to load the newly generated robot program and continue execution.

Appendix E

Design Tables

Here we use the term *Design Table* to refer to a data table that is designed for an experiment. It should contain all relevant factors for later analyzing the experimental results.

Experiments on microwell plates can easily involve hundreds of unique liquid combinations, so specifying them manually can be tedious and error-prone. Here we present a short-hand for creating design tables.

WARNING: This is very abstract. Do not bother with it if you are looking for something easy. That said, it makes complex experiments much, much easier to design, trouble-shoot, and analyze.

NOTE: In this chapter, we use the following terms interchangeably: column, factor, variable.

E.1 First examples

Let us create a design table with a single row like this:

plate	source	destination	volume
plate1	water	A01	25 ul

This table could be used to specify a single pipetting operation that dispenses 25 ul of water into well `A01` of `plate1`. We will specify this in Roboliq as follows:

```
roboliq: v1
objects:
  data1:
    type: Data
    description: First example
    design:
      plate: plate1
      source: water
      destination: A01
      volume: 25 ul
---
```

In a Roboliq protocol, designs are placed under the `objects` property. In this case, its name is `data1`, its type is `Data`, and it has a description. The `design` property is where factors are specified.

Let us expand the design table a bit to dispenses 25 ul of water into several destinations.

plate	source	destination	volume
plate1	water	A01	25 ul
plate1	water	B01	25 ul
plate1	water	C01	25 ul

To specify this in Roboliq, we change the `destination` property to a *branching* factor:

```
roboliq: v1
objects:
  data1:
    type: Data
    description: First example
    design:
      plate: plate1
      source: water
      destination*: [A01, B01, C01]
      volume: 25 ul
```

Notice the asterisk in `destination*`. An asterisk at the end of a factor name indicates *branching*. Branching factors require an array of values, and for each value in the array, the existing rows are first replicated and then the value is assigned to the factor in that row.

Now let us assign a different volume to each row:

plate	source	destination	volume
plate1	water	A01	25 ul
plate1	water	B01	50 ul
plate1	water	C01	75 ul

To specify this in Roboliq, we change the `volume` property to an array:

```
roboliq: v1
objects:
  data1:
    type: Data
    description: First example
    design:
      plate: plate1
      source: water
      destination*: [A01, B01, C01]
      volume: [25 ul, 50 ul, 75 ul]
```

When a factor value is an array, a new column is added with those values.

Try it. Copy the above code to a new file in Roboliq's root directory named `data1Test.yaml`. Open a terminal and change directory to the Roboliq root, and run the following command:

```
npm run design -- --path objects.data1 data1Test.yaml
```

It should produce this output:

```
plate   source  destination  volume
======  ======  ===========  ======
plate1  water   A01          25 ul
plate1  water   B01          50 ul
```

```
plate1  water   C01          75 ul
======  ======  ===========  ======
```

Branches can also be specified as integers. If you specify an integer, it creates that many branches which are each given an integer index, as follows:

```
roboliq: v1
objects:
  designInteger:
    type: Data
    design:
      a*: 3
      b*: 3
```

Which creates the following table:

a	b
1	1
1	2
1	3
2	1
2	2
2	3
3	1
3	2
3	3

E.2 Nested branching

Nested branching provides a lot of power to the design specification, but it is also where the complexity starts. Consider this table where two sources are nested in each destination, and each source has its own volume:

plate	destination	source	volume	liquidClass
plate1	A01	water	50 ul	Roboliq_Water_Air_1000
plate1	A01	dye	25 ul	Roboliq_Water_Air_1000
plate1	B01	water	50 ul	Roboliq_Water_Air_1000
plate1	B01	dye	25 ul	Roboliq_Water_Air_1000
plate1	C01	water	50 ul	Roboliq_Water_Air_1000
plate1	C01	dye	25 ul	Roboliq_Water_Air_1000

This can be described in Roboliq as follows:

```
roboliq: v1
objects:
  data2:
    type: Data
    description: Nested example
    design:
      plate: plate1
      destination*: [A01, B01, C01]
      source*:
        water:
```

```
          volume: 50 ul
        dye:
          volume: 25 ul
      liquidClass: Roboliq_Water_Air_1000
```

Create a file named `data2Test.yaml` with those contents and run:

```
npm run design -- --path objects.data2 data2Test.yaml
```

Let us walk through this example step-by-step to see how the desired table is achieved.

Step 0: A design starts as a single empty row.

