the labscript suite

Release 3.3.0.dev0+g1f18c50.d20230419

labscript suite contributors

Apr 19, 2023

DOCUMENTATION

CHAPTER

ONE

EXPERIMENT CONTROL AND AUTOMATION SYSTEM

The *labscript suite* is a powerful and extensible framework for experiment composition, control, execution, and analysis. Developed for quantum science and quantum engineering; deployable in laboratory and in-field devices. Also applicable to optics, microscopy, materials engineering, biophysics, and any application predicated on the repetition of parameterised, hardware-timed experiments.

1.1 Features

- Flexible and automated oversight of heterogeneous hardware.
- The most mature and widely used open-source control system in quantum science.
- Multiple analysis-based feedback modes.
- Extensible plugin architecture (e.g. machine learning online optimisation).
- Readily integrates with other software, including image acquisition, analysis, and even other control systems.
- Compose experiments as human-readable Python code, leveraging modularity, revision control and re-use.
- Dynamic visualisation of experiment composition and results.
- Remote operation: different modules can run on physically separate hosts / single modules can be run on multiple hosts (including hardware supervisor, blacs).
- Auto-generating user-interfaces.
- High-level scripting: user-interface interaction can be programatically synthesised.

1.2 Citing the *labscript suite*

If you use the *labscript suite* to control your experiment or perform analysis, please cite one or more of the following publications:

```
@phdthesis{starkey_phd_2019,
   title = {State-dependent forces in cold quantum gases},
   author = {Starkey, P. T.},
   year = {2019},
   url = {https://doi.org/10.26180/5d1db8ffe29ef},
   doi = {10.26180/5d1db8ffe29ef},
   school = {Monash University},
}
```

```
@phdthesis{billington_phd_2018,
   title = {State-dependent forces in cold quantum gases},
   author = {Billington, C. J.},
   year = {2018},
   url = {https://doi.org/10.26180/5bd68acaf0696},
   doi = {10.26180/5bd68acaf0696},
   school = {Monash University},
}
```

```
@article{labscript_2013,
   author = {Starkey, P. T. and Billington, C. J. and Johnstone, S. P. and
             Jasperse, M. and Helmerson, K. and Turner, L. D. and Anderson, R. P.},
   title = {A scripted control system for autonomous hardware-timed experiments},
   journal = {Review of Scientific Instruments},
   volume = {84},
   number = {8},
   pages = {085111},
   year = {2013},
   doi = {10.1063/1.4817213},
   url = {https://doi.org/10.1063/1.4817213},
   eprint = {https://doi.org/10.1063/1.4817213}
}
```

1.2.1 Installing the *labscript suite*

We're excited to announce that accompanying the recent migration of the codebase from BitBucket to GitHub, *labscript suite* components are now distributed as Python packages on PyPI and Anaconda Cloud.

This makes it far easier to get started using the *labscript suite*, as you no longer require a Mercurial or Git installation (or any knowledge of version control software); components can be installed and upgraded using:

- `pip`: the standard package manager common to all Python distributions; or
- `conda`: a binary package and environment manager, part of the Anaconda Python distribution.

Setting up a Python environment

We recommend installing the *labscript suite* (regular or developer mode) in a virtual environment. This helps sandbox the codebase without interfering with (or being interfered with) your system Python installation, or Python environments used for other purposes. Below we outline how to create and activate a virtual environment for Anaconda Python and other CPython distributions (which we call 'Regular' Python here).

Anaconda Python

Anaconda Python includes a virtual environment manager as part of the `conda` executable. Here's an example (on Windows):

Note: Make sure you have opened the Anaconda Prompt on Windows (available from the Start menu). `conda` is not available from the standard terminal by default. Launching 'Anaconda Prompt' will activate the `base` conda environment.

Warning: We do not recommend using the base conda environment for any project (*labscript suite* or otherwise). Working in a separate conda environment ensures any package resolution or update errors (however unlikely) are limited to that environment, and base is not compromised.

Quickstart

```
(base) C:\> conda create -n py38 python=3.8
(base) C:\> conda activate py38
(py38) C:\>
```

Once activated, the name of the virtual environment (in this case, py38) will prefix the command line.

Detailed Instructions

1. Create a virtual (conda) environment. Here we name it py38 and ask conda to use Python 3.8 within the virtual environment (name and Python version are variable but these are conventional choices):

   ```
   (base) C:\> conda create -n py38 python=3.8
   ```

2. Activate the virtual (conda) environment:

   ```
   (base) C:\> conda activate py38
   (py38) C:\>
   ```

Regular Python

There are a number of ways to configure a virtual environment. If you are unfamiliar with doing so, we recommend using the venv module, part of the Python Standard Library. Here's an example (on Windows):

Quickstart

```
C:\Users\wkheisenberg> mkdir labscript-suite
C:\Users\wkheisenberg> cd labscript-suite
C:\Users\wkheisenberg\labscript-suite> python -m venv .venv
C:\Users\wkheisenberg\labscript-suite> .venv\Scripts\activate
(.venv) C:\Users\wkheisenberg\labscript-suite> python -m pip install --upgrade pip␣
↪setuptools wheel
```

Once activated, the name of the virtual environment (in this case, .venv) will prefix the command line.

Detailed Instructions

1. From a new terminal, create a directory for the virtual environment and enter it. Here we use a directory named `labscript-suite` in the user's home directory, also the location of the labscript suite profile directory. You need create the virtual environment here, but it is a convenient choice.

   ```
   C:\Users\wkheisenberg> mkdir labscript-suite
   C:\Users\wkheisenberg> cd labscript-suite
   ```

2. Create a virtual environment. Here we name it `.venv`, located inside the labscript suite profile directory.

   ```
   C:\Users\wkheisenberg\labscript-suite> python -m venv .venv
   ```

3. Activate the virtual environment:

   ```
   C:\Users\wkheisenberg\labscript-suite> .venv\Scripts\activate
   ```

   Note: This step is OS specific, e.g. on Linux it's `source .venv/bin/activate`.

4. Update the Python package installer and other installation packages of your virtual environment.

   ```
   (.venv) C:\Users\wkheisenberg\labscript-suite> python -m pip install --upgrade␣
   ↪pip setuptools wheel
   ```

Choosing an installation method

Once you have a virtual environment up and running, choose from one of the following 4 installation methods:

1. *Regular installation (Python Package Index)*;
2. *Regular installation (Anaconda Cloud)*;
3. *Developer installation (Python Package Index)*; or
4. *Developer installation (Anaconda Cloud)*.

Regular installation (Python Package Index)

In this example, we will use an existing virtual environment named `.venv` located in `C:\Users\wkheisenberg\labscript-suite`. Skip the first two lines/steps if continuing on from the instructions to *set up this environment*.

