Introduction to LabBench

Kristian Hennings

LabBench 4.0

Contents

1 Introduction **3**

1.1 LabBench and the LabBench Language 3

1.2 Why we created LabBench 3

1.3 What can LabBench do? 5

1.4 Who is LabBench intended for? 5

1.5 What do I need to know to use LabBench? 6

2 Overview of how to run a study **7**

3 Language essentials **7**

3.1 Syntax 9

3.2 Development tools 10

3.3 Visual Studio Code 10

4 Example protocol **11**

5 Tests **12**

5.1 Structure 13

5.2 Dependencies 14

5.3 Exclusions 14

5.4 Events 14

5.5 Properties 17

6 Localization **19**

6.1 Assets 20

7 Scripting **22**

7.1 Attributes 22

7.2 Simple types 22

7.2.1 Integers 22

7.2.2 Floating numbers 22

7.2.3 Text 22

7.2.4 Boolean 22

7.2.5 Enumerations 23

7.3 Data structures 23

7.3.1 Arrays 23

7.3.2 Dictionaries 23

7.3.3 Structures 23

7.4 Calculated attributes 23

7.5 Dynamic text attributes 26

7.6 Defines 27

7.7 Scripts 28

7.7.1 Calling a script 28

7.7.2 Defining functions 28

7.7.3 Test context 29

8 Experimental setups **31**

8.1 Instruments 31

8.2 Devices 32

8.3 Device assignments 34

9 Sessions **36**

9.1 Session identifiers 37

9.2 Post-session actions 38

10 Afterword **40**

1 Introduction

Welcome to the introduction to LabBench, a system for describing experimental protocols in a human and machine-readable format that can be used to reduce the complexity of running experiments. In this guide, we will take you through the concepts necessary for describing and running experiments, which we will do by constructing an experiment that can be used to study the relationship between pain sensitivity and depression, anxiety, and stress.

1.1 LabBench and the LabBench Language

LabBench is a complex tool that is difficult to describe in a few words. One way to introduce LabBench is with the concept that LabBench is two things that work together: 1) the LabBench Language and 2) the LabBench Software.

The LabBench Language is designed to specify protocols with terminology and concepts that scientists are already familiar with when they describe experiments with more conventional tools, such as writing an experimental protocol in a text document.

The LabBench Software consists of two programs, the LabBench Designer and LabBench Runner. The LabBench Designer is used to set up and manage experiments and export data for analysis. The LabBench Runner is used to run experiments written in the LabBench Language.

It is also helpful to understand what LabBench is not. LabBench is not a system for analysing data. Today, there is excellent software for the analysis of scientific data. Python, R, MATLAB, and SPSS are non-exclusive choices for scientific data analysis. These are all excellent choices for data analysis, and there is no need for another tool. Consequently, LabBench focuses exclusively on running experiments and ensuring that data is in an efficient format for later analysis in other 3rd party analysis software.

1.2 Why we created LabBench

LabBench is motivated by the ever-increasing complexity of research methods and their dissemination in research papers. A research paper is highly efficient at disseminating context, hypotheses, and discussion of research results but less efficient at disseminating research methods. Over the years, we have reproduced numerous studies from the literature. However, despite the very best efforts of everyone involved in their publication, we found that it is not uncommon for descriptions of research methods to be accidentally incomplete in manuscripts. A second problem comes from the complexity of contemporary research methods. It is not uncommon for study protocols to require numerous complex steps to

be taken by the scientist during an experiment. Steps include recording data, performing analysis/interpretation of these results during the experiment, and using these results for subsequent steps in the experiment. All of these tasks take a considerable amount of attention from the researcher.

Both problems can be alleviated by executing studies with a computer system, as the study must then be described in a machine-readable and executable format. This ensures that all parameters and methods are wholly specified; otherwise, a computer cannot execute the study. Furthermore, as the computer must execute the study, it must have automatic control over all equipment. Thus, all manual steps that do not require the interpretation or control of the researcher can be automated. This significantly reduces the complexity of running the study for the researcher.

Technically, all the building blocks for creating self-contained and automated protocols are available from the field of computer science. The most essential building block is a suitable scripting language. Today, open, powerful, and relatively easy-to-learn scripting languages, such as Python, can be used for scripts to encode protocols into machine-executable and human-understandable formats. However, the strength of general-purpose scripting/programming languages is also their weaknesses. Because they are general-purpose languages, they are powerful, and basically, there is nothing you cannot create with them; with sufficient effort, you can completely automate an experiment. As general-purpose languages, they place no restrictions on what they can be used for; however, simultaneously, they do not assist in implementing any task. This leaves the architecture and implementation details entirely up to the researcher. In practice, this means that even though the language itself may be relatively easy to learn, a very steep learning curve must be overcome before implementing even the simplest experimental protocol. Another problem is disseminating these protocols so that other researchers may use them. Without standards for their implementation, the protocols are unique to that specific experiment. This makes it challenging for other researchers to replicate experiments and extend them.

The problems inherent in general-purpose scripting languages can be alleviated by designing a domain-specific language (DSL) for protocols that can be extended with a general-purpose scripting language. A DSL is a language intended to capture expert knowledge of one specific domain or problem in a form that significantly simplifies using the language compared to a general-purpose language. In this case, the purpose of the DSL language is to describe protocols in a standard way that both humans and machines easily understand. Yet, in a format that is also flexible and extendable enough that most protocols can be described in this DSL language.

To fulfil the above aims, we have created a DSL called the LabBench Language for protocols, particularly neuroscience protocols and an associated protocol runner

called LabBench for executing these protocols.

1.3 What can LabBench do?

LabBench can do many things on different levels, from simple experiments using preprogrammed blocks of procedures that can be set up in minutes to highly complex protocols with advanced scripting and logic.

However, regardless of how you use LabBench, you should expect to:

1. Reduce the researcher's workload by moving as much of the decision-making from the experimental session to the study's planning phase.
2. Reduce the technical competencies required to take advantage of scripted protocols by eliminating or drastically reducing the amount and complexity of code required to implement automatic scripting of an experiment.
3. Ensure all data is recorded consistently by removing the save button from the program. Data are automatically saved according to the protocol for the study.
4. Ensure that detailed logs are created throughout the experiment by automatically logging every program action during a session. Events outside of the program can be added to the log by the researcher.
5. Facilitate data analysis by automatically recording all data in a machine-readable format.
6. Ameliorate dissemination and reproducibility by describing protocols in a simultaneous human and machine-readable format that includes the structure, execution, and parameters of the study.

1.4 Who is LabBench intended for?

To put it as succinctly as possible, LabBench is intended for any scientist who wants a way to make it easier to run experiments to reduce the change of errors and increase the reproducibility of their experiments. It is our experience that you do not need a background in technical sciences, such as computer science, to use and benefit from LabBench.

LabBench takes an entirely different approach than most contemporary software for non-technical professionals. No code platforms are everywhere, often accompanied by the statement "no programming skills are necessary", which is considered a quality statement. We will argue that it is the opposite of a quality statement. All no code software does is provide a graphical representation of code and one that

provides an additional layer of potential errors and misinterpretations of what the computer will do. It does not remove code but transforms it into another language you must learn. A language that has not matured from continuous improvement and experience stretching back 180 years to the very first programming language by Ada Lovelace in 1833.

It is our experience that a DSL, like the LabBench Language, is easier to learn than a visual "no code" programming environment and that it reduces the complexity of conventional programming to a level where anyone can reach sufficient proficiency in the same period as it takes to become truly proficient in a graphical "no code" programming environment.

Therefore, we believe that LabBench is for every researcher regardless of background.

1.5 What do I need to know to use LabBench?

That depends on your ambitions and needs, but LabBench is designed so you can progress in stages and that you will be able to run relevant research studies at each stage of proficiency.

In the first stage, a LabBench Protocol Repository is available through the LabBench Designer and on GitHub. You can install ready-made protocols from this repository, for example, quantitative sensory testing, perception threshold tracking, psychophysical research tasks such as the Stroop, Flanker, or Stop-Signal tasks, and similar research protocols. This will allow you to see what LabBench can do without having to write any code, and if your need is to run standardised protocols reproduced from published papers, then that may be all you will need to use and benefit from LabBench.