Step 1: `plate: plate1`

This assigns the value `plate1` to the property `plate` in the first row:

plate
plate1

Step 2: `destination*: [A01, B01, C01]`

Three copies of the previous row are created, and a column for `destination` is added to each row, each with its own value:

plate	destination
plate1	A01
plate1	B01
plate1	C01

Step 3: `source*`

This branch has two keys: water and dye. So to start with, two copies are made of each of the previous three rows. The first copy is updated according to the first key, giving us:

plate	destination	source
plate1	A01	water
plate1	B01	water
plate1	C01	water

Then the conditions embedded under `water:` are applied to those rows – in this case, `volume = 50 ul`:

plate	destination	source	volume
plate1	A01	water	50 ul
plate1	B01	water	50 ul
plate1	C01	water	50 ul

For the second table copy, an analogous process sets `source = dye` and `volume = 25 ul`:

plate	destination	source	volume
plate1	A01	dye	25 ul

plate	destination	source	volume
plate1	B01	dye	25 ul
plate1	C01	dye	25 ul

Next those two tables are concatenated, giving us:

plate	destination	source	volume
plate1	A01	water	50 ul
plate1	A01	dye	25 ul
plate1	B01	water	50 ul
plate1	B01	dye	25 ul
plate1	C01	water	50 ul
plate1	C01	dye	25 ul

Step 4: `liquidClass: Roboliq_Water_Air_1000`

Finally, `liquidClass = Roboliq_Water_Air_1000` is assigned to all rows:

plate	destination	source	volume	liquidClass
plate1	A01	water	50 ul	Roboliq_Water_Air_1000
plate1	A01	dye	25 ul	Roboliq_Water_Air_1000
plate1	B01	water	50 ul	Roboliq_Water_Air_1000
plate1	B01	dye	25 ul	Roboliq_Water_Air_1000
plate1	C01	water	50 ul	Roboliq_Water_Air_1000
plate1	C01	dye	25 ul	Roboliq_Water_Air_1000

E.3 Hidden factors

There are generally many ways to achieve the same results. As an example, an alternative way of achieving the same result as above is:

```
roboliq: v1
objects:
  data2:
    type: Data
    description: Nested example
    design:
      plate: plate1
      destination*: [A01, B01, C01]
      .sourceId*:
      - source: water
        volume: 50 ul
      - source: dye
        volume: 25 ul
      liquidClass: Roboliq_Water_Air_1000
```

In this case, the branching factor is `.sourceId*` and it is an array. The period prefix hides that column, and the final results are the same as above.

E.4 Actions

Roboliq provides various designs “actions” that can be used for more sophisticated values. The most important ones are:

- allocateWells
- range
- calculate
- case

E.4.1 `allocateWells`

Let us take a look at an example:

```
replicate*: 2
well=allocateWells:
  rows: 8
  columns: 12
```

An action is indicated with the “=”-infix. So in the case of `well=allocateWells`, the factor name is `well`, the action is `allocateWells`, and the properties are the arguments to the action. In this case, `rows` and `columns` tells the plate dimension we want to get wells for, and 2 wells will be allocated since the table has two rows:

```
{replicate: 1, well: A01}
{replicate: 2, well: B01}
```

E.4.2 `range`

The `range` action gives you an integer sequence. It accepts these arguments:

- `from`: the integer to start at (optional)
- `till`: the integer to end at (optional)
- `step`: the distance between generated integers (default = 1)

Here’s an example:

```
a*: 2
b*: 2
c=range: {}
d=range: {from: 10, step: 10}
```

Which produces this result:

a	b	c	d
1	1	1	10
1	2	2	20
2	1	3	30
2	2	4	40

The first range, `c`, just numbers all the rows starting with 1. The second range, `d`, starts numbering at 10 and procedes in steps of 10.

E.4.3 `calculate`

The `calculate` action takes a string to be parsed by mathjs. The calculation will be made for each row individually. Here's an example:

```
a*: 3
volume=calculate: '(a * 10) ul'
more=calculate: '(50 ul) - volume'
```

And here is the result:

a	volume	more
1	10 ul	40 ul
2	20 ul	30 ul
3	30 ul	20 ul

Alternatively, the `calculate` action can accept parameters:

- `value`: the string to parse
- `units`: the units of the final output

```
a*: 3
volume=calculate:
  value: '(a * 10)'
  units: ul
```

With this output:

a	volume
1	10 ul
2	20 ul
3	30 ul

E.4.4 `case`

A `case` action takes an array of cases and tests them against each row of the table. The first case whose `where` statement is missing or evaluates to `true` will be applied. The individual case items take these arguments:

- `where` - an optional mathjs statement that will be evaluated on each row
- `design` - a design specification that will be applied to the matching rows

Here's an example:

```
a*: 3
volumeCase=case:
  - where: a < 2
    design:
      volume: 10 ul
  - design:
      volume: 12354 ul
```

a	volumeCase	volume
1	1	10 ul
2	2	12345 ul

a	volumeCase	volume
3	2	12345 ul

E.5 Step and data nesting

You can only load one design per step, but you can nest steps and load another design in the sub-step. Consider these two excerpts of designs:

```
data1:
  design:
    a: Alice
    b: *3
    c: Charles
    d: Daniel

data2:
  design:
    d: David
```

Let us use them in these steps:

```
1:
  data: {source: data1}
  description: "`{{a}} {{c}} {{d}}`"
  1:
    data: {source: data2}
    description: "`{{a}} {{c}} {{d}}`"
```

The descriptions should be expanded as follows:

1.description: "Alice Charles Daniel" 1.1.description: "Alice Charles David"

In 1.1, "$$b" does not exist, but "$a" and "$c" still do. That is to say: column data from a previous `data` directive are not carried into sub-steps with a new `data` directive, but the values that were the same for all columns remain in scope.