Quick start

```
C:\Users\wkheisenberg\labscript-suite> .venv\Scripts\activate
(.venv) C:\Users\wkheisenberg\labscript-suite> python -m pip install --upgrade pip␣
↪setuptools wheel
(.venv) C:\Users\wkheisenberg\labscript-suite> pip install labscript-suite
(.venv) C:\Users\wkheisenberg\labscript-suite> pip install PyQt5
(.venv) C:\Users\wkheisenberg\labscript-suite> labscript-profile-create
(.venv) C:\Users\wkheisenberg\labscript-suite> desktop-app install blacs lyse runmanager␣
↪runviewer
```

Detailed instructions

1. Activate the virtual environment (this step is OS specific, e.g. on Linux it's `source .venv/bin/activate`).

   ```
   C:\Users\wkheisenberg\labscript-suite> .venv\Scripts\activate
   ```

2. Update the Python package installer and other installation packages of your virtual environment.

   ```
   (.venv) C:\Users\wkheisenberg\labscript-suite> python -m pip install --upgrade␣
   ↪pip setuptools wheel
   ```

3. Install the meta-package (`labscript-suite`) from PyPI. This will install blacs, labscript, labscript-devices, labscript-utils, lyse, runmanager, runviewer, and all dependencies:

   ```
   (.venv) C:\Users\wkheisenberg\labscript-suite> pip install labscript-suite
   ```

4. Install PyQt5, the bindings to the GUI toolkit (not installed above for licensing reasons):

   ```
   (.venv) C:\Users\wkheisenberg\labscript-suite> pip install PyQt5
   ```

5. Create (or populate) a profile directory in your home directory (the location of user data; see *Recent changes to the labscript suite*):

   ```
   (.venv) C:\Users\wkheisenberg\labscript-suite> labscript-profile-create
   ```

6. (Optional) Create shortcuts for the GUI applications (blacs, lyse, runmanager, and runviewer) and place them in the start-menu (or non-Windows OS equivalent).

   ```
   (.venv) C:\Users\wkheisenberg\labscript-suite> desktop-app install blacs lyse␣
   ↪runmanager runviewer
   ```

   These will be named, e.g. 'runmanager – the labcript suite' which when clicked on will:

   - Launch the application without a terminal window, using the virtual environment the above command was called in.
   - Display the application with an application-specific shortcut in the taskbar (which can be pinned, like any other desktop application).

Note: Virtual environments named anything other than `.venv` will be included in the name of the shortcut, e.g. 'runmanager – the labscript suite (py38)' for a virtual environment named `py38`.

Alternatively, you can launch the applications from a terminal, e.g.

```
(.venv) C:\> runmanager
```

This will print debugging information to the console.

To launch the applications detached from the console, suffix the application name with `-gui`, e.g.

```
(.venv) C:\> runmanager-gui
```

Note: You must have activated the virtual environment in which the *labscript suite* was installed to use these commands.

Updating a regular installation

Individual components of the labscript suite can be updated using the `--upgrade` (`-U`) flag of `pip`. For example:

```
(.venv) C:\Users\wkheisenberg\labscript-suite> pip install -U runmanager
```

To upgrade to a pre-release version, you can use the `--pre` (pre-relase) flag:

```
(.venv) C:\Users\wkheisenberg\labscript-suite> pip install -U --pre runmanager
```

If updating multiple components, use a single `pip install` command to assist dependency resolution:

```
(.venv) C:\Users\wkheisenberg\labscript-suite> pip install -U labscript lyse runmanager
```

You can also update (or downgrade) to a specific version:

```
(.venv) C:\Users\wkheisenberg\labscript-suite> pip install runmanager==2.5.0
```

Regular installation (Anaconda Cloud)

In this example, we will use an existing conda environment named `py38`. Skip the first line/step if continuing on from the instructions to *set up this environment*.

Quick start

```
(base) C:\> conda activate py38
(base) C:\> conda config --env --add channels labscript-suite
(py38) C:\> conda install labscript-suite pyqt
(py38) C:\> labscript-profile-create
(py38) C:\> desktop-app install blacs lyse runmanager runviewer
```

Detailed instructions

1. Activate the conda environment from the Anaconda Prompt.

   ```
   (base) C:\> conda activate py38
   ```

2. Add the `labscript-suite` channel on Anaconda Cloud to the current conda environment:

   ```
   (py38) C:\> conda config --env --add channels labscript-suite
   ```

3. Install the meta-package (`labscript-suite`) and bindings to the GUI toolkit (`pyqt`) from Anaconda Cloud. This will install blacs, labscript, labscript-devices, labscript-utils, lyse, runmanager, runviewer, and all dependencies:

   ```
   (py38) C:\> conda install labscript-suite pyqt
   ```

4. Create a profile directory in your home directory (the location of user data; see *Recent changes to the labscript suite*):

```
(py38) C:\> labscript-profile-create
```

5. (Optional) Create shortcuts for the GUI applications (blacs, lyse, runmanager, and runviewer) and place them in the start-menu (or non-Windows OS equivalent).

   ```
   (py38) C:\> desktop-app install blacs lyse runmanager runviewer
   ```

   These will be named, e.g. 'runmanager – the labcript suite (py38)' which when clicked on will:

   - Launch the application without a terminal window, using the virtual environment the above command was called in.
   - Display the application with an application-specific shortcut in the taskbar (which can be pinned, like any other desktop application).

Note: Conda environments named anything other than `base` will be included in the name of the shortcut, e.g. 'runmanager – the labscript suite (py38)' for a conda environment named `py38`.

Alternatively, you can launch the applications from the Anaconda Prompt in the , e.g.

```
(py38) C:\> runmanager
```

This will print debugging information to the console.

To launch the applications detached from the console, suffix the application name with `-gui`, e.g.

```
(.venv) C:\> runmanager-gui
```

Note:

- You must have activated the conda environment in which the *labscript suite* was installed to use these commands.
- For the `-gui` entry points to function in Anaconda Python, Step 5 (above) must be completed.

Updating a regular installation

Individual components of the labscript suite can be updated using the `conda update` command. For example:

```
(py38) C:\> conda update -c labscript-suite runmanager
```

To upgrade to a pre-release version, you can use the test label:

```
(py38) C:\> conda updadte -c labscript-suite/label/test runmanager
```

If updating multiple components, use a single `conda update` command to assist dependency resolution:

```
(py38) C:\> conda update -c labscript-suite labscript lyse runmanager
```

You can also update (or downgrade) to a specific version:

```
(py38) C:\> conda update runmanager==2.5.0
```

Developer installation (Python Package Index)

Developer installations are useful for those who want to customise the *labscript suite*.

Note: You need not fork, clone, and install editable versions of all *labscript suite* repositories to customise your installation and/or contribute changes back to the base repositories. For example, if you only want to develop custom labscript device drivers, you might only fork and clone the labscript-devices repository. Moreover, there is now an option to write and use custom labscript device drivers outside of the labscript-devices installation directory.