The next stage is to develop your protocols using the preprogrammed research methods with LabBench. You will need to use and understand the LabBench Language at that stage. Using the LabBench Language consists of writing a text configuration file for the experiment in a format known as eXtensible Markup Language (XML). You will also be introduced to the first simple scripting at this stage. Part of the LabBench Language is that tests can be configured with the results of previous tests. This can be done by writing mathematical formulae of the same form as on a calculator in the text configuration file. Accessing the LabBench Protocol Repository on GitHub can also make this stage significantly more accessible, as it is easier to assemble a protocol from parts of existing protocols than to write a protocol from scratch.

Going further, preprogrammed research protocols that can be directly configured by the LabBench Language configuration file can be extended by writing Python scripts that are called from the configuration code. With this approach, you can

implement truly advanced protocols. However, you will need to learn a bit of Python programming. We chose Python as the scripting language for LabBench as it is an easy-to-learn language that is very mature, for which numerous excellent textbooks exist. But you do not need to be an expert Python programmer. One of the advantages of combining Python with the LabBench Language is that it drastically reduces the amount and complexity of the code required to implement an experimental paradigm.

2 Overview of how to run a study

Before examining and discussing each element of a LabBench protocol, starting with a broad overview of all required to run an experiment with LabBench is often helpful.

The key steps and concepts that are required for any experiment, regardless of their complexity and scale, are the following:

1. An Experiment Definition File in the LabBench Language must be written. Depending on the complexity of the study, this may also involve writing one or more Python scripts that are called from the Experiment Definition File. Files such as images, instructions, sound files, and similar may also be included with and used by the Experiment Definition File.
2. The Experiment Definition File and its Python scripts and assets must be placed in a protocol repository and assigned a unique ID. This ensures that protocols can be uniquely identified when results are published.
3. An experiment is created with the LabBench Designer program from an Experiment Definition File in a protocol repository.
4. The experimental protocol is performed using the LabBench Runner program.
5. When the experiment has been completed and all the sessions have been recorded, the data from the experiment is exported in the LabBench Designer program for analysis in 3rd party data analysis software.

All studies using LabBench will go through this series of steps, only differing in the complexity and scale of each step.

3 Language essentials

The minimum for running an experiment with LabBench is to write an Experiment Definition File (`*.expx`). The Experiment Definition File consists of a definition of

the experiment and its protocol. The protocol defines the experimental procedures. The protocol may contain localization of the protocol into different languages and additional files required by the protocol. Assets can be files such as images for visual stimuli, text files for instructions to the experimenter or subject, or scripts that extend the functionality of the built-in tests in LabBench. The Experiment Definition File also contains information required to run the protocol in a specific experiment. This includes information about the experimental setup, valid session IDs, export of data, etc.

The Experiment Definition File is a plain text file, which means any text editor can be used. The format for these files is based on the eXtensible Markup Language (XML), a general-purpose markup language that can be used to implement DSL.

Listing 1 provides the structure of an Experiment Definition File:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <experiment xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3              xsi:schemaLocation="http://labbench.io https://labbench.io/
   xsd/4.0.0/experiment.xsd">
4      <subject-validator regex="^S[0-9]{3}$"
5                         advice="Please enter an ID in the form of SXXX,
   where X is a digit" />
6      <experimental-setup>
7          <devices>
8              <!-- Contents omitted for brevity -->
9          </devices>
10         <device-mapping>
11             <!-- Contents omitted for brevity -->
12         </device-mapping>
13     </experimental-setup>
14     <protocol>
15         <languages>
16             <!-- Contents omitted for brevity -->
17         </languages>
18         <defines>
19             <!-- Contents omitted for brevity -->
20         </defines>
21         <tests>
22             <!-- Contents omitted for brevity -->
23         </tests>
24         <assets>
25             <!-- Contents omitted for brevity -->
26         </assets>
27     </protocol>
28     <post-actions>
29         <!-- Contents omitted for brevity -->
```

```
30     </post-actions>
31 </experiment>
```

Listing 1: Experiment Definition File.

3.1 Syntax

We will now give a brief but complete introduction to what you need to know about XML for writing Experiment Definition Files. The first line in Listing 1 identifies the file as an XML file and its encoding. All files include this line, and even though other character encodings exist today, there is little reason to use other encodings than UTF-8. As a result, the first line is invariant, and the Experiment Definition Files must always start with this line.

XML documents are built from a hierarchical structure of XML elements. All XML documents have one root element that may contain nested elements, which, in turn, can contain more nested elements in an arbitrary deep hierarchical structure. XML elements consist of a starting tag, a closing tag, and content between these tags. In Listing 1, the root element is an experiment element, with a starting tag of `<experiment>` and a closing tag of `</experiment>`.

The exception to this rule is self-closing elements. The `<subject-validator>` is an example of a self-closing tag, as it is not followed by a `</subject-validator>` tag; instead, the starting tag ends with `/>`. Self-content tags allow for a more concise and for a document that is easier to read. The use of self-closing elements is widespread in the LabBench Language.

The `<experiment>` element has other elements as its content: a `<subject-validator>`, a `<experimental-setup>`, a `<protocol>` element, and a `<post-actions>` element. Their order is significant; some of these elements are optional, but if used, they must be used in the order given in Listing 1.

XML elements can be adorned with characteristics in the form of XML attributes. In Listing 1, `regex` and `advice` are attributes of the `<subject-validator/>` element, which provides a regular expression to validate session IDs and help the experimenter if a session ID fails this validation. Attributes are in the form of `name="value of the attribute"` and are listed after the element's name and before the `>` character or `/>` character for self-closing elements.

The last XML syntax that may be useful for you are comments. Comments are text in the file that will be ignored by LabBench and can be used to add additional documentation to a protocol not supported by the language. No language, no matter how domain-specific, can hope to contain all constructs that are needed to describe its content fully. With comments, additional information such as an explanation for the rationale for the design of a protocol, references, and similar information can be added to protocols. In Listing 1, the

Figure 1: Illustration of code completion where the IDE lists valid XML elements.

text `<!-- Contents omitted for brevity -->` is a comment, and in general, LabBench will ignore everything written between the `<!--` and `-->` tags.

The above description of the XML format is all the knowledge of XML itself needed to write Experiment Definition Files. Naturally, one needs significant additional knowledge of the LabBench Language, such as the correct content of the elements, etc. However, this knowledge does not need to be rote learning. The two additional attributes, `xmlns:xsi` and `xsi:schemaLocation`, are used to leverage existing tools from computer science that make rote learning of all the valid elements and attributes in the LabBench Language unnecessary.

3.2 Development tools

For writing Experiment Definition Files, as well as Python scripts, an Integrated Development Environment (IDE) makes the process significantly easier due to code completion. Code completion means that you do not need to remember which XML elements or attributes are valid at a given place in your definition file. Instead, tap `CTRL + SPACE` (for Visual Studio Code) to get a list of valid elements or attributes Figure 1. Code completion also reduces the need for manual typing, as you can simply select the required element or attribute and then press enter, at which point the IDE will complete the code for you without requiring typing.

3.3 Visual Studio Code

To use Visual Studio Code to develop LabBench experiments, first download the installer from:

- https://code.visualstudio.com/

Then, install the program. However, Visual Studio Code does not support XML code completion without a suitable editor extension. The Red Hat XML extension can be recommended to enable XML code completion.

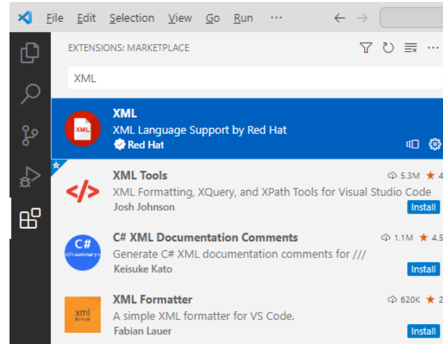

Figure 2: Red Hat XML extension

To install this extension: 1) open Visual Studio Code, 2) choose extensions by clicking the fifth icon in the leftmost pane in the editor, 3) search for XML and select the XML extension by Red Hat, and 4) click install. When the extension has been installed code completion will be available in Visual Studio Code by pressing `CTRL + SPACE`.

4 Example protocol

To study the relationship between depression, anxiety and stress and quantitative sensory testing in the form of cuff pressure algometry, we will need an experiment that quantifies the depression, anxiety, and stress of the subject with a suitable self-report instrument, followed by a protocol for cuff pressure algometry that measures pain detection and tolerance threshold, temporal summation and conditioned pain modulation.

To quantify depression, anxiety, and stress, we will use the DASS scale. The DASS scale consists of 42 items that are designed to measure the three related negative emotional states of depression, anxiety, and stress. Each item consists of a statement the subject is asked to evaluate on a 4-point Likert scale.

The Aalborg Protocol for Cuff Pressure Algometry will be used for quantitative sensory testing of pain. This protocol consists of:

1. measurement of the pain detection and tolerance threshold for the dominant leg,
2. measurement of temporal summation wherein the subject is asked to rate ten consecutive pressure stimuli,
3. measurement of the pain detection and tolerance threshold for the non-dominant leg, and finally,

4. measurement conditioned modulation by measuring the pain detection and tolerance threshold for the dominant leg while a conditioned pressure stimulus is applied to the non-dominant leg.

As we will need to change the parameters for the Cuff Pressure Algometry protocol depending on the subject's dominant side, we will also need to collect additional information in an initial step in the protocol. This initial step will consist of the experimenter recording the subjects' sex, age, dominant side, and whether cuff pressure algometry can be performed on both legs, only a single leg or not at all. If cuff pressure algometry cannot be performed or can only be partially performed, this initial step will also consist of recording the reason for the inability to perform the full cuff pressure algometry protocol.

5 Tests

LabBench tests are the basic building blocks of a protocol, where each test implements a specific type of experimental procedure that can be configured with parameters and extended by Python scripts. These tests are the central elements of the Experiment Definition File, which describes the experimental procedure that will be performed in the experiment. These procedures are written in the `<tests> </tests>` element of the Experiment Definition File, and in the LabBench Language, experimental procedures are referred to as tests.

One of the tests we will need in our protocol is the <algometry-stimulus-response> test, which will allow us to obtain a subject's visual analogue rating of a linearly increasing pressure stimulus. From these ratings, we can determine the pain detection (PDT) and tolerance (PTT) thresholds; the pain detection threshold is taken as the pressure at which the subject rated higher than the `vas-pdt` level, and the pain tolerance threshold is taken as the pressure at which the subject stopped the test by pressing a response button.

The definition of this <algometry-stimulus-response> test is shown in Listing 2. This example consists of elements/attributes common to all tests regardless of their type and attributes specific to the <algometry-stimulus-response> test. The <algometry-stimulus-response> test has only test-specific attributes. However, other types of tests also have test-specific elements. Test-specific elements must be written below the elements common for all types of tests.