Appendix F

Additional Figures

```
phDesign:
  type: Data
  randomSeed: 0
  design:
    gfp*: [sfGFP, Q204H_N149Y, tdGFP, N149Y, Q204H]
    buffer*:
      acetate:
        acid: acetate_375
        base: acetate_575
        acidPH: 3.75
        basePH: 5.75
        baseVolume*=range: {count: 8, from: 0, till: 30, decimals: 1, units: ul}
      mes:
        acid: mes_510
        base: mes_710
        acidPH: 5.10
        basePH: 7.10
        baseVolume*=range: {count: 7, from: 0, till: 30, decimals: 1, units: ul}
      pipes:
        acid: pipes_575
        base: pipes_775
        acidPH: 5.75
        basePH: 7.75
        baseVolume*=range: {count: 5, from: 0, till: 30, decimals: 1, units: ul}
      hepes:
        acid: hepes_650
        base: hepes_850
        acidPH: 6.50
        basePH: 8.50
        baseVolume*=range: {count: 5, from: 0, till: 30, decimals: 1, units: ul}
    acidVolume=calculate: "30ul - baseVolume"
    pH=calculate:
      expression: "(acidPH * acidVolume + basePH * baseVolume) / (30ul)"
      decimals: 2
    replicate*: 3
    order=range: {order: shuffle}
    well=allocateWells: {rows: 16, columns: 24, orderBy: order}
  select: [gfp, buffer, pH, replicate, acid, base, acidVolume, baseVolume, order, well]
```

Figure F.1: Design table that generated Table 3.1

Appendix G

Quality Control of Evaporation

For some long-running experiment, evaporation can become an important issue. The major factors we considered were:

- the labware model
- the position of the labware on the lab bench
- the position of the well on the labware
- the volume within the well

The labware model has an impact, because different well shapes and sizes will expose different amounts of the liquid surface to evaporation. The position of the labware on the bench has an effect, because of varying airflow rates and temperatures. The evaporation rate also depends on the position of the well on the labware, with the more exposed wells on the border experiencing higher evaporation rates than the inner wells. The well volume also matters, with higher volumes leading to higher evaporation rates as more of the liquid surface is subject to airflow.

G.1 Results

Evaporation rates over wells filled to 150 µL are shown in Figure G.1A. We see the commonly reported phenomenon that the wells on the border evaporate more quickly than those in the middle – however, based on small size of the differences (2.72 vs 3.16 $\mu L\,h^{-1}$), we conclude that the differences can be ignored during analysis and design.

The average evapoartion rates at different bench locations are shown in Figure G.1B for our seven primary pipetting sites with the wells filled to 300 µL. The evaporation rates in Panel B are higher than in Panel A, because the wells are fuller (we found that evaporation rates increase with larger well volumes). The higher evaporation rates at the back of the bench ($y = 3$) can be explained by a nearby opening in the robot enclosure which leads to greater airflow there.

G.2 Methods

This protocol has two parts. The first part finds the evaporation rates over plate wells. It fills the wells with dye and leaves them to evaporate for a multiple hours — in these experiments, I allowed for evaporation times between 2 and 16 hours. Since evaporation removes water from the well but leaves the dye particles, the dye concentration increases proportionally. We measured the total evaporation volume by using the scale, while the change in concentration in individual wells was measured by extracting aliquots to a new plate and measuring their absorbance (Figure G.1).