Quick start

```
C:\Users\wkheisenberg\labscript-suite> .venv\Scripts\activate
(.venv) C:\Users\wkheisenberg\labscript-suite> pip install \
        --src . -e git+https://github.com/wkheisenberg/blacs#egg=blacs \
        --src . -e git+https://github.com/wkheisenberg/labscript#egg=labscript \
        --src . -e git+https://github.com/wkheisenberg/labscript-devices#egg=labscript-
↪devices \
        --src . -e git+https://github.com/wkheisenberg/labscript-utils#egg=labscript-
↪utils \
        --src . -e git+https://github.com/wkheisenberg/runmanager#egg=runmanager \
        --src . -e git+https://github.com/wkheisenberg/runviewer#egg=runviewer \
        --src . -e git+https://github.com/wkheisenberg/lyse#egg=lyse
(.venv) C:\Users\wkheisenberg\labscript-suite> pip install PyQt5
(.venv) C:\Users\wkheisenberg\labscript-suite> labscript-profile-create
(.venv) C:\Users\wkheisenberg\labscript-suite> desktop-app install blacs lyse runmanager␣
↪runviewer
```

Detailed instructions

The following is a detailed explanation of the steps provided in the Quick start section above.

1. Fork the labscript-suite repositories you want to develop using the GitHub online interface. Below we will include all repositories (except the labscript-suite metapackage).

2. Use `pip` to both clone these forks locally and install them into your environment. In this example (on Windows), the forks are owned by the (non-existent) GitHub user wkheisenberg.

   ```
   C:\Users\wkheisenberg\labscript-suite> pip install ^
   --src . -e git+https://github.com/wkheisenberg/blacs#egg=blacs ^
   --src . -e git+https://github.com/wkheisenberg/labscript#egg=labscript ^
   --src . -e git+https://github.com/wkheisenberg/labscript-devices#egg=labscript-
   ↪devices ^
   --src . -e git+https://github.com/wkheisenberg/labscript-utils#egg=labscript-utils ^
   --src . -e git+https://github.com/wkheisenberg/runmanager#egg=runmanager ^
   --src . -e git+https://github.com/wkheisenberg/runviewer#egg=runviewer ^
   --src . -e git+https://github.com/wkheisenberg/lyse#egg=lyse
   ```

   Note:

   - This will set your forked repository(ies) to be the 'origin' remote.

- On Linux / macOS the line continuation character is \ rather than ^.

Alternatively, manually clone the repositories using `git clone` and then install them using `pip` by running the following from the common parent directory:

```
C:\Users\wkheisenberg\labscript-suite> pip install -e blacs -e labscript \
    -e labscript-devices -e labscript-utils -e lyse -e runmanager -e runviewer
```

For a single package, this would look like:

```
C:\Users\wkheisenberg\labscript-suite> git clone https://github.com/wkheisenberg/
↪runmanager.git
C:\Users\wkheisenberg\labscript-suite> pip install -e runmanager
```

3. For each repository, set the upstream remote to the base labscript-suite repository:

```
C:\Users\wkheisenberg\labscript-suite> cd blacs
C:\Users\wkheisenberg\labscript-suite> git remote add upstream https://github.com/
↪labscript-suite/blacs.git
C:\Users\wkheisenberg\labscript-suite> cd ..
```

Repeat for the other repositories.

4. Continue from step 4 (install PyQt5) in the *Regular installation (Python Package Index)* instructions.

Updating a developer installation

This assumes you have already completed the developer installation above and have:

- Forked a *labscript suite* repository on GitHub;
- Cloned the repository;
- Set your fork to be the 'origin' remote; and
- Set the labscript-suite base repository to be the 'upstream' remote.

1. Use one of the following to keep your repository (and feature branches) up-to-date:

Fetch changes, and merge with your local master branch.

```
> git checkout master
> git fetch upstream master --tags
> git merge upstream/master
```

Or using Git Pull:

```
> git checkout master
> git pull upstream master --tags
```

Or using hub sync command-line extension (does not require current local working branch to be master):

```
> hub sync
```

2. Update your feature branches by merging them with master or rebasing them to master:

```
> git checkout your-feature-name
> git merge master <OR> git rebase master --autostash
```

3. Update your fork by pushing any changes resulting from steps 1–2 and/or subsequent local development:

```
> git checkout master
> git push origin master --tags
> git checkout your-feature-name
> git push origin your-feature-name master
```

Note: If the feature branch has not yet been created on your fork, you need to include `-u` above, i.e.

```
> git push -u origin your-feature-name
```

4. Checkout the commit you want to install. This might be a specific release version (which can be specified by tag):

```
> git checkout v0.3.2
```

   or using the commit SHA:

```
> git checkout 59651b5
```

5. (Optional) Update the package using (from within the root of a repository):

```
> pip install -e .
```

   As the installations are in editable mode and the version is being introspected at runtime, this step is not always necessary, but is required for any change requiring setup.py to be run to take effect, e.g. dependency changes, console entry points, etc.

Developer installation (Anaconda Cloud)

Developer installations are useful for those who want to customise the *labscript suite*.

Note: You need not fork, clone, and install editable versions of all *labscript suite* repositories to customise your installation and/or contribute changes back to the base repositories. For example, if you only want to develop custom labscript device drivers, you might only fork and clone the labscript-devices repository. Moreover, there is now an option to write and use custom labscript device drivers outside of the labscript-devices installation directory.

In this example, we will use an existing conda environment named `py38`. Skip the first line/step if continuing on from the instructions to *set up this environment*.

Attention: For the following to work correctly on Windows, you need to use the Anaconda Powershell Prompt, not the Anaconda Prompt. This will be the case until the following bugs are fixed:

- conda-build#3813
- conda#9445

Quick start

Note: After the first line, the current directory is ommited from the command prompt for brevity.

```
(base) C:\Users\wkheisenberg> mkdir labscript-suite
(base) > cd labscript-suite
(base) > git clone https://github.com/wkheisenberg/labscript
(base) > git clone https://github.com/wkheisenberg/runmanager
(base) > git clone https://github.com/wkheisenberg/blacs
(base) > git clone https://github.com/wkheisenberg/lyse
(base) > git clone https://github.com/wkheisenberg/runviewer
(base) > git clone https://github.com/wkheisenberg/labscript-devices
(base) > git clone https://github.com/wkheisenberg/labscript-utils
(base) > conda activate py38
(py38) > conda config --env --add channels labscript-suite
(py38) > conda install setuptools-conda pyqt pip desktop-app
(py38) > setuptools-conda install-requirements ^
         labscript runmanager blacs lyse runviewer labscript-devices labscript-utils
(py38) > pip install --no-build-isolation --no-deps ^
         -e labscript -e runmanager -e blacs -e lyse ^
         -e runviewer -e labscript-devices -e labscript-utils
(py38) > labscript-profile-create
(py38) > desktop-app install blacs lyse runmanager runviewer
```

Detailed instructions

The following is a detailed explanation of the steps provided in the Quick start section above.

1. Fork the labscript-suite repositories you want to develop using the GitHub online interface. Below we will include all repositories (except the labscript-suite metapackage).
2. Manually clone the forked repositories using `git clone`.

   Note: This will set your forked repository(ies) to be the 'origin' remote.