```
1 <algometry-stimulus-response ID="SR01"
2                              name="Stimulus-Response (Cuff 1)"
3                              experimental-setup-id="blank"
4                              delta-pressure="DeltaPressure"
5                              pressure-limit="100"
```

```
                            primary-cuff="func: Setup.StimulatingCuff(tc
)"
                            second-cuff="false"
                            stop-mode="STOP_CRITERION_ON_BUTTON_PRESSED"
                            vas-pdt="VasPDT">
    <dependencies>
        <dependency ID="SUBJECT" />
    </dependencies>
    <condition expression="SUBJECT['CLEARING'] != 0"
               help="Subject is not cleared for pressure algometry"/>
```

Listing 2: Example of a LabBench test for determining the pain detection and tolerance thresholds to cuff pressure stimuli.

The `ID`, `name`, and `experimental-setup-id` are attributes common to all tests regardless of their type. The `ID` attribute is used to uniquely identify and reference the test result of this test from other tests in the protocol. The `name` attribute gives the test a human-readable name. This one is used in LabBench Runner during the experiment to enable researchers to identify the test in the protocol. The `experimental-setup-id` is an optional parameter, and it is only required if the experimental setup contains equipment that can be reconfigured during an experiment.

5.1 Structure

Tests have more common elements than the example in Listing 2. The attributes and elements common for all tests are listed in Listing 3.

```
<test-type ID="[Identifier of the test]"
           Name="[Human understandable name of the test]"
           experimental-setup-id="[Experiment Setup Configuration
Identifier]">
    <test-events>
        <!-- Contents omitted for brevity -->
    </test-events>
    <properties>
        <!-- Contents omitted for brevity -->
    </properties>
    <dependencies>
        <!-- Contents omitted for brevity -->
    </dependencies>
    <condition>
        <!-- Contents omitted for brevity -->
    </condition>

```

```
17     <!-- Additional test-specific elements -->
18 </test-type>
```

Listing 3: General structure of Labbench tests with the elements and attributes that are common for all tests regardless of their type..

5.2 Dependencies

The <dependencies> element is used to prevent tests from running if the test depends on results from tests that have not yet been completed. In our protocol, we will have an initial <survey> test with ID="SUBJECT" in which the researcher, in collaboration with the subject, will answer a series of questions to set up the rest of the experiment.

One of these questions is whether the subject is right or left-handed, which is used in the "primary-cuff" attribute to automatically select the cuff placed on the subject's dominant side. As a result, the test has a dependency on the survey test (¡dependency ID="SUBJECT" /¿) and cannot be executed before the survey test has been completed.

5.3 Exclusions

The second common element that the example contains is the <condition> element. This element excludes tests from a protocol if they cannot be performed.

For our protocol, we will, in the SUBJECT <survey> test, also determine whether cuff pressure algometry can be performed on both legs, only on the left or right, or not at all. If cuff pressure algometry cannot be performed, then all cuff pressure algometry tests in the protocol will be excluded, but if cuff pressure algometry can be performed on only one leg, then the tests for pain detection and tolerance threshold for the dominant side and temporal summation will be performed for the available leg, but the tests for the non-dominant side and conditional pain modulation will be excluded.

This type of logic to exclude tests in protocols can be implemented with the <condition> element that is common for all tests, and thus, exclusion logic can be implemented for all tests in a protocol regardless of their type.

5.4 Events

The <test-events> element is used to specify Python scripts that are executed when tests are selected, started, completed, or aborted. This can be used to extend tests with functionality outside the scope of what the test was originally designed for.

Test events are not used in our current protocol with cuff pressure algometry, but to illustrate their use, an example where the base functionality of a <electrophysiology-evoked-potentials> test is extended with test events is provided in Listing 4.

```
<electrophysiology-evoked-potentials ID="Cond"
                                     name="Conditioning"
                                     trigger-update-rate="20000"
                                     response-collection="none">
    <test-events start="func: Functions.Condition(tc)"
                 abort="func: Functions.Stop(tc)"
                 complete="func: Functions.Stop(tc)">
        <instrument interface="pressure-algometer"
                    name="Algometer" />
    </test-events>

    <stimulation-pattern time-base="seconds">
        <uniformly-distributed-sequence iterations="12"
                                        Toffset="5"
                                        minTperiod="10"
                                        maxTperiod="15" />
    </stimulation-pattern>

    <stimuli order="round-robin">
        <stimulus name="Stimulus"
                  count="1"
                  intensity="StimIntensity.Intensity">
            <!-- Stimulus specification omitted for brevity -->
        </stimulus>
    </stimuli>
</electrophysiology-evoked-potentials>
```

Listing 4: Example of a test where its base functionality is extended with test events.

The <electrophysiology-evoked-potentials> test in Listing 4 is intended to generate evoked potentials by presenting a set of stimuli to the subject according to a given stimulation pattern. The test is intended to work only with short electrical, auditory, visual, or tactile stimuli suitable for generating an evoked potential, which can be used, for example, EEG experiments with the oddball paradigm, electrically evoked potentials, etc.

One research group wanted to study how electrically evoked potentials are modulated by a conditioning pressure pain stimulus, which the built-in functionality of the <electrophysiology-evoked-potentials> cannot do. The built-in functionality does have any mechanism that provides a way to apply a stimulus for the entire test duration, as it is only intended for short stimuli that are applied according to

a stimulation pattern.

In their study, they wanted to use a LabBench CPAR+ device to apply a conditioning pain stimulus while electrically evoked potentials were generated with a DS7 stimulator. This was made possible with a test event that ran a script starting the conditioning stimulus when the test was started and stopping it when the test was stopped or aborted. The <test-event> in Listing 4 has three attributes `start`, `abort`, `complete` that is executed when the test is started, aborted, and completed, respectively.

These attributes can contain either a single-line Python statement or reference a function in a Python script. Starting and stopping the pressure stimuli is too complicated to achieve in a single Python code line. Instead, a Python function was called to start and stop the stimulation. Using Python functions, and scripting in general, is first discussed in the section “Scripting”; however, for completeness, the code for these functions is provided in Listing 5.