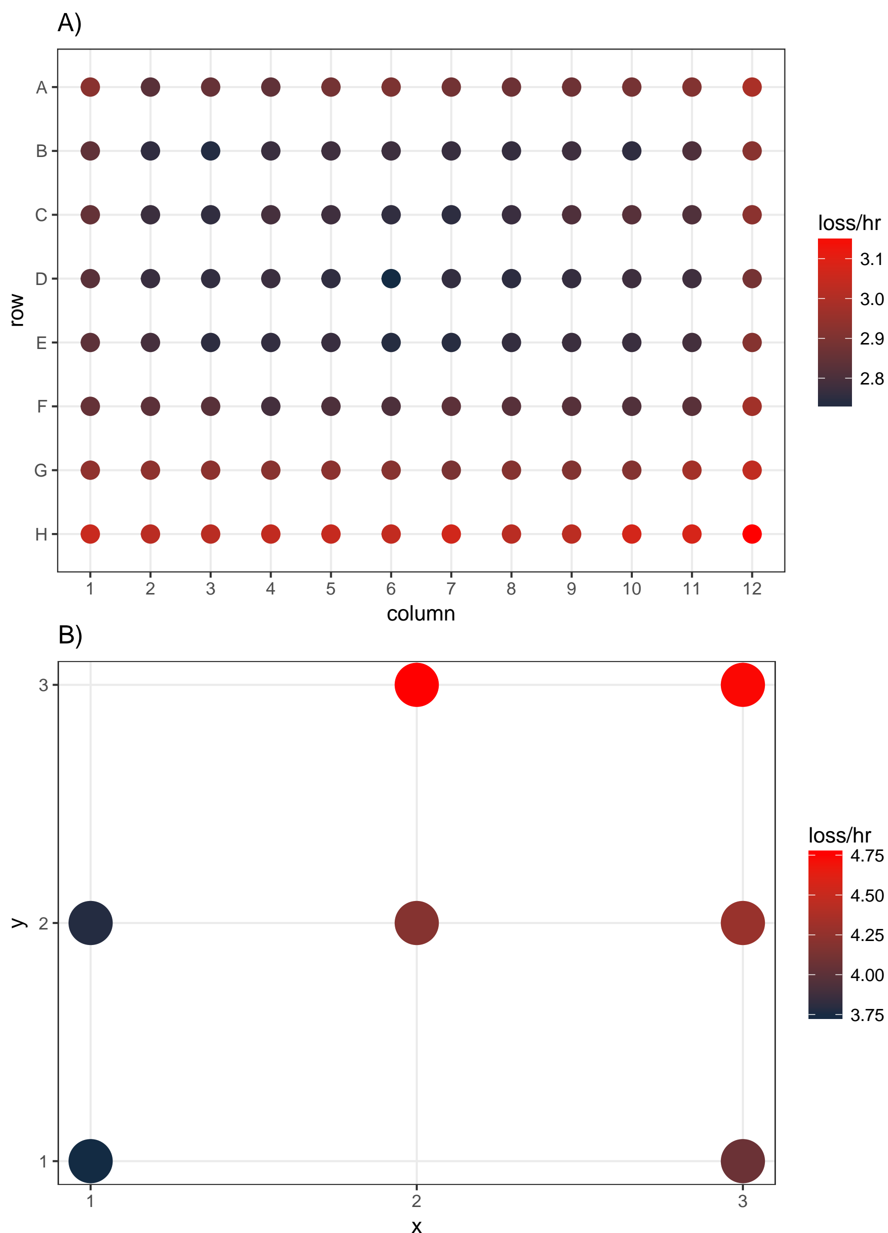


Figure G.1: **Evaporation rates over wells and sites**. **A)** The graph represents a plate, with columns on the x-axis and rows on the y-axis. Each dot represents a well, and the color represents the rate of evaporation in $\mu L\,h^{-1}$, with red being the maximum rate, and black the minimum. The maximum evaporation rate is $3.16\,\mu L\,h^{-1}$ in well H12 in the bottom right corner, and the minimum is $2.72\,\mu L\,h^{-1}$ in the middle. **B)** Evaporation rates of various bench locations for wells filled to $300\,\mu L$. The bench layout has six primary pipetting sites, which are represented as dots. The x and y axes correspond to x and y positions on the bench. The color indicates the rate of evaporation, with bright red being $4.75\,\mu L\,h^{-1}$, and black being $3.75\,\mu L\,h^{-1}$.

The protocol uses 2 transparent 96-well reader plates, one trough of dye0150, the system water source, and the absorbance reader. It finds the evaporation rate over wells on a given plate model at a given bench location. The protocol steps are:

1. On plate1, fill these wells with 250ul dye: A01 down block H02 + A03 down block B10 + G03 down block H10 + A11 down block H12 + D06 down block E07
2. On plate1, fill the remaining 24 wells with 250ul water
3. Wait for 2 hours for evaporation to take place
4. Transfer 125ul from all dye wells on plate1 to the same wells on plate2
5. On plate2, fill 8 wells with 250ul water
6. On plate2, fill 8 wells with 250ul dye
7. On plate2, transfer 125ul from the previous wells to 8 more empty wells
8. Top off all wells on both plates to 250ul
9. Measure absorbance of both plates

The electronic supplement has folders named `qc06-evaporation1` and `qc06-evaporation2` for these experiments that contains the Roboliq protocols, the R reports, and the measurements.

Bibliography

[1] Kitano, H., Systems biology: A brief overview. *Science* 2002, *295*, 1662–1664.

[2] Szallasi, Z., Stelling, J., Periwal, V., System modeling in cellular biology. *From Concepts to* 2006.

[3] Alon, U., An introduction to systems biology: Design principles of biological circuits, CRC press, 2006.

[4] Wilkinson, D.J., Stochastic modelling for systems biology, CRC press, 2011.

[5] Ideker, T., Galitski, T., Hood, L., A new approach to decoding life: Systems biology. *Annual Review of Genomics and Human Genetics* 2001, *2*, 343–372.

[6] Hood, L., Heath, J.R., Phelps, M.E., Lin, B., Systems Biology and New Technologies Enable Predictive and Preventative Medicine. *Science* 2004, *306*, 640–643.