3. Use the `setuptools-conda install-requirements` command to introspect the dependencies from the cloned repositories.
4. Now use `pip` to install the cloned repositories in develop mode without build isolation or installing dependencies via `pip`:

   ```
   C:\Users\wkheisenberg\labscript-suite> pip install --no-build-isolation --no-deps ^
   -e labscript -e runmanager -e blacs -e lyse -e runviewer -e labscript-devices -e␣
   ↪labscript-utils
   ```

   Note: On Linux / macOS the line continuation character is \ rather than `^`.

5. For each repository, set the upstream remote to the base labscript-suite repository:

```
C:\Users\wkheisenberg\labscript-suite> cd blacs
C:\Users\wkheisenberg\labscript-suite> git remote add upstream https://github.com/
↪labscript-suite/blacs.git
C:\Users\wkheisenberg\labscript-suite> cd ..
```

Repeat for the other repositories.

6. Continue from step 4 (create the labscript profile) in the *Regular installation (Anaconda Cloud)* instructions.
7. (Optional, but Recommended) Remove `conda` and its dependencies from the `py38` environment. This will allow you to use the standard Anaconda Prompt again with this environment without issues. The particular issue being addressed is that `setuptools-conda` installs the `conda` package in a non-base environment, which can cause issues. Once the installation is complete, `setuptools-conda` and its dependices are no longer needed and can be safely removed using:

```
conda remove conda
```

Note that this command will only work if you use the Anaconda Powershell Prompt and have installed the labscript suite into a non-base environment as described above.

Updating a developer installation

This assumes you have already completed the developer installation above and have:

- Forked a *labscript suite* repository on GitHub;
- Cloned the repository;
- Set your fork to be the 'origin' remote; and
- Set the labscript-suite base repository to be the 'upstream' remote.

All instructions for updating the labscript developer installation are identical to those for a *pip developer installation*.

1.2.2 Configuring the *labscript suite* for first run

Once installed, the **labscript suite** requires some manual configuration. The following sections detail the required steps necessary for all components of the suite to run for the first time. Further details on how to use each component can be found in the documentation for that component via the **labscript suite* components* link in the left side bar.

The *labconfig.ini* File

Before running any **labscript-suite** components after a fresh installation, you should first open the automatically generated labconfig file and modify some of its default settings to match your needs. This file is found in the user space `labscript-suite/labconfig` directory created by the `labscript-profile-create` command of the installation process. This directory is, by default, located at the top level of the user's directory. The labconfig file will be named after your system's name. So if your system hostname is `heisenberg`, the labconfig file will be named `heisenberg.ini`. This directory will also contain a copy of the default labconfig file as `example.ini`. Only changes made to the labconfig file named after your system will have any effect.

The current contents of the default labconfig file are available here.

Configuration

Many of the options in this file do not need to be changed, but a few things should be changed before running any **labscript-suite** module the first time. Once the appropriate changes have been made, save the file.

- `[DEFAULT]`
  - **`apparatus_name` should be set to a name describing the experiment.** It should not have spaces. This name is used to create sub-directories within the various **labscript-suite** directories where the various shots and scripts will be stored. If these sub-directories do not exist, the suite will create them and populate them with the bare minimum of requirements to function. The relative paths for the items are described by the other keys in the `DEFAULT` section, and can be modified if desired. Note that while multiple apparatus names can exist for the same installation, only one can be used at a time. In order for a changed `apparatus_name` to take effect, you will need to reload **labscript-suite** components.
  - **`shared_drive` is the path to where the individual shots are stored and accessed by the labscript-suite components.** If your installation spreads the **labscript-suite** components across multiple separate computers, set this value to the path to the common network location where shots are stored. This allows all components seamless access to the same shots.
- `[servers]`
  - **`zlock` should be set to the network address of the computer that runs the common zlock server.** In an installation spanning multiple computers, only one computer runs the zlock server. This ensures only one device can access files at a time. At least one computer in an installation must be set to `localhost`, indicating it is the zlock server.
  - **`runmanager` should be set to the network address of the computer that runs a common runmanager server for compi** Rules to set it are the same as `zlock`. It can be the same computer as the zlock server.
- `[security]`
  - **`shared_secret` should be set to the full path to the zprocess secret key for encryption.** The default installation will create this file for you, place it in the correct folder, and update the reference in the config file to reflect its automatically generated name. In a multi-computer installation, all computers will need access to the same secret key in order to commuicate with encryption. Each computer's labconfig file will need to be updated to point to the same key file.
  - **`allow_insecure` can be added to this section and set to `True` to disable secure communications.** This value will need to be the same across all computers in a multi-computer setup.

With these configurations made, you should now be able to run any module of the **labscript-suite**, except BLACS which requires more configuration.

Initial Compile of the Connection Table

Once your installation is configured, you will need to compile a connection table for the first start-up of BLACS. A general description of the connection table can be found here. In short, the connection table defines the hardware devices connected to BLACS and are available for use in an experiment shot. The connection table should be considered a superset of devices for use in experiment shots (i.e. not all devices connected to BLACS must be used in an experiment shot run by that BLACS).

The connection table is defined by writing a `connection_table.py` file that is essentially an experiment run without any instructions. This involves importing the specific device code and instantiating each device you wish to connect to. At the end of the file, you will call the **labscript** functions `start` and `stop`, without any actual instructions commanded to any device in the shot. This file must be saved as the name defined and at the location specified by the `connection_table_py` key of the labconfig file. The defaul location is in the `apparatus_name/labscriptlib` sub-folder of the userlib.

A very simple connection table that defines a PrawnBlaster pseudoclock and an NI DAQ with two named channels is as follows.

```
from labscript import *

# Import classes needed for the devices which will be used
from labscript_devices.PrawnBlaster.labscript_devices import PrawnBlaster
from labscript_devices.NI_DAQmx.models.NI_USB_6363 import NI_USB_6363

# Create a PrawnBlaster, saved to the variable 'prawn',
# It will be used as the single pseudoclock that triggers other devices
PrawnBlaster(name='prawn', com_port='COM6', num_pseudoclocks=1)

# Create a NI USB-6363 multifunction I/O device, clocked by the PrawnBlaster
NI_USB_6363(name='daq', MAX_name='Dev1',
            parent_device=prawn.clocklines[0], clock_terminal='/Dev1/PFI0',
            acquisition_rate=100e3)

# Add analog output channels to the USB-6363
AnalogOut('ao0', daq, 'ao0')
AnalogOut('ao1', daq, 'ao1')

# The following is standard boilerplate necessary for the file to compile
if __name__ == '__main__':

   start(0)

   stop(1)
```

More specific examples of connection tables can be fould in the **labscript-devices** repository here.

Note: BLACS will instantiate a control for all available hardware channels on a device, even if they are not specifically named in the connection table. However, connection tables with identical devices but different names for the attached channels are considered unique by **labscript**. Remember that the connection table used by an individual shot must be a subset of the connection table used by BLACS, so chaning channel names will require re-compiling the connection table.

With the `connection_table.py` file written, you will then need to compile it using runmanager. Take the output compiled file and save it by the name and in the location specified by the `connection_table_h5` key of the labconfig file. The default name of the file is `connection_table.h5` and it is located in the experiment shot storage for `apparatus_name`.