```python
from Serilog import Log
from LabBench.Interface.Instruments.Algometry import *

def Condition(tc):
    algometer = tc.Devices.Algometer
    chan = algometer.Channels[0]

    chan.SetStimulus(1, chan.CreateWaveform()
                           .Step(0.70 * tc.SR.PTT, 9.9 * 60))
    algometer.ConfigurePressureOutput(0, ChannelID.CH01)
    algometer.StartStimulation(AlgometerStopCriterion.
    STOP_CRITERION_ON_BUTTON_PRESSED, True)

    Log.Information("Starting conditioning: {intensity}", 0.07 * tc.SR.
    PTT)
    return True

def Stop(tc):
    algometer = tc.Devices.Algometer
    algometer.StopStimulation()
    return True
```

Listing 5: Python functions for conditioning electrical evoked potentials with a static pressure stimulus.

In the section “Scripting”, we will introduce in detail how Python is used for scripting protocols and to extend the functionality of LabBench. Consequently, if you have never programmed in Python, the code in Listing 5 may be new to you. Nevertheless, the example in Listing 5 may hint that it is defining

("def") two functions, `Condition(tc)` and `Stop(tc)`, that is intended to start and stop the pressure stimulus, respectively. What causes these functions to be called from the <test-event> is that, for example, the `start` attribute contains `func: Functions.Condition(tc)`. The `func:` statement is a keyword that tells LabBench that a Python function must be called, which in this case is the `Condition(tc)` function located in the Functions script.

In this case, the test event is used to condition/modulate evoked potentials. However, as test events are common to all tests, this test event could be inserted into any other LabBench test. For example, if we wanted to see how the responses to the DASS scale are influenced by a simultaneous painful stimulus, it could be inserted into the <survey> test, which we will later use for implementing the DASS scale. Furthermore, test events do not need to be used for applying a stimulus to a subject. If we need general code to run when a test starts, we can run it in the `start` test event. This could be used, for example, to initialize a random sequence of visual stimuli or similar, or if we want to add custom information to the log system, then that can also be accomplished with test events.

5.5 Properties

The last test element we are yet to discuss is the <properties> element. The <properties> element is an optional element that can be used for modifying the execution of tests. Listing 6 provides an overview of all possible test properties and their attributes.

```
<properties>
    <auto-start value="true" />
    <extended-data-collection value="true"/>
    <instructions default-instructions=""
                  instructions=""
                  start-instruction=""
                  override-results="true"/>
    <next ID=""/>
    <annotations>
        <!-- Contents omitted for brevity -->
    </annotations>
</properties>
```

Listing 6: Overview of test properties.

The <auto-start> property can be used to chain tests. LabBench will automatically select the test in the protocol when a test is completed. Setting the value attribute to `true` on the <auto-start> attribute will cause the test to start automatically when it is selected after the previous test has been completed. The effect of the <extended-data-collection> property depends on the type of test. Certain tests

support an extended collection of data that will be enabled if this property is true.

All tests support displaying information to the researcher when they are selected and not running. The <instructions> element is used to enable the showing of instructions to the researcher. If used, the `default-instruction` is mandatory and must provide the ID of the information file to display to the researcher. This instruction file is used if the test is blocked due to an unsatisfied dependency, has been excluded, or if the instructions attribute is not specified. If the `instruction` attribute is specified, then the information file specified by this attribute will be used when the test can run. The difference between the `default-instruction` and `instruction` attribute is that the latter can be scripted, while the former cannot.

In our protocol, we could use the `instruction` attribute to select different instructions to present to the experimenter depending on whether the subject is right or left-handed. The `override-results` attribute controls whether the results of a test or its instructions will be shown when the test is completed. By default, selecting a completed test will display its results; however, this behaviour can be overridden by the `override-results` attribute to true, which will always display instructions when the test has been completed. The `start-instruction` property can be used to modify the instruction given to the researcher when a test is selected and can start. By default, LabBench will display "Test is ready to start" when a test can be started, but if the `start-instruction` attribute is used, then this can be used to display a custom message. As this attribute can be scripted, this message can also depend on previous results recorded in the protocol.

The `<next>` element is included to support the logical flow of the protocol when tests have been excluded. By default, when a test is completed, LabBench will select the next test in the protocol unless the `<next>` element is used for the completed test. If the `<next>` element is used, its value must be the ID of the next test to select when the test is completed and because this property can be scripted, it is possible to select this ID at runtime depending on the previous results in the protocol. Consequently, if the results of, for example, a `<survey>` test have caused a test to be excluded, then the `<next>` element can be used to select the correct next test in the protocol instead of the test that has been excluded.

The `<annotation>` property does not influence how a test is executed. Instead, it can add information to the test that will be exported with the results. This can be used, for example, if a strength-duration curve is determined to specify the duration of the stimuli used in the test. Adding numbers, Boolean values, text strings, and a list of numbers as annotations to a test is possible. Listing 7 provides an example of all possible test annotations.

```
1 <annotations>
```

```
    <number name="ChargeBalRatio"
            value="4"/>
    <bool name="ChargeBalanced"
          value="true"/>
    <string name="string" value="This is the value of the text string"/>
    <numbers name="Ts">
        <number value="0.1"/>
        <number value="0.2"/>
        <number value="0.5"/>
        <number value="1.0"/>
    </numbers>
</annotations>
```

Listing 7: Example with all possible types of test annotations.

Annotations can also be added programmatically to test results from a function defined in a Python script. This can be used to add data from, for example, the start test event.

6 Localization

Localization refers to adapting software to suit a specific region or culture, typically involving language, cultural norms, and regional preferences. Most importantly, this involves that subject-facing text is displayed in a language that the subject understands, but it may also involve customizing other aspects of the protocol, such as formats for dates, numbers, currency, and similar culture dependent items.

Localization goes beyond mere translation and often requires a deep understanding of the subject's culture. For example, localizing questionnaires goes beyond a translation of the text in the questionnaire but also requires a validation that the translated text holds the same meaning to the subject as the original language of the questionnaire to native speakers.

Internationalization, also often abbreviated as i18n, is the process of designing protocols in a way that facilitates adapting protocols to different cultures by separating culture-dependent content from the code and structure of the protocol.

Protocols provide a mechanism for i18n in the form of the <languages> element that can be specified in protocols, for example:

```
<languages>
    <language code="en"
              name="English"/>
    <language code="da"
              name="Danish"/>
```

```
6 </languages>
```

Listing 8: Example of a languages element.

Specifying a <languages> element in a protocol creates a Language define (see section Defines) that can be used in calculated parameters to return different values depending on the language/culture selected by the experimenter when a session is started, and it will be possible to use different protocol assets depending on the selected language/culture.

LabBench does not enforce any conventions for language codes. However, it is highly recommended that language codes from the "ISO 639-1" standard be used, as there is considerable work involved in localizing, for example, questionnaires, and by using standardized codes, it is possible to reuse a questionnaire implemented for one study in other studies that require the same questionnaire.

Language codes for the ISO-639-1 standard can be found on the net by searching for "ISO 639-1 codes," which are extensively used and commonly available. Please note that in the example above, only the language code was specified for English. It is also possible to specify the culture; for example, code="en-gb" would have been the code for English as spoken in the United Kingdom.

6.1 Assets

Assets in the form of files can be included in the protocols: Python scripts, documents, images, and similar files. Assets that are included in a protocol are specified within the `<assets>` tag:

```
1 <assets>
2     <file-asset id="TEXT" file="TEXT_EN.py">
3         <language code="da" file="TEXT_DA.py"/>
4     </file-asset>
5     <file-asset id="Calculations" file="Calculations.py" />
6     <file-asset id="INSTRUCTIONS" file="Instructions.rtf" />
7     <file-asset id="Setup" file="Setup.py" />
8 </assets>
```

Listing 9: Example of assets that are included in a protocol

Each asset has two mandatory attributes: an `id` attribute, used to identify the asset within the Experiment Definition File, and a `file` attribute with the file name to use.

The assets in Listing 9 contain both localized and non-localized assets. Localized assets have additional `<language>` elements within them, which makes it possible to use different files depending on which language/culture has been selected by the experimenter. Localization of assets is why the file name is not used as the

identifier (`id`) of the asset, as if it were, then it would not have been possible to use different files depending on the selected language/culture. The way localization works with assets is that LabBench will first look up if an asset has a `<language>` element for the selected language/culture and will use that file if found. If no `<language>` element is found for the selected language, then the file specified on the `<file-asset>` element will be used and thus is the default file if no language element is found.

The LabBench Language supports the following types of assets:

- **Python Scripts (*.py):** Functions within Python scripts can be called from the Experiment Definition File to extend the base functionality of the language and to implement more complex logic and calculations in experiments.
- **Markdown Text (*.md):** Is used to format text that are displayed during an experiment.
- **Plain Text (*.txt):** Is used for text in experiments.
- **Rich Text Format (*.rtf):** Rich text format documents can be authored in Word or similar programs and can contain both text, headings, tables, images, etc. They are used for instructions that can be displayed to either the experimenter or subject.
- **Portable Network Graphics (*.png):** Is used for displays within the program or for images that are displayed to subjects. For example, they can be used for cues that are displayed to the subjects in Stroop, Flanker tasks, and similar psychophysical research paradigms.
- **Zip Files (*.zip):** Is used as a container for all other types of assets. Technically, it is possible but not recommended to have zip files within zip files.
- **Wave Files (*.wav):** Is used for auditory stimuli for, for example auditory evoked potentials.
- **Comma Separated Values (*.csv):** General data files that can be used to import data into experiments.
- **Scalable Vector Graphics (*.svg):** Can be used for visual stimuli.

7 Scripting

7.1 Attributes

Throughout this guide, we have already used attributes extensively. For example, in Listing 7, attributes provide the data for the test annotations. Each attribute has a name and is assigned a value in the form of:

`name_of_attribute="value of attribute".`

An attribute also has a type; for example, the value attribute of the number element in Listing 9 is of type float that represents rational numbers of the form 0.1, 1.2, etc. The LabBench Language has simple types for integers, floats (rational numbers), Boolean values, text strings, and value lists (enums).

7.2 Simple types

7.2.1 Integers

An integer is a whole number, meaning it is a number that does not have any fractional or decimal parts. Integers can be either positive, negative, or zero. They represent values that can be counted or measured in whole units without fractions or decimals.

`Example: 1, 2, 3, 4, -10`

7.2.2 Floating numbers

A float represents a rational number, which is a number that can be expressed as the quotient or fraction of two integers, where the numerator is an integer and the denominator is a non-zero integer.

`Example: 2.8, -1.2, 0.1`

7.2.3 Text

Text variables, often referred to as a ”string variable,” are data types used to store and manipulate sequences of characters.

`Example: 'This is text string'`

7.2.4 Boolean

Boolean variables can only hold one of two possible values: ”true” or ”false.” v
`Example: true, false (and nothing else)`

7.2.5 Enumerations

An `enum`, short for "enumeration," is a data type in computer programming that defines a set of named constant values. Enumerations create a collection of symbolic names (enumerators) that represent a finite list of related, distinct values or options.

`Example: single-sample`

7.3 Data structures

Besides the simple types in table TBD there are also types for structuring data. Data can be structured as lists (array), dictionaries (dict), and structures (struct).

7.3.1 Arrays

An array holds a list of values that are accessed by their index in the list. Each of the values in the array can be either a simple type or another structured type.

Nomenclature: `array<type>`

Examples: `intensity[2]`, `answer[3]`

7.3.2 Dictionaries

A dictionary holds a set of (key, value) pairs where elements in the set is accessed by their key, and the value can be either a simple type or another structured type. The key is always a simple type, most often a text type.

Nomenclature: `dictionary<key_type, value_type>`

Examples: `SETUP['RECT']`

7.3.3 Structures

A set of values where each value in the set is named. Each of the values in the array can be either a simple type or another structured type.

Nomenclature: `struct(name1<type>,name2<type>,...,nameN<type>)`

Examples: `SR.PTT, SR.PDT`

7.4 Calculated attributes

Most attributes are calculated, meaning they can be specified in a protocol with a mathematical expression. Calculated attributes in a protocol allow the LabBench

Language to automate manual tasks during an experiment. As attributes configure tests, it is possible to configure tests from the results of previous tests in the protocol.

The following example illustrates this. In our protocol on the relationship between DASS scores and pain sensitivity, we wish to determine the temporal summation of pressure stimuli. The intensity of the pressure stimuli must be set to 70% of the Pain Tolerance Threshold (PTT) to linearly increasing pressure, which is determined by the ¡algometry-stimulus-response¿ test with ID="SR01". To do this, we will set the pressure of the stimuli (pressure-stimulate) in the ¡algometry-temporal-summation¿ test to:

```
pressure-stimulate="0.7 * SR01.PTT"
```

We can do this because the text that is between the " " quotes of a calculated attribute is a single-line Python statement, which gets evaluated by the LabBench Runner, and when the code is specified in the Experiment Definition File, this is termed as a single-line calculated attribute, in contrast to a script calculated attribute that is evaluated by calling a Python function in a script. A second reason why calculated attributes can automate manual tasks is that from calculated parameters, we have access to a set of variables:

Variables	Name	Description
Results of tests in the protocol	The ID of the test	Results of tests in the protocol. These variables are available to all calculated attributes except attributes of defines. Examples: pressure-stimulate="0.7 * SR01.PTT" conditional-pressure="0.7 * SR02.PTT"

Instruments	ID within the test of the instrument.	Instruments that are used by the test. These variables are only available for calculated attributes of tests. Knowing their names requires looking up their names in the LabBench Language Specification [REF] to which Instruments each type of test requires and their ID within that specific type of test. Example: Imax="Stimulator.Imax"
Defines	ID of the define	Defines holds values used in multiple calculated attributes in a protocol. They are named after their ID in the list of defines. Example: delta-pressure="DeltaPressure"
Mathematical functions	exp, round, log, log10, sin, sinh, asin, cos, cosh, tan, tanh, abs, sqrt, max, min, pow	Common mathematical functions. Examples: exp(10) round(1.67) min(4 * SETUP['RECT'], Stimulator.Imax)
Free parameter	x	Only used tests for which a response is determined for a change in a free parameter. For example, the calculated attributes for a stimulus in a threshold estimation test will have access to the free parameter x. Example: Is="x" Is="-x/5"

Language	Language	If the experiment is localized with a ¡languages¿ element, then the selected language will be available as a variable named Language. Example: Language == ‘en’
Session ID	SESSION_NAME	ID of the current session. The type of the variable is text.
Session time	SESSION_TIME	Time that the current session was started. The type of the variable is text.

A short note on nomenclature. Throughout the LabBench documentation, it is necessary to specify the types of attributes. A calculated attribute will be required to return either 1) a simple type, 2) a list, dictionary, or structure of simple types, or 3) a value of any type.

The calculated attribute is specified as calculated<type>, meaning that:

- calculated<int> is a calculated parameter that must return an integer, and
- calculated<list<float>> is a calculated parameter that must return a list of floats and
- calculated<any> is a calculated attribute can return any type.

7.5 Dynamic text attributes

Text is used extensively in Experiment Definition Files and may be needed to customize the text based on either the localization of the protocol or results that have been recorded. These could be achieved with calculated attributes. However, in many cases, only literal text strings are needed. If calculated attributes were used directly for text attributes, then all text attributes would need to be specified as attribute-name=”’test’” because the value of the attribute would be a Python statement and, within Python, text is specified with single quotes (i.e.‘example text’). Using Python syntax for all the text that needs to be customized would be an inconvenience and a source of error because it would be very easy to forget the single quotes ’ within the double quotes “.

To solve this problem, the LabBench Language has a special calculated attribute type called dynamic text, which allows literal text to be specified as:

```
instruction="What is the reason for skipping the pressure tests?"
```

But if a dynamic: keyword is added before the text, then it will be evaluated as a calculated¡text¿ attribute:

```
instruction="dynamic: Text['I01']"
```

Which, in the above example, returns the text that is stored in the dictionary Text under the key 'I01'.

7.6 Defines

Often, you will need to use the same value for a parameter for multiple attributes in an Experiment Definition File. When the same values are repeated in multiple places, it is a common error that this value will be wrong in one or more places. A common principle in programming is the "Don't Repeat Yourself" (DRY) principle. The DRY principle states that if you need the same information at multiple sites, a mechanism must be available to avoid that.

To avoid this problem, the LabBench Language allows you to define variables in a protocol that can be used in subsequent definitions, tests, and post-session actions.

Each define has two attributes: 1) a `name` attribute that is the name the variable can be referred to in calculated attributes and scripts, and 2) its `value`. Listing 10 provides an example of how three variables can be defined:

```
1 <defines>
2     <define name="DeltaPressure" value="1"/>
3     <define name="VasPDT" value="0.1"/>
4     <define name="Text" value="func: TEXT.CreateText(tc)"/>
5 </defines>
```

Listing 10: The defines that are used in our protocol.

The variables defined in the `<defines>` element can subsequently be used in defines, tests, and post-session actions. Below is an example of how the DeltaPressure variable is used in a <algometry-stimulus-response> test:

```
1 <algometry-stimulus-response ID="SR01"
2                              name="Stimulus-Response (Cuff 1)"
3                              experimental-setup-id="blank"
4                              delta-pressure="DeltaPressure"
5                              pressure-limit="100"
6                              primary-cuff="func: Setup.StimulatingCuff(tc
    )"
7                              second-cuff="false"
```

```
                                  stop-mode="STOP_CRITERION_ON_BUTTON_PRESSED"
                                  vas-pdt="VasPDT">
    <!-- Content omitted for brevity -->
</algometry-stimulus-response>
```

Listing 11: Example of test attributes where the defines are used.

7.7 Scripts

Single-line calculated attributes allow for simple calculations of attributes but fall short of implementing complex logic or extending the base functionality of tests. For more complex calculations and actions, you will need to use a Python script called from the Experiment Definition File.

7.7.1 Calling a script

LabBench provides access to a full Python scripting engine with the ability to use the Python standard library. This access makes it possible to call functions defined in a script from calculated attributes. Below is an example where a function is called to set which cuff to use as the primary cuff in an ¡algometry-stimulus-response¿ test:

`primary-cuff="func: Setup.StimulatingCuff(tc)"`

The keyword `func:` tells LabBench to call a function in a Python script instead of treating it as a single-line calculated attribute. When the value of the calculated attribute starts with this keyword, it will require that the rest of the line is of the form:

`[ID of Script].[Function Name](tc)`, or

`[ID of Script].[Function Name](tc,x)`.

The ID of the script is the ID of the script assigned to it in its <file-asset> element in the `<assets>` section of the protocol.

7.7.2 Defining functions

The StimulatingCuff function is defined in a script included in the Experiment Definition File that is implemented as:

```
def StimulatingCuff(tc):
    retValue = 0

    if tc.SUBJECT['CLEARING'] == 3:    # Both sides cleared
        if tc.SUBJECT['HAND'] == 1:    # Right hand is dominant
```

```
            retValue = 2
        elif tc.SUBJECT['HAND'] == 2: # Left hand is dominant
            retValue = 1
        else:                         # Both hands are dominant
            retValue = 2
    elif tc.SUBJECT['CLEARING'] == 2: # Left side cleared
        retValue = 1
    elif tc.SUBJECT['CLEARING'] == 1: # Right side cleared
        retValue = 2

    return retValue;
```

Listing 12: Definition of a function that can be called by LabBench.

This defines a function that must be called with one parameter, the test context (tc), and returns an integer. As this function is used for the calculated¡int¿ primary-cuff attribute of the <algometry-stimulus-response> this will control which cuff will be inflated by the test based on the handedness of the subject and whether the subject is cleared for pressure algometry on both sides, the right or left side.

Since functions are always called from calculated attributes, all functions must return the correct type for the calculated attribute from which they are called. Returning the wrong type may result in an error when the experiment is performed.

7.7.3 Test context

For single-line calculated attributes, results, defines, and instruments can be used from the single-line statement as individual variables; however, for script calculated attributes, these variables and more are accessible through the Test Context. When the function is called, the test context (tc) is always passed as a parameter to the function, followed by the free parameter "x" if that variable is available to the calculated attribute.

The test context is a struct variable that gives access to the following variables:

Variable	Description	Access
defines	Defines from the `<defines>` element in the `<protocol>`.	Each define is a variable in the tc struct named by their id attribute in the `<define>` element. Example: `tc.DeltaPressure`

results	Results from each test in the protocol.	Each result is a variable in the tc struct named by their id attribute in the `<test>` element. Example: `tc.SUBJECT['CLEARING']` (the result from the 'CLEARING' question in the questionnaire with id="SUBJECT".
parameters	Parameters added by tests that make it easier to configure the tests based on other parts of its configuration.	Each test-specific parameter is a variable in the tc struct with a name automatically assigned by the test. Example: `tc.NumberOfStimuli` (the number of stimuli in the stimulus set of an <psychophysics-evoked-potentials> test.

devices	Instruments that are used by the test.	These instruments are available through a Devices variable in the tc structure. This Device variable is itself a struct where each instrument is named by the name it is identified within the test.
		Examples:
		`tc.Devices.Stimulator`
		`tc.Devices.Trigger`
		`tc.Devices.Display`

assets	Assets from the <assets> element in the Experiment Definition file.	Assets are available through a Assets variable ni the tc structure. This Assets variable is itself a struct where each Asset is named by its id attribute in its ¡file-asset¿ element. Example: `tc.Assets.Images`

Language	If the experiment is localized the selected Language will be available.	The selected language is available as a variable named Language in the tc struct. Example: `tc.Language`
Session ID	The ID of the session.	The session ID is available as a variable named `SESSION_NAME` in the tc struct. Example: `tc.SESSION_NAME`
Session time	The time the session was started.	The session starting is available as a variable named `SESSION_TIME` in the tc struct. Example: `tc.SESSION_TIME`

As variables cannot have the same name, it will be an error to give defines and tests the same ID, and it must be ensured that the IDs of defines, and tests are not the same as parameters and devices added automatically by the tests. As the names of parameters and devices are constant and given by the tests, there might be IDs that cannot be used based on which tests are included in the protocol. For example, the ¡psychophysics-threshold-estimation¿ test will add its Stimulator and Trigger devices to the test context, and this implies that defines and tests cannot have Stimulator or Trigger as their ID when the <psychophysics-threshold-estimation> test is used.

8 Experimental setups

8.1 Instruments

Each test in a protocol may require a set of instruments to carry out their experimental procedure. Instruments are abstractions over functionality provided by research devices that define the capabilities of a generalized class of devices.

For example, for a stimulus-response test in cuff pressure algometry, you will need an IPressureAlgometer and an IRatingScale instrument. The IPressureAlgometer is for the stimulus-response test used to provide a linearly increasing pressure, and the IRatingScale instrument is used by the subject to rate the pain and to stop

the stimulation when the Pain Tolerance Threshold (PTT) is reached. However, the IPressureAlgometer is not tied to a specific pressure algometry device; instead, it defines what LabBench expects all pressure algometers to be capable of, and today the IPressureAlgometer instrument is provided by two pressure algometers: the Nocitech CPAR and LabBench CPAR+ devices.

Using Instruments in tests instead of concrete devices makes it easier to add new research devices to LabBench. For example, currently, there two types of Cuff Pressure Algometry devices available, where the Nocitech CPAR devices predates the LabBench CPAR+. The CPAR+ offers better performance than the CPAR device and has a trigger system; however, the two devices are largely functionally equivalent. Without Instruments, there would have been two sets of identical tests, one set for the CPAR device, and another set for the CPAR+ device. With Instruments, there is no need to implement a second identical set of tests for the CPAR+ device; all that was required for the CPAR+ device to be used in the tests that were originally intended for the CPAR device was to implement the IPressureAlgometer interface, which greatly reduced the amount of work required to add this new device to LabBench.

The same mechanism can be used to add custom-designed devices to LabBench by identifying Instruments that are provided by the device and implementing a custom LabBench Instrument Driver that implements these Instruments.

8.2 Devices

A specific research device typically also implements multiple instruments. For example, an IRatingScale instrument is built into both the Nocitech CPAR and LabBench CPAR+ devices. Consequently, for cuff pressure algometry tests, you do not need a separate device providing the IRatingScale instrument because this is built into and provided by these devices in addition to the IPressureAlgometer instrument.

Devices are actual research devices such as a CPAR+ device or a NI DAQmx card. When an Experimental Definition File is written the ¡experimental-setup¿ element contains two nested elements the ¡devices¿ and ¡device-mapping¿ elements.