[7] Querec, T.D., Akondy, R.S., Lee, E.K., Cao, W., et al., Systems biology approach predicts immunogenicity of the yellow fever vaccine in humans. *Nature Immunology* 2009, *10*, 116–125.

[8] Lorenzo, V. de, Systems biology approaches to bioremediation. *Current Opinion in Biotechnology* 2008, *19*, 579–589.

[9] Andrianantoandro, E., Basu, S., Karig, D.K., Weiss, R., Synthetic biology: New engineering rules for an emerging discipline. *Molecular Systems Biology* 2006, *2*.

[10] Purnick, P.E., Weiss, R., The second wave of synthetic biology: From modules to systems. *Nature Reviews Molecular Cell Biology* 2009, *10*, 410–422.

[11] Khalil, A.S., Collins, J.J., Synthetic biology: Applications come of age. *Nature Reviews. Genetics* 2010, *11*, 367.

[12] Sprinzak, D., Elowitz, M.B., Reconstruction of genetic circuits. *Nature* 2005, *438*, 443.

[13] Endy, D., Foundations for engineering biology. *Nature* 2005, *438*, 449.

[14] Linshiz, G., Stawski, N., Poust, S., Bi, C., et al., PaR-PaR Laboratory Automation Platform. *ACS Synthetic Biology* 2013, *2*, 216–222.

[15] Check Hayden, E., The automated lab. *Nature* 2014, *516*, 131–132.

[16] Armbruster, D.A., Hawes, L.C., Winter, C.T., Accuracy and precision of a robotic sample processor. *Journal of the International Federation of Clinical Chemistry* 1992, *4*, 166–173.

[17] Jung, H.R., Sylvänne, T., Koistinen, K.M., Tarasov, K., et al., High throughput quantitative molecular lipidomics. *Biochimica et Biophysica Acta (BBA) - Molecular and Cell Biology of Lipids* 2011, *1811*, 925–934.

[18] Bourbeau, P.P., Ledeboer, N.A., Automation in Clinical Microbiology. *Journal of Clinical Microbiology* 2013, *51*, 1658–1665.

[19] King, R.D., Rowland, J., Oliver, S.G., Young, M., et al., The Automation of Science. *Science* 2009, *324*,

85–89.

[20] Blow, N., Lab automation: Tales along the road to automation. *Nature Methods* 2008, *5*, 109–112.

[21] Chao, R., Liang, J., Tasan, I., Si, T., et al., Fully automated one-step synthesis of single-transcript talen pairs using a biological foundry. *ACS Synthetic Biology* 2017, *6*, 678–685.

[22] Si, T., Chao, R., Min, Y., Wu, Y., et al., Automated multiplex genome-scale engineering in yeast. *Nature Communications* 2017, *8*, 15187.

[23] Eisenstein, M., Living factories of the future. *Nature* 2016, *531*, 401–403.

[24] Gill, R.T., Halweg-Edwards, A.L., Clauset, A., Way, S.F., Synthesis aided design: The biological design-build-test engineering paradigm? *Biotechnology and Bioengineering* 2016, *113*, 7–10.

[25] Chao, R., Mishra, S., Si, T., Zhao, H., Engineering biological systems using automated biofoundries. *Metabolic Engineering* 2017, *42*, 98–108.

[26] Bates, M., Berliner, A.J., Lachoff, J., Jaschke, P.R., et al., Wet lab accelerator: A web-based application democratizing laboratory automation for synthetic biology. *ACS Synthetic Biology* 2017, *6*, 167–171.

[27] Gupta, V., Irimia, J., Pau, I., Rodríguez-Patón, A., BioBlocks: Programming protocols in biology made easier. *ACS Synthetic Biology* 2017.

[28] Beal, J., Adler, A., Yaman, F., Managing bioengineering complexity with ai techniques. *Bio Systems* 2016, *148*, 40–46.

[29] Sadowski, M.I., Grant, C., Fell, T.S., Harnessing qbd, programming languages, and automation for reproducible biology. *Trends in Biotechnology* 2016, *34*, 214–227.

[30] Taylor, C.F., Field, D., Sansone, S.-A., Aerts, J., et al., Promoting coherent minimum reporting guidelines for biological and biomedical investigations: The MIBBI project. *Nature Biotechnology* 2008, *26*, 889–896.

[31] Sansone, S.-A., Rocca-Serra, P., Field, D., Maguire, E., et al., Toward interoperable bioscience data. *Nature Genetics* 2012, *44*, 121–126.

[32] Jones, A.R., Miller, M., Aebersold, R., Apweiler, R., et al., The Functional Genomics Experiment model (FuGE): An extensible framework for standards in functional genomics. *Nature Biotechnology* 2007, *25*, 1127–1133.

[33] Soldatova, L.N., Nadis, D., King, R.D., Basu, P.S., et al., EXACT2: The semantics of biomedical protocols. *BMC Bioinformatics* 2014, *15*, S5.