With the connection table in place, you can now open BLACS. Changes to `connection_table.py` will now be recognized by BLACS, and BLACS will prompt you to recompile the connection table using a prompt within BLACS itself.

1.2.3 Troubleshooting

This section contains general tips for troubleshooting issues with the **labscript-suite**. Other good places to look for help include the list serve as well as open/closed issues and pull requests on github. If you cannot find the help you need, feel free to send a message on the list serve or create a github issue in the relevant repository.

I clicked the launch icon for *component* and nothing happened!

This occurs when there is a stopping error that prevents the GUI from loading (when standard error handling would take over). In order to see the error that is preventing the component from launching, you should launch the program from the command line. To do so, run the following command in the python environment you installed the **labscript-suite** into.

```
python -m <component>
```

Replace `<component>` with the name of the module you wish to run (i.e. `runmanager`, `blacs`, `lyse`, or `runviewer`). The blocking error should print to the console you ran the command in. You can then use that error to isolate the problem and find the appropriate solution. If the component starts without any error, try to run the component from the command line with *deactivated* python environment (for that you need to find the path to the executable file from the desktop shortcut).

`zlock`/ `zlog` is preventing *component* from loading!

`zlock` and `zlog` are the **labscript-suite** utilities that handle file locking and log files for the suite. They are built on the zprocess package. Generally, they are run as background processes; automatically launched when any **labscript-suite** component first launches and left running in perpetuity. If their configuration changes or the labscript-utils/zprocess packages are updated while the background processes are running, errors often ensue. These problems can often be fixed by restarting the background processes. You can stop the these processes by either:

- Restarting the host computer.
- Killing the processes running `zlock` and `zlog` manually using a task manager. The processes display as a python executable. You can determine which python processes are running these components by inspecting their command line arguments.

You restart the two processes by either starting a **labscript-suite** component or starting them by hand at the command line as described in the API documentation in *labscript-utils*.

1.2.4 Building the *labscript suite* docs

The **labscript-suite** documentation is hosted online at https://docs.labscriptsuite.org/en/latest/. It is composed from documentation for each module of the **labscript-suite**. If you desire to build the documentation for a module locally, either for off-line usage or to test changes to the documentation, you will need to apply extra configuration beyond a normal installation. This is because our documentation building pipeline relies on sphinx, which introspects documentation from inline docstrings from the source code. This introspection process requires that the code to be documented is importable and the necessary `sphinx` dependencies are installed.

The below sections describe how to configure your environment for building docs as well as some guidance for writing and contributing documentation.

Setting up the environment

The build environment generally requires a full **labscript-suite** installation, with a few extra dependencies. The simplest method is to use the environment of an existing, functioning installation and install the build dependencies. If you do not want to install the docs dependencies into a working environment of an experiment, or wish the develop docs on a system that does not control an experiment, you can create a new environment.

Note: These instructions assume you have generally followed the installation instructions for a developer **labscript-suite** installation, using either the *pip* or *anaconda* method.

Install build dependencies

If you have an existing, functioning environment for using the **labscript-suite**, you can configure it to build docs by simply installing the docs dependencies found in the `setup.cfg` file of each repository. For a functioning **labscript** environment, the only additional dependencies should be `sphinx`, `sphinx-rtd-theme`, `recommonmark`, and `m2r`.

The versions pinned in the `setup.cfg` file will be the versions of these packages that are used by Read the Docs when building the documentation. In order to get an accurate reproduction of what the online docs look like, you should ensure the versions you install match the pins.

Once these dependencies are installed in your **labscript** environment, you are ready to build the docs.

Using a cloned environment

If you have an existing, functioning **labscript** environment, but do not want to install the many build dependecies into it, you can instead build the docs using a cloned copy of the functioning environment. You clone a conda enviroment using

```
conda create -n labscript-sphinx -c labscript
```

Here we clone an existing `labscript` conda environment into a new environment `labscript-sphinx`. This new enviroment will be an exact duplicate of the original at the time of cloning. Changes in one environment will not affect the other.

Once the new environment is created, you can activate it then install the dependencies for the docs as described above.

When using this method, both enviroments will use the same installation of the **labscript-suite**.

Using a new environment

If you desire to build the docs on a computer without an existing **labscript-suite** installation, you will need to first install the suite using one of the developer installation methods described *here*. Once installed, follow the directions above to install the docs dependencies into the environment.

Note: The normal installation steps to create a labscript profile and the launcher shortcuts should not be necessary when only building docs.

This installation method should also be used if you desire to develop documentation in local repositories separate from an existing installation (i.e. having two environments using different installations of the **labscript-suite**). This can be desirable if you want to avoid docs dependencies in a function environment *and* want to isolate local changes for a functioning experiment from documentation developement.

Building the Docs

The **labscript-suite** documentation is built one repository at a time. To build the documentation for a single repository:

1. Activate the environment where the **labscript-suite** and sphinx dependencies are installed.
2. Change directories to the `docs` subfolder of the repository.
3. Run the appropriate `make` command, described below.

Assuming the appropriate dependencies are installed, the documentation will be built and placed in a subfolder of `docs/build` corresponding to the build command used.

html

The web-based documentation, which is what is hosted at https://docs.labscriptsuite.org/en/latest/ by Read the Docs, is built locally using the command

```
make html
```

The home page is found at `docs/build/html/index.html`. Repeated calls of this (and the other) build commands will introspect which source files have changed and only update the corresponding build outputs.

Note that the build on Read the Docs uses the closely related `make dirhtml` command. This build command organizes the html documentation in a way suitable for web hosting. For locally inspecting the documentation, the `make html` command is preferred to preserve normal inter-page links.

Note: Some cross-referencing used in the markdown files is not cross-compatible between the `html` and `dirthtml` build commands. When using markdown source files, please ensure cross-references actually work when built on Read the Docs.

latexpdf

Building the pdf documentation is a bit more complicated than the other builds. Normally it would be done by running the command

```
make latexpdf
```

This would create latex source files which are automatically compiled using an existing, local installation of latex. The latex compilation requires `perl` and the `latexmk` latex package. It also requires a great many other latex dependencies. Successfully building the pdf documentation locally is made easier if your latex installation can install dependencies as required.

Unfortunately, this simple build command *does not succeed* for **labscript-suite** documentation. To build the pdf docs locally, you will need to instead build using the `make latex` command followed by the latex compiling command used on Read the Docs.

```
latexmk -r latexmkrc -pdf -f -dvi- -ps- -jobname=repository-name -interaction=nonstopmode
```

This command is run from within the `docs/build/latex` directory where the `latexmkrc` file resides.

epub

This builds the documentation in the EPUB format, for use with e-readers. It is built using the command

```
make epub
```

clean

This make target will clean the entire `build` directory. It ensures that a fresh build can be made. It is helpful when stale build files are interfering with new changes or you wish to see the *coverage*. The command is

```
make clean
```

Writing Docs

Documentation for the **labscript-suite** is generally written at two levels: manually created top-level narrative docs and automatically introspected API docstrings. These two types of documentation live in different places within the code repository and have different purposes, as described in the following sections.