```
<experimental-setup>
    <devices>
        <!-- Research devices used in the experimental setup -->
    </devices>
    <device-mapping>
        <!-- Assignment of devices to instruments required by the tests
    in the protocol -->
    </device-mapping>
```

```
</experimental-setup>
```

Listing 13: Structure of experimental setup definitions.

The ¡devices¿ element list which devices are required for the experiment and specifies their configuration, whereas the ¡device-mapping¿ assigns these devices to the instruments that are required for each test. All devices required by a protocol is listed in the ¡devices¿ as nested elements. These elements assign an ID that is used to identify the device when they are mapped to the Instruments required by tests in the ¡device-mapping¿ element, and they also configure the devices for the experiment.

In our experiment on the relation between DASS scores and pain sensitivity, we will need the subject to answer the DASS questionnaire, which they can do with a joystick. This joystick must be assigned to the ¡meta-survey¿ test that implements the DASS questionnaire and it must be configured with the button actions that is required for the answers. Below it is demonstrated how the Joystick is configured in the protocol:

```
<joystick id="joystick">
    <map experimental-setup-id="survey">
        <button-assignment code="1" button="up" label="Up"/>
        <button-assignment code="2" button="increase" label="Increase"/>
        <button-assignment code="4" button="down" label="Down"/>
        <button-assignment code="8" button="decrease" label="Decrease"/>
        <button-assignment code="16" button="previous" label="Previous"/>
        <button-assignment code="64" button="previous" label="Previous"/>
        <button-assignment code="32" button="next" label="Next"/>
        <button-assignment code="128" button="next" label="Next"/>
    </map>
    <map experimental-setup-id="blank">
    </map>
</joystick>
```

Listing 14: Joystick configuration.

The ¡joystick¿ element in Listing [joystickconfiguration.xml] contains one attribute id, which is used to map the device in the ¡device-mapping¿ element to the IButton instrument required by the ¡meta-survey¿ tests. The rest of the elements within the ¡joystick¿ are elements that are specific to the configuration of the Joystick device. Configuring a Joystick device consists of assigning a button actions (button) to physical buttons (code) on the joystick, and optionally giving them (labels) that may later aid in the analysis of the data. Technically, this is not helpful for the results of ¡meta-survey¿ tests; however, it is included in the example for completeness.

You may note that there is more than one button map and that each is identi-

fied by an experimental-setup-id attribute. Different tests may require different configurations of the devices in the experimental setup. This is implemented with the experiment-setup-id attribute, which in this case is used to define two button maps; one that will be active when the DASS questionnaire is active, and another empty button map that will be active for all the other tests in the protocol. All tests can be defined with an experimental-setup-id that will be activated when the test is selected in the protocol.

The Experiment Definition File does not, however, specify the actual physical devices that are in your laboratory. The reason for this is that the same experiment may be performed at multiple sites, and consequently, the Experiment Definition File cannot know which devices are present. Instead, when an experiment is added to a test site, physical devices are assigned by LabBench Designer to the devices in the ¡devices¿ element.

8.3 Device assignments

The ¡devices¿ elements specify and configure the research devices that are used in the experiment, but it does not configure which devices are used in the tests in the protocol. Each test will require a set of Instruments to perform their experimental procedures. In our experiment, the tests require the following Instruments:

Test	Name	Function
<meta-survey>	Response	Is used by the subject to answer the questionnaire.
	Display	Is used to display the questionnaire to the subject.
<algometry-stimulus-response>	Algometer	Is used for applying the pressure stimuli.
<algometry-temporal-summation>	Algometer	Is used for applying the pressure stimuli.
<algometry-conditioned-pain-modulation>	Algometer	Is used for applying the pressure stimuli.

The names of the required instruments, in this case Response, Display, and Algometer are determined by the tests, which names that must be assigned a device from the ¡devices¿ element can be looked up in the LabBench Language Specification. Each device that is assigned to an Instrument, required by a test, must implement the required Instrument for that name. Assigning a device that does not implement the required Instrument will result in an error that will prevent the experiment from being run by the LabBench interpreter.

In our case, devices are assigned with the following ¡device-mapping¿ element:

```
<device-mapping>
    <device-assignment test-type="meta-survey"
                       instrument-name="Response"
                       device-id="joystick" />
    <device-assignment test-type="meta-survey"
                       instrument-name="Display"
                       device-id="display.survey" />
    <device-assignment test-type="algometry-stimulus-response"
                       instrument-name="Algometer"
                       device-id="cpar" />
    <device-assignment test-type="algometry-temporal-summation"
                       instrument-name="Algometer"
                       device-id="cpar" />
    <device-assignment test-type="algometry-conditioned-pain-modulation"
                       instrument-name="Algometer"
                       device-id="cpar" />
</device-mapping>
```

Listing 15: Mapping of devices to Instruments required by tests in the protocol.

Each mapping of a device to an Instrument is done with a ¡device-assignment¿ element, where the device-id is the id of the device from the ¡devices¿ element, instrument-name is the name of the Instrument from test, and test-type is the type of test that requires the Instrument. You may note from Listing [device-mapping.xml] that one of the assigned devices is identified by [root device].[sub device]. Devices may have sub-devices – in our protocol, the display device has two sub-devices; 1) an ISurvey device (id = "survey"), and 2) an IImageDisplay device (id = "image"). They cannot be active at the same time, the display will either work as an ISurvey instrument, or an IImageDisplay instrument, depending on which experimental-setup-id has been specified by the currently selected test. Assigning an experimental-setup-id that the IImageDisplay to be active when the DASS questionnaire expects an ISurvey device will result in an error.

In the device mapping in Listing [device-mapping.xml] only the type of test is specified (test-type), which means that devices are assigned to all tests of a given type. This is possible because we have a relatively simple protocol where all tests of a given type are used for purposes that do not conflict with each other. For example, we have two ¡meta-survey¿ tests that all get assigned a Response and Display instrument, even though only the ¡meta-survey¿ test for the DASS questionnaire will use these instruments. However, this does not conflict with the purpose of these two tests and will not result in an error while running the experiment. Assigning a device to an Instrument name is never an error, it is only an error to not assigning a device to an Instrument name required by a test.

The simple mapping of a given device to all tests of a given type is possible for our experiment, but there are experiments where this simple mapping breaks down and cannot be used. One example of a protocol that required a more specific mapping was an experiment where auditory and tactile sensitivity was determined for children living with autism. In that case, both the auditory and vibrotactile sensitivity was determined by the same type of test, an <psychophysics-threshold-estimation> test that requires an IAnalogGenerator instrument named Stimulator to be assigned in the <device-mapping>. In the experiment, a sound card and an EA Tactor Device were used to study the auditory and vibrotactile sensitivity, respectively. Both devices implement the IAnalogGenerator instrument, and thus both can be used by the <psychophysics-threshold-estimation> test to determine auditory and vibrotactile thresholds. However, since both thresholds are determined by the same type of test, assigning both devices to the Stimulator name would result in a conflict. For that situation, it is possible to specify both a test-type and test-id on the ¡device-assignment¿ elements. When a test-id is specified, the device will be assigned only to one specific test with the given test-id. With this mechanism it is possible to specify that the test for the auditory thresholds will be using the sound card and the test for the vibrotactile thresholds will be using the EA Tactor Device. The ¡device-mapping¿ element for this mapping is shown below as an example:

```
1  <device-mapping>
2      <device-assignment test-type="psychophysics-threshold-estimation"
3                         test-id="TA1"
4                         instrument-name="Stimulator"
5                         device-id="sound" />
6      <device-assignment test-type="psychophysics-threshold-estimation"
7                         test-id="TV1"
8                         instrument-name="Stimulator"
9                         device-id="tactor" />
10
11     <!--  Other required mapping has been omitted for brevity -->
12 </device-mapping>
```

Listing 16: Device mapping where devices are mapped the Stimulator instrument required by the psychophysics-threshold-estimation test.

9 Sessions

Sessions are central to how data is recorded and stored. Running an experiment with LabBench consists of running a series of sessions that each is identified by an ID. During each session, the tests defined in the protocol are executed.

9.1 Session identifiers

To ensure valid session IDs, it is possible to specify their validation in the Experiment Definition File with the <subject-validator> element. Session `ID` are validated with a regular expression that can be specified by the regex attribute of the <subject-validator> element. The <subject-validator> also has an advice attribute with text that will be displayed to the researcher if they enter an invalid ID. This attribute can be used to provide help to the researcher on what is a valid session ID in the experiment.

A regular expression is a versatile pattern-matching language for string manipulation tasks like searching, matching, and extracting information from text data. It's a sequence of characters that defines a search pattern. They provide a concise and flexible way to describe and match complex patterns within strings. In LabBench, it is used to validate session IDs by checking if the entered ID matches the regular expression. If there is a match the ID is valid.

Below is an example of a <subject-validator> that will ensure that session IDs are in the form "Sxxx" where x is a digit from 0-9:

```
1 <subject-validator regex="^S[0-9]{3}$"
2                    advice="Please enter an ID in the form of SXXX, where
    X is a digit" />
```

Listing 17: Validation of session IDs.

This regular expression encodes the following rules:

1. `^`: is the start of the string.
2. `S`: The string must start with the letter S.
3. `[0-9]`: Digits from 0-9.
4. `{3}`: The letter S must be followed by precisely three digits.
5. `$`: is the end of the string, meaning there cannot be any whitespace for letters after the three digits.

A full tutorial on regular expressions is beyond the scope of this introduction. Regular expressions are widespread and there are numerous excellent tutorials on the net for learning regular expressions, as well as online tools, such as `regex101.com`, that can greatly help in developing and testing regular expressions that can be used to validate session IDs.

9.2 Post-session actions

All data will always be saved during the session in an internal database from which the full dataset for all sessions can always later be exported into a single file. Post-session actions make it additionally possible to perform tasks when a session is completed. Currently, there exists post-session actions for exporting data in Jave Script Object Notation (JSON) (*.json), MATLAB (*.mat), and Comma-Separated-Values (*.csv) formats. Consequently, post-session actions also provide a way to save data automatically to a secure location, such as for example a folder in a network share.

Post-session actions for exporting to JSON and MATLAB formats are simple and only requires a location and filename to which the data is saved. The post-session-action for exporting to the CSV format is considerably more powerful and allows for pre-processing to be performed in the export, which can make subsequent data analysis easier and faster. This ability comes from a limitation of the CSV format. With JSON and MATLAB formats it is easy to export everything that was recorded during the session to a single file, however, the CSV format is not suitable for large data sets and instead, you must select which data points you need to be written to the CSV file. This is implemented so that the post-session action for CSV exports allows you to define a set of data points to be exported each is specified as a calculated attribute. Since the data points are specified by calculated attributes a data analysis for each of these data points, and thus makes it possible to preprocess the results before they are analysed in 3rd party data analysis software.

In our protocol we will use a CSV post-session action to, 1) calculate and export values for stress, anxiety, and depression from the answers to the DASS questionnaire, and 2) calculate statistics for temporal summation and conditioned pain modulation for the pressure algometry, and 3) export other unprocessed datapoints from the results of a session. Post-session actions are specified in the ¡post-actions¿ element in the Experiment Definition File, below is an abbreviated version of the post-session action used for the experiment:

```
<post-actions>
    <export-to-csv name="Calculate scores and export to CSV"
                   location="C:\LabBenchExport\intro"
                   header="true"
                   seperator=";"
                   culture="en-US"
                   filename="dynamic: '{session}.csv'.format(session =
    SESSION_NAME)">
        <item name="ID"
              value="SESSION_NAME"
              default="NA"/>
```

```
11         <!-- Rest of SUBJECT questionnaire omitted for brevity -->
12         <item name="Stess"
13               value="func: Calculations.Stess(tc)"
14               default="NA"/>
15         <item name="Anxiety"
16               value="func: Calculations.Anxiety(tc)"
17               default="NA"/>
18         <item name="Depresion"
19               value="func: Calculations.Depresion(tc)"
20               default="NA"/>
21         <item name="SR01.PDT"
22               value="SR01.PDT"
23               default="NA"/>
24         <item name="SR01.PTT"
25               value="SR01.PTT"
26               default="NA"/>
27         <item name="TS"
28               value="TS[9] - TS[0]"
29               default="NA"/>
30         <item name="CPM"
31               value="CPM.PTT / SR01.PTT"
32               default="NA"/>
33     </export-to-csv>
34 </post-actions>
```

Listing 18: Abbreviated version of the post-session action used for exporting data to a CSV file

The post-session action in Listing 18 uses a combination of calculated attributes for the data points in the CSV file. Some, like the ID and SR01.PDT data points, simply export a single value from the test context to the CSV file. The ID data point is special. It is primarily included because CSV post-session actions can be used in two ways. They will always be executed at the completion of a session and will export the selected data points to a CSV file named by the calculated attribute filename. That export will result in a CSV file with an optional header and one row with the data points from the session. However, it can also be used from the LabBench Designer to export all sessions to a single CSV file with an optional header and one row for each session in the experiment. For that use, the session ID needs to be part of the data set, and consequently, it is good practice when writing CSV post-sessions to always include the session ID (`SESSION_NAME`) as one of the exported data points as otherwise, it would not be possible to identify the session of each row in a CSV file exported from the LabBench Designer program.

However, as the data points are generated from calculated attributes it is possible to perform data analysis to generate the data points. This data analysis can be

performed by single-line calculated attributes or script-calculated attributes for more complex data analysis. The Conditioned Pain Modulation (CPM) is an example of a single-line calculated attribute, whereas the scores for depression, anxiety and stress are calculated by calling functions defined in a Python script.

10 Afterword

The intention of this introduction to LabBench is to provide you with the necessary context and the purpose of all the parts you will need for describing and running an experiment with LabBench. Consequently, it is not a comprehensive guide to all aspects of the LabBench Language and the LabBench Software, rather, it aims to provide you with enough context that you can study the more detailed and technical documentation.

To fully become proficient in describing and running experiments with LabBench, we suggest that you run the protocol described in this introduction. This protocol is available and can be installed from the public LabBench Protocol Repository, which is automatically added to LabBench when the program is installed. This guide has also only described how to describe protocols in the LabBench Language and not how to install, manage, and run them with the LabBench software.

The introduction has also glossed over some of the technical skills required to be fully proficient in LabBench, most notably Python programming. If you want to fully unlock the possibilities offered by LabBench you will need to know a bit of Python programming, a skill that will also be very useful when it comes to analysing the results of your experiments. Several excellent free learning resources and textbooks exist for learning Python programming. We would like to direct your attention to the following resource: Think Python, How to Think Like a Computer Scientist by Allen B. Downey.