[34] Ananthanarayanan, V., Thies, W., Biocoder: A programming language for standardizing and automating biology protocols. *Journal of Biological Engineering* 2010, *4*, 13.

[35] Synthace Ltd., Antha programming language. 2017.

[36] Hillson, N.J., Rosengarten, R.D., Keasling, J.D., J5 DNA Assembly Design Automation Software. *ACS Synthetic Biology* 2012, *1*, 14–21.

[37] Leguia, M., Brophy, J., Densmore, D., Anderson, J.C., Automated assembly of standard biological parts, in: *Methods Enzymol*, 2011, pp. 363–397.

[38] Yehezkel, T., Nagar, S., Mackrants, D., Marx, Z., et al., Computer-aided high-throughput cloning of bacteria in liquid medium. *BioTechniques* 2011, *50*, 124–127.

[39] Linshiz, G., Stawski, N., Goyal, G., Bi, C., et al., PR-PR: Cross-Platform Laboratory Automation System. *ACS Synthetic Biology* 2014.

[40] Smith, D.E., Frank, J., Jónsson, A.K., Bridging the gap between planning and scheduling. *The Knowledge Engineering Review* 2000, *15*, 47–83.

[41] Niiniluoto, I., Scientific Progress, in: Zalta, E.N. (Ed.), *The Stanford Encyclopedia of Philosophy*, Summer

2015, Metaphysics Research Lab, Stanford University, 2015.

[42] Radder, H., The philosophy of scientific experimentation, University of Pittsburgh Pre, 2003.

[43] Klahr, D., Fay, A.L., Dunbar, K., Heuristics for scientific experimentation: A developmental study. *Cognitive Psychology* 1993, *25*, 111–146.

[44] Quinn, G.P., Keough, M.J., Experimental design and data analysis for biologists, Cambridge University Press, 2002.

[45] Thimbleby, H., Better programming. 2004.

[46] Fomel, S., Claerbout, J.F., Reproducible research. *Computing in Science & Engineering* 2009, *11*, 5–7.

[47] Donoho, D.L., An invitation to reproducible computational research. *Biostatistics* 2010, *11*, 385–388.

[48] Peng, R.D., Reproducible Research in Computational Science. *Science* 2011, *334*, 1226–1227.

[49] Hanson, B., Sugden, A., Alberts, B., Making Data Maximally Available. *Science* 2011, *331*, 649–649.

[50] Russell, J.F., If a job is worth doing, it is worth doing twice. *Nature* 2013, *496*, 7–7.

[51] Sandve, G.K., Nekrutenko, A., Taylor, J., Hovig, E., Ten Simple Rules for Reproducible Computational Research. *PLoS Computational Biology* 2013, *9*, e1003285.

[52] Stodden, V., Resolving Irreproducibility in Empirical and Computational Research « IMS Bulletin. 2013.

[53] Ioannidis, J.P.A., Why Most Published Research Findings Are False. *PLoS Med* 2005, *2*, e124.

[54] Wickham, H., Tidy Data. *Journal of Statistical Software* 2014, *59*.

[55] Lynch, M., Protocols, practices, and the reproduction of technique in molecular biology*. *The British Journal of Sociology* 2002, *53*, 203–220.

[56] Soldatova, L.N., Aubrey, W., King, R.D., Clare, A., The EXACT description of biomedical protocols. *Bioinformatics* 2008, *24*, i295–i303.

[57] Taylor, G., An Automatic Block-Setting Crane. *The Meccano Magazine* 1937, *23*, 172.

[58] Devol, J.G.C., Programmed article transfer, US2988237 A, 1961.

[59] Boyd, J., Robotic Laboratory Automation. *Science* 2002, *295*, 517–518.

[60] Pfeiffer, R., Bongard, J.C., How the body shapes the way we think, MIT Press Cambridge, 2006.

[61] Brooks, R.A., others, Intelligence without reason. *Artificial Intelligence: Critical Concepts* 1991, *3*, 107–63.

[62] Tonietti, G., Schiavi, R., Bicchi, A., Design and Control of a Variable Stiffness Actuator for Safe and Fast Physical Human/Robot Interaction, in: *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*, 2005, pp. 526–531.

[63] Wojtczyk, M., Panin, G., Röder, T., Lenz, C., et al., Teaching and implementing autonomous robotic lab walkthroughs in a biotech laboratory through model-based visual tracking., in: *Intelligent Robots and Computer Vision XXVII: Algorithms and Techniques*, 2010.

[64] Bray, T., The JavaScript Object Notation (JSON) Data Interchange Format. 2014.

[65] Ben-Kiki, O., Evans, C., Ingerson, B., YAML Ain't Markup Language (YAML™) Version 1.1. *Working*

Draft 2008-05 2009, *11*.

[66] Nau, D.S., Au, T.-C., Ilghami, O., Kuter, U., et al., SHOP2: An HTN planning system. 2003.

[67] Sack, W., A JavaScript-based HTN Planner. 2010.