As a general guideline for writing documentation, please ensure no more than one sentence is on a line in the source file. This ensures diffs will only highlight the changed senstences of the paragraph, instead of large blocks of text. Note that paragraph breaks require a blank line between lines of text.

Top-Level

Higher level documentation that outlines the general usage of the **labscript-suite** components are found in the `docs` folder of each project repository. These documents can be written in either ReStructuredText (rst) or MarkDown (md). Specific details for writing documentation in both formats is given below.

The navigational structure of the documentation is governed by `toctree` rst directives, where each file included in a `toctree` forms the basis of a new page. It is considered good practice to organise documentation logically by subject into individual files and sub-directories. Examples of documentation sources can be readily found by clicking the `Edit on GitHub` link at the top right corner of any page in the on-line html documentation.

ReStructuredText

ReStructuredText is the native language for writting documention in a sphinx-based system. As such, most documentation should be written in rst. Examples and syntax guides are available via links *here*.

An important note when creating new rst doc files is to try to use unique filenames within a single repository. These filenames are ultimately used in the html page paths and are used when cross-referencing pages within the documentation. Files in a single project with identical names can be difficult to uniquely reference.

MarkDown

Markdown syntax documentation can also be used within the **labscript-suite** via the m2r package. Given markdown's simpler syntax, it can be an ideal choice for long prose documentation. Note that, unlike rst files, markdown files cannot be used indepently to form a new page. Instead, markdown source files must be included directly into a parent rst file via an `mdinclude` rst directive. An example of how this is done is found in the homepage of each project (i.e. the `index.rst` and `main.md` files in the `docs` root directory). Documentation that requires higher level features (particularly cross-referencing) are often better written in rst directly.

API/Docstrings

Docstrings are written within the source code itself. Their purpose is to document the inputs, outputs, and usage details of the classes, functions, and attributes in the API. The **labscript-suite** uses Google style formatting for the docstrings. A detailed example of the syntax is here and further details can be found in other places on-line.

Docstring Coverage

As part of the build process, a rough estimation of the API coverage is reported to the command line at the end of the build. It counts the number of documentated objects relative to the total number of objects processed. This report is only approximately accurate when starting from a clean build, as sphinx does not re-process items that are unchanged from a prior build. Note that an object is counted documented if it's docstring is not empty, so incomplete docstrings are not caught.

Committing Docs

Committing documentation updates/fixes is an important way to contribute to the **labscript-suite** community. Increasing the amount of technical details in the documentation that are necessary to use the suite but are not obvious from the source is of particular value.

The mechanics of committing documentation to a **labscript-suite** component is nearly identical to committing general code updates/fixes: *a pull request needs to be created*. The main difference is that documentation changes necessitate checking the read the docs builds to confirm what was desired actually makes it into the on-line documentation as intended.

Making a PR

The general steps for creating the PR are outlined in here. When making PRs with documentation changes, a few extra checks are required. In particular, it is a good idea to locally build the documentation as described *here*. If you cannot build locally, the docs will be built automatically for your PR by Read the Docs and can be viewed there (described below).

Checking the RTD Builds

When a PR is created or updated with new commits, an automated check of the documentation build is performed by Read the Docs. The result of this build can be viewed online by looking at the details of the automated checks. If the build is completed, this link will take you directly to the home page of your built documentation. If the build is still in progress, this link will take you to the build progress which shows the commands being run and their outputs. If you wish to see this progress after the build succeeds, you can find it by clicking the bottom left corner of the Read the Docs page. This will bring up a small window pane. Selecting 'Builds' will bring up the build logs for all of the online builds. If the build did not succeed, this link takes you to the build progress stage with the failing command and corresponding outputs displayed.

Note that Read the Docs will only build the html documentation for a pull request. When the pull request is merged, Read the Docs will build the html documentation again, as well as downloadable pdf and epub versions. These downloads are available via the 'Downloads' link next to the 'Builds' link in the bottom left corner pop-up pane on-line.

Documentation Resources

A list of links to relevant resources used by our sphinx configuration.

Relevant Sphinx Docs

- autodoc Extension
  - Used extensively to automatically introspect the API documentation.
- autosummary Extension
  - Used to automatically generate stubs for the API documentation for all modules except *labscript-devices*.
- intersphinx Extension
  - Used to generate cross references between repositories. The cross-reference syntax is described better here.
- sphinx-apidoc
  - Used in *labscript-devices* to generate stubs for the NI-DAQmx models subclasses.

Syntax Cheatsheets

- Read the Docs Example project
  - Shows a built project that uses many rst and sphinx features. Go to the associated github repo to see the source that produces it.
- Google-style Docstring Example
  - Example that contains most of the syntax necessary to write docstrings.

    Note: You must follow this syntax for correctly formatted docstrings.

- RST cheatsheet
  - Cheatsheet for RST directives, with a focus on use with Sphinx.
- A ReStructuredText Primer
  - Authoritative summary reference for full RST sytanx.

Minutia

Here is a list of minor details that come up when dealing with our docs that are not well documented in the Sphinx docs or are very unique to our project.

autosummary can't find a module

When you get an error while building the docs stating autosummary could not find a module, it could be due to one of two things:

1. The module actually cannot be found, due to a typo in the `autosummary` command.
2. The much more likely reason, the module is actually found, but cannot be imported.

In order to see the actual error to diagnose the problem properly, temporarily remove that module from the `autosummary` directive and do an explicit `automodule` directive. Once the import error is fixed, you can move the module back into the `autosummary` directive.

labscript-devices API

Unlike the rest of the **labscript-suite**, **labscript-devices** does NOT use recursive `autosummary` calls to generate the API documentation. This means that, unlike other modules, adding a new device to **labscript-devices** will NOT automatically be captured in the docs build. When adding a new device, you are expected to make a top-level rst document that implements the necessary `autodoc` calls to document the device.

Note: In all modules, adding a sub-module at the top-level likely will not be automatically caught either. It will need to be added to the relevant `autosummary` directive manually.

latexPDF local builds

Local builds of the latexPDF version of the documentation will not work using the standard `make latexpdf` command. This is because the **labscript-suite** uses svg figure icons that latex cannot process. This build **does** work on RTD because they do not use the `latexpdf` make target, but instead call `latexmk` with customized options that ignore these little errors. In order to build the PDF documentation locally, you will need to use the same call as RTD uses.

Note that the RTD latex build is fairly stable, if really ugly, so long as the html docs build. That said, if there is an error unique to the latex build, discovering its origin can be very difficult since the build process has to ignore errors to complete normally anyway.

One specific error already encounted is when an API documentated value is too long for latex to parse as a single line (a raw bitmap image saved as a class attribute). This specific error can be overcome by instructing sphinx to not publish the value in the docs using the `:meta hide-value:` rst key for the offending object.

intersphinx references won't link

Using intersphinx links to reference documentation in other packages (or even the same package) can be tricky. This is because the exact convention for referring to things is not guaranteed between projects. The best way to determine the exact reference label to use is to introspect the intersphinx object inventory for that project (the `objects.inv` file). This can be done by calling, for example,

```
python -m sphinx.ext.intersphinx https://www.sphinx-doc.org/en/master/objects.inv
```

See this SO post for many details.

This file is typically stored at the top level directory of a project, but not always. If the above command does not work to obtain the objects inventory for online docs, it can be run on a local copy of the file without issue. Obtaining the file for online published docs amounts to guessing the correct location and pointing your browser there. To run the introspection locally, use

```
python -m sphinx.ext.intersphinx objects.inv > objects.txt
```

In this call, we redirect the output to a text file for easier inspection.

1.2.5 Recent changes to the *labscript suite*

Upon migrating the code base to GitHub and publishing distributions on PyPI in April–May 2020, existing users should be aware of the following recent changes.

Profile directories

The *labscript suite* profile directory, containing application configurations, logs, and user-side code, is now located by default in the current user's home directory, e.g. for a local user named `wkheisenberg` this is:

- `C:\Users\wkheisenberg\labscript-suite` on Windows.
- `~/labscript-suite` or `/home/wkheisenberg/labscript-suite` on Linux and Mac OS X.

A typical structure of the profile directory is:

```
    ~/labscript-suite/
    ├── app_saved_configs/
    │   ├── default_experiment/
    ├── labconfig/
    ├── logs/
    └── userlib/
        ├── analysislib/
        ├── labscriptlib/
        ├── pythonlib/
        └── user_devices/
```

This structure is created by calling the command `labscript-profile-create` in a terminal after installing `labscript-utils` (per the installation instructions).

Note: As of labscript-suite/labscript-utils#37 an editable installation can be located within the labscript-suite profile directory.

Secure communication

Interprocess communication between components of the *labscript suite* is based on the ZeroMQ (ZMQ) messaging protocol. We have supported secure interprocess communication via encrypted ZMQ messaging since February 2019 (labscript-utils 2.11.0).

As of labscript-utils 2.16.0, **encryted interprocess communication will be the default**. If you haven't already, this means you'll need to create a new shared secret (or pre-shared key) as follows:

1. Run `python -m zprocess.makesecret` from the labconfig directory.
2. Specify the path of the resulting `shared_secret` in your labconfig. For example:

   ```
   [security]
   shared_secret = %(labscript_suite)s/labconfig/zpsecret-09f6dfa0.key
   ```

3. Copy the same pre-shared key to all computers running the *labscript suite* that need to communicate with each other, repeating step 2 for each of them.

Treat this file like a password; it allows anyone on the same network access to *labscript suite* programs.

If you are on a trusted network and don't want to use secure communication, you may instead set:

```
[security]
allow_insecure = True
```

Notes:

- Steps 1 and 2 are executed automatically as part of the `labscript-profile-create` command. However, for multiple hosts, step 3 above must still be followed to ensure the same public-key is used by all hosts running *labscript suite* programs.
- There is an outstanding issue with the ZMQ Python bindings on Windows (zeromq/pyzmq#1148), whereby encryption is significantly slower for Python distributions other than Anaconda. Until this issue is resolved, we recommend that Windows users on an untrusted network use the Anaconda Python distribution (and install `pyzmq` using `conda install pyzmq`).

Application shortcuts

Operating-system menu shortcuts, correct taskbar behaviour, and environment activation for the Python GUI applications (blacs, lyse, runmanager, and runviewer) is now handled by a standalone Python package desktop-app (per installation instructions above). This currently supports Windows and Linux (Mac OS X support is forthcoming).

Lab configuration

The `experiment_name` item has been renamed to `apparatus_name` in the labconfig (.ini) file, to better reflect the distinciton between the infrasturcture that experiment shots are executed on. The old keyword can still be used for this item, but a warning will be issued to remind you to update your labconfig.

Source code structure (developer installation)

Existing users who move to a developer (editable) installation, please note the following structural changes to the *labscript suite* source code:

- Each package has a top-level folder containing setup.py and setup.cfg used to build a distribution from source. The functional code base now resides in a subfolder corresponding to the name of the Python module, e.g. an editable installation might contain folders:

```
<path-to-your-labscript-installation>/
├── blacs/
│   └── blacs/
├── labscript/
│   └── labscript/
├── labscript-devices/
│   └── labscript_devices/
├── labscript-utils/
│   └── labscript_utils/
├── lyse/
│   ├── lyse/
├── runmanager/
│   └── runmanager/
└── runviewer/
    └── runviewer/
```

- Package names (shared by repositories and top-level folders) are now hyphenated, e.g. labscript-devices and labscript-utils.
- Module names remain underscored, e.g. labscript_devices and labscript_utils.
- The mixing of hyphen and underscores is inelegant but conventional.
- All references to blacs are now lowercase.
- As installation no longer requires a separate package, the repository formerly named 'installer' has been renamed to 'labscript-suite', and is a metapackage for the *labscript suite* (installing it via `pip`/`conda` installs the suite).

Versioning (developer installation)

Aside from the maintenance branches described here, versions of the labscript suite packages are introspected at run-time using either the importlib.metadata library (regular installations) or setuptools_scm (developer installations). Thus any changes to an editable install will be traceable by local version numbers, e.g. editing the released version of a package with version 2.4.0 will result in 2.4.0dev1+gc28fe94, for example. This will help us diagnose issues users have with their editable installations.

1.2.6 Contributing to the *labscript suite*

We are very grateful for all the contributions users have made in the past decade to make the *labscript suite* the most widely used open-source experiment control and automation system in quantum science. These include development, suggestions, and feedback, and we look forward to this continuing on GitHub.

Issue tracking

The issue tracking on GitHub is very similar to BitBucket, with the added advantage that you can add inter-repository issue references, e.g. referring to labscript-suite/runmanager#68 in any issue or pull request will link to the corresponding issue. We have imported all issues from the BitBucket repositories into the GitHub repositories. This import is not perfect (as each comment is now posted by Phil Starkey) but the comments have been modified to contain the original author attribution. We have also updated all links to files, pull requests, issues, and commits so that they point to the equivalent GitHub location and/or the archived copy of the data (as discussed above).

Please use the issue tracker of the relevant GitHub repository for:

- Reporting **bugs** (when something doesn't work or works in a way you didn't expect);
- Suggesting **enhancements**: new features or requests;
- Issues relating to **installation**, **performance**, or **documentation**.

For advice on *how* to use the existing functionality of the *labscript suite*, please use our mailing list.

Request for developers

We would like to reaffirm our invitation for users to directly contribute toward developing the *labscript suite*. We have established a separate discussion forum on Zulip for discussing development direction and design. If you are interested in being a part of these discussions, and/or testing and merging pull requests, please reach out to us.

Pull requests

We will continue the same feature-branch workflow as before.