[68] Erol, K., Hierarchical task network planning: Formalization, analysis, and implementation. 1996.

[69] Wickham, H., Francois, R., Dplyr: A grammar of data manipulation. *R Package Version 0.4* 2015, *1*, 20.

[70] Join (SQL). *Wikipedia* 2017.

[71] Dasu, T., Johnson, T., Exploratory data mining and data cleaning, John Wiley & Sons, 2003.

[72] Terrizzano, I.G., Schwarz, P.M., Roth, M., Colino, J.E., Data Wrangling: The Challenging Yourney from the Wild to the Lake., in: *CIDR*, 2015.

[73] Appleton, B., Berczuk, S., Cabrera, R., Orenstein, R., Streamed lines: Branching patterns for parallel software development, in: *Proceedings of PloP*, 1998.

[74] Loeliger, J., McCullough, M., Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development, "O'Reilly Media, Inc.", 2012.

[75] Hernández, M.A., Stolfo, S.J., The merge/purge problem for large databases, in: *ACM Sigmod Record*, ACM, 1995, pp. 127–138.

[76] Mimram, S., Di Giusto, C., A Categorical Theory of Patches. *Electronic Notes in Theoretical Computer Science* 2013, *298*, 283–307.

[77] Roundy, D., Darcs - Theory. 2017.

[78] Pijul - Model. 2017.

[79] Hunt, J.W., MacIlroy, M.D., An algorithm for differential file comparison, Bell Laboratories Murray Hill, 1976.

[80] Nottingham, M., Bryan, P., JavaScript Object Notation (JSON) Patch. 2013.

[81] Ghallab, M., Nau, D., Traverso, P., Automated Planning: Theory and practice, Elsevier, 2004.

[82] Tate, A., Project planning using a hierarchic non-linear planner, Department of Artificial Intelligence, University of Edinburgh, 1976.

[83] Hart, P.E., Nilsson, N.J., Raphael, B., A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* 1968, *4*, 100–107.

[84] Andrews, H., Wright, A., JSON Schema: A Media Type for Describing JSON Documents. 2017.

[85] Whitehead, E., Roboliq: Github repository. 2017.

[86] Whitehead, E., Roboliq: User Manual, 2017.

[87] Whitehead, E., Roboliq: Commands & Types. 2017.

[88] Whitehead, E., Roboliq: Programmer Documentation. 2017.

[89] Jiang, H., Ouyang, Z., Zeng, J., Yuan, L., et al., A user-friendly robotic sample preparation program for fully automated biological sample pipetting and dilution to benefit the regulated bioanalysis. *Journal of Laboratory Automation* 2012, *17*, 211–221.

[90] Kong, F., Yuan, L., Zheng, Y.F., Chen, W., Automatic Liquid Handling for Life Science A Critical Review of the Current State of the Art. *Journal of Laboratory Automation* 2012, *17*, 169–185.

[91] Hawkins, R.C., Laboratory Turnaround Time. *The Clinical Biochemist Reviews* 2007, *28*, 179–194.

[92] Murray, P.R., Laboratory automation: Efficiency and turnaround times. *Microbiology Australia* 2014, *35*,

49–51.

[93] Valeur, B., Berberan-Santos, M.N., Molecular fluorescence: Principles and applications, John Wiley & Sons, 2012.

[94] Jameson, D.M., Introduction to fluorescence, Taylor & Francis, 2014.

[95] Zimmer, M., Green fluorescent protein (GFP): Applications, structure, and related photophysical behavior. *Chemical Reviews* 2002, *102*, 759–782.

[96] Yang, F., The molecular structure of green fluorescent protein, PhD thesis, Rice University, 1997.

[97] Remington, S.J., Green fluorescent protein: A perspective. *Protein Science* 2011, *20*, 1509–1519.

[98] Shaner, N.C., Patterson, G.H., Davidson, M.W., Advances in fluorescent protein technology. *Journal of Cell Science* 2007, *120*, 4247–4260.

[99] Roberts, T.M., Rudolf, F., Meyer, A., Pellaux, R., et al., Identification and characterisation of a pH-stable gfp. *Scientific Reports* 2016, *6*, 28166.

[100] Pédelacq, J.-D., Cabantous, S., Tran, T., Terwilliger, T.C., et al., Engineering and characterization of a superfolder green fluorescent protein. *Nature Biotechnology* 2006, *24*, 79–88.

[101] Cranfill, P.J., Sell, B.R., Baird, M.A., Allen, J.R., et al., Quantitative assessment of fluorescent proteins. *Nature Methods* 2016, *13*, 557–562.

[102] Campbell, R.E., Tour, O., Palmer, A.E., Steinbach, P.A., et al., A monomeric red fluorescent protein. *Proceedings of the National Academy of Sciences* 2002, *99*, 7877–7882.

[103] Myers, R.J., One-hundred years of pH. *Journal of Chemical Education* 2009, *87*, 30–32.