1. **Fork one or more of the labscript suite repositories.**
2. **Create a named branch on your fork for a new feature.** It is good practice to use a unique, brief, yet descriptive, name for this branch.
3. **Check out this branch and commit code to it.** It is good practice to commit often to this branch to avoid very large diffs in a single commit that are hard to analyse. Ideally, your commits should accomplish a single thing at a time. It is also good to include detailed commit messages that explain the intended purpose of the commit and details of any specific usage changes, if necessary.
4. **Once you feel your changes are complete, please ensure they are well tested**. This includes ensuring your proposed changes actually do what you think they should. It also includes ensuring other components are not negatively affected by your changes.
5. **Push your commits to your fork.** This can be done at any time in this process, but it must be done at least once at the end to ensure your changes are made available on GitHub for the PR.
6. **With changes complete, tested and pushed to your fork, you create the PR**. This can be done in a few ways, but here we describe (briefly) how this is done via the web interface. Detailed documenation of this process is available at the links below.
    1. When you go the home page of your fork, you will see a notice that there are new changes to a branch on your fork, do you want to merge? You can also access the PR creation menu by going to the original **labscript-suite** component and selecting Pull Request.
    2. You next select which branch of your fork you would like to merge into which branch of the upstream fork. Ensure you select your fork that you wish to commit and the master branch of of the upstream **labscript-suite** component repository.
    3. Fill in a descriptive title for your proposed changes and a description of the changes you have made.

    4. Select "Create Pull Request". Your PR is now created and the **labscript-suite** core developers will be notified.

7. **The core dev team will review the changes, and often ask for changes.** By pushing new commits to the same branch as that used by the PR, those changes will automatically be added to the PR.

8. **Once all concerns are addressed, the PR can be merged by the core dev team.** Note that the dev team may decide that your PR should not be merged, and ultimately reject it. We will do our best to explain the rationale behind this decision. You are obviously welcome to use your code within your own installation of the **labscript-suite**, even if it is not merged into the upstream mainline. Please do not take a rejected PR personally.

The official GitHub documentation outlining this process is available here. These steps are broadly covered in the GitHub Hello World guide, and in detail on the NumPy development workflow.

Branching model/strategy

The move to GitHub for source control and PyPI for distribution is accompanied by a slight change in the branching strategy, to improve deployment and stability of the *labscript suite*. Whereas before all versions corresponded to single commits on the master branch; dedicated branches will be used to release and service minor versions. For example, releasing v0.1.0 would see the creation of a branch named maintenance/0.1.x, used to service all 0.1 versions. As we adhere to semantic versioning, bug-fixes would be applied in this branch, bumping the minor (final) version number each time, e.g. 0.1.1, 0.1.2, etc. No development will occur in these branches; new features are merged into master, and bug-fixes are cherry-picked from master.

You can learn more about this branching model at:

- releaseflow.org
- NumPy development workflow
- Release Flow – Azure DevOps

Learning Git

As our former development, installation, and upgrading practices involved Mercurial revision-control, some of you may not be familiar with Git. While you no longer need to use *any* revision control system to use the *labscript suite*, those of you wanting to contribute to development who aren't acquainted with Git may benefit from these resources:

- NumPy: Getting started with Git development
- GitHub Guides: Very cogent information for beginners. We recommend starting with:
    - Hello World
    - Git Handbook *Note:* You may notice references to 'GitHub Flow' in these guides (and 'Git Flow' elsewhere). These share some aspects of the Release Flow branching-workflow we use, but are distinct.
- Atliassian Git Tutorials: Despite the many references to BitBucket (ignore these); there is a wealth of excellent beginner information for using Git at the command line here; and finally
- Oh Shit, Git!?! Mistakes happen. This is a good place to start fixing them. (Censored version.)

1.2.7 Community Resources

This page contains a list of general resources developed by the community. A separate list of 3rd-party device repositories is maintained in the *labscript-devices* documentation.

These resources are not specifically developed or maintained by the **labscript-suite** development team. They have been graciously provided by community members for the benefit of other **labscript-suite** users in the hope they may be useful. Please direct any questions to their respective owners.

If you have something you would like to add to this list, *please contact us or make a pull request*.

Labscript-Suite-Tutorial

https://github.com/josiahsinclair/Labscript-Suite-Tutorial

Developed by Josiah Sinclair at the MIT-Harvard Center for Ultracold Atoms (CUA), this detailed step-by-step tutorial covers everything from installing python through running your shot. It uses *regular python* with the *regular pip installation method*. For hardware it assumes you have a PrawnBlaster, a NI DAQ with analog outputs, and an oscilloscope.

1.2.8 BitBucket archive

In April–May 2020 the *labscript suite* code base was migrated from BitBucket to GitHub. All commit history and issues was preserved, however some repository metadata (such as pull request discussions) could not be migrated directly. As such, we have created an archived copy of everything that was on BitBucket. This includes:

- Issues (as they appear on BitBucket);
- Pull requests discussions;
- Commit comments for every labscript suite repository; and
- Every public fork (as of 1st February, 2020).

This archive can be found at bitbucket-archive.labscriptsuite.org (this page can take some time to load for the first time). Copies of every public fork of our repositories are at github.com/labscript-suite-bitbucket-archive. As this is an archive, we will not be transferring ownership of these repositories back to their original owners. However, should you wish to continue development on one of those repositories you can fork it into your own account through the GitHub web interface. Should you have uncommitted changes (or changes made after 1st February, 2020) that you wish to have archived, please contact us to discuss the best approach to including these. Please note that we are not recommending continuing development in such forks long term, due to the changes in package structure outlined above.

What to do if you had custom code in a fork on BitBucket

labscript experiment scripts and lyse analysis scripts can be copied or moved to the new labscriptlib/analysislib folders. We deem these user-side code as they are not within the codebase of the labcript suite programs, and thus do not require a developer (editable) installation.

Customisations of the labscript suite will need to be reintegrated into the new package structure, using a developer installation. For example, to include your own custom labscript devices, you should undertake the developer installation procedure for the labscript-devices repository, and copy your custom or modified device files into the labscript_devices folder alongside the existing device files. Please also consider contributing these back to the main project by pushing them to your fork and issuing a pull request.

The procedure for migrating customisations of other components will depend on how up-to-date your fork is. Please open a thread on the mailing list to discuss with us how to migrate your custom features and/or how to contribute them back to the base *labscript suite* repositories.

Migrating other repositories to GitHub

Should you have other repositories on BitBucket such as labscriptlib, analysislib, userlib, or labconfig (or any project unrelated to the *labscript suite*) we strongly suggest using the tools we developed to migrate the *labscript suite* to GitHub. These are philipstarkey/bitbucket-hg-exporter and chrisjbillington/hg-export-tool which can be used together. See the documentation of those projects for further details.

1.2.9 *labscript suite* components

The *labscript suite* is modular by design, and is comprised of:

Table 1: Python libraries

	labscript — Expressive composition of hardware-timed experiments
	labscript-devices — Plugin architecture for controlling experiment hardware
	labscript-utils — Shared modules used by the *labscript suite*

Table 2: Graphical applications

	runmanager — Graphical and remote interface to parameterized experiments
	blacs — Graphical interface to scientific instruments and experiment supervision
	lyse — Online analysis of live experiment data
	runviewer — Visualize hardware-timed experiment instructions