[104] Boron, W.F., Boulpaep, E.L., Medical Physiology, 2e Updated Edition E-Book: With STUDENT CONSULT Online Access, Elsevier Health Sciences, 2012.

[105] Alberts, B., Johnson, A., Lewis, J., Raff, M., et al., The Shape and Structure of Proteins. 2002.

[106] Pace, C.N., Determination and analysis of urea and guanidine hydrochloride denaturation curves. 1986, *131*.

[107] Huang, J.-r., Craggs, T.D., Christodoulou, J., Jackson, S.E., Stable Intermediate States and High Energy Barriers in the Unfolding of GFP. *Journal of Molecular Biology* 2007, *370*, 356–371.

[108] Andrews, B.T., Schoenfish, A.R., Roy, M., Waldo, G., et al., The rough energy landscape of superfolder gfp is linked to the chromophore. *Journal of Molecular Biology* 2007, *373*, 476–490.

[109] Samarkina, O.N., Popova, A.G., Gvozdik, E.Y., Chkalina, A.V., et al., Universal and rapid method for purification of gfp-like proteins by the ethanol extraction. *Protein Expression and Purification* 2009, *65*, 108–113.

[110] Davidian, M., Giltinan, D.M., Nonlinear models for repeated measurement data: An overview and update. *Journal of Agricultural, Biological, and Environmental Statistics* 2003, *8*, 387–419.

[111] Frauenfelder, H., Sligar, S.G., Wolynes, P.G., The energy landscapes and motions of proteins. *Science* 1991, *254*, 1598–1603.

[112] Noordhoek, G.T., Embden, J.D. van, Kolk, A.H., Reliability of nucleic acid amplification for detection of Mycobacterium tuberculosis: An international collaborative quality control study among 30 laboratories. *Journal of Clinical Microbiology* 1996, *34*, 2522–2525.

[113] Dequeker, E., Cassiman, J.-J., Genetic testing and quality control in diagnostic laboratories. *Nature Genetics* 2000, *25*, 259–260.

[114] Taylor, J.K., What is Quality Assurance?, in: *Quality Assurance for Environmental Measurements*,

ASTM International, 1985.

[115] Thompson, M., Wood, R., Harmonized guidelines for internal quality control in analytical chemistry laboratories (Technical Report). *Pure and Applied Chemistry* 1995, *67*, 649–666.

[116] Taguchi, G., Introduction to quality engineering: Designing quality into products and processes, 1986.

[117] Consortium, M., others, The MicroArray Quality Control (MAQC) project shows inter-and intraplatform reproducibility of gene expression measurements. *Nature Biotechnology* 2006, *24*, 1151.

[118] Hastings, W.K., Monte Carlo sampling methods using Markov chains and their applications. *Biometrika* 1970, *57*, 97–109.

[119] Gilks, W.R., Richardson, S., Spiegelhalter, D., Markov chain Monte Carlo in practice, CRC press, 1995.

[120] Betancourt, M.J., Girolami, M., Hamiltonian Monte Carlo for Hierarchical Models. *ArXiv:1312.0906 [Stat]* 2013.

[121] Carpenter, B., Gelman, A., Hoffman, M.D., Lee, D., et al., Stan: A Probabilistic Programming Language. *Journal of Statistical Software* 2017, *76*.

[122] Guo, J., Lee, D., Sakrejda, K., Gabry, J., et al., Rstan: R Interface to Stan. *R Package Version* 2016, *2*, 0–3.

[123] Bessemans, L., Jully, V., Raikem, C. de, Albanese, M., et al., Automated Gravimetric Calibration to Optimize the Accuracy and Precision of TECAN Freedom EVO Liquid Handler. *Journal of Laboratory Automation* 2016, 2211068216632349.

[124] Giardine, B., Riemer, C., Hardison, R.C., Burhans, R., et al., Galaxy: A platform for interactive large-scale genome analysis. *Genome Research* 2005, *15*, 1451–1455.

[125] Brinkman, R.R., Courtot, M., Derom, D., Fostel, J.M., et al., Modeling biomedical experimental processes with OBI. *Journal of Biomedical Semantics* 2010, *1*, S7.

[126] Takátsy, G., A new method for the preparation of serial dilutions in a quick and accurate way. *Kiserletes Orvostudomany* 1950, *2*, 393.

[127] Bär, H., Hochstrasser, R., Papenfuß, B., SiLA Basic Standards for Rapid Integration in Laboratory Automation. *Journal of Laboratory Automation* 2012, *17*, 86–95.

[128] Ford, B., Parsing expression grammars: A recognition-based syntactic foundation, in: *ACM SIGPLAN Notices*, ACM, 2004, pp. 111–122.

[129] PEG.js – Parser Generator for JavaScript. 2017.

[130] Whitehead, E., Roboliq: Evoware Commands & Types. 2017.

[131] Folk, M., Heber, G., Koziol, Q., Pourmal, E., et al., An overview of the HDF5 technology suite and its applications, in: *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, ACM, 2011, pp. 36–47.