Contents:

1 TCLab: Temperature Control Laboratory 1
   1.1 Installation ................................................................. 1
   1.2 Hardware setup .......................................................... 2
   1.3 Checking that everything works ..................................... 2
   1.4 Troubleshooting .......................................................... 2
   1.5 Next Steps ................................................................. 3
   1.6 Course Websites ......................................................... 3

2 Diagnose and Troubleshooting 5
   2.1 Problems and solutions .................................................. 5
   2.2 Software fixes ............................................................. 6

3 TCLab Overview 7
   3.1 TCLab Architecture ...................................................... 8
   3.2 Getting Started .......................................................... 8
   3.3 Next Steps ................................................................. 9

4 Accessing the Temperature Control Laboratory 11
   4.1 Importing tclab ........................................................... 11
   4.2 Using TCLab with Python's with statement ....................... 12
   4.3 Reading Temperatures .................................................. 12
   4.4 Setting Heaters ............................................................ 13
   4.5 Setting Maximum Heater Power ...................................... 14
   4.6 tclab Sampling Speed ................................................... 15

5 Synchronizing with Real Time 17
   5.1 Simple use of tclab.clock() ........................................... 17
   5.2 Optional Parameters .................................................... 17
   5.3 Using tclab.clock() with TCLab ................................... 19

6 TCLab Emulation for Offline Use 21
   6.1 Choosing Real or Emulation mode with setup() ................. 21

7 TCLab Historian 23
   7.1 Basic logging ............................................................. 23
   7.2 Accessing the Data Log from the Historian ..................... 24
   7.3 Saving to a file .......................................................... 25
7.4 Accessing log data using Pandas .................................................. 26
7.5 Specifying Sources for tclab.Historian ....................................... 26
7.6 Sessions ......................................................................................... 29
7.7 Persistence ..................................................................................... 30

8 TCLab Plotter .................................................................................... 31
  8.1 Specifying layout ........................................................................... 31

9 NotebookUI ....................................................................................... 33

10 Experiment Class ............................................................................. 35
  10.1 Accessing results ......................................................................... 35
  10.2 Even simpler experiments ............................................................. 36

11 The Labtime Class ............................................................................ 37
  11.1 Basic Usage .................................................................................. 37
  11.2 Advanced Usage .......................................................................... 39
  11.3 Auxiliary Functions .................................................................... 40

12 Search Page ..................................................................................... 43
TCLab: Temperature Control Laboratory

Master:

Development:

TCLab provides a Python interface to the Temperature Control Lab implemented on an Arduino microcontroller over a USB interface. TCLab is implemented as a Python class within the tclab package. The tclab package also includes:

- **clock** A Python generator for soft real-time implementation of process control algorithms.
- **Historian** A Python class to log results of a process control experiment.
- **Plotter** Provides an historian with real-time plotting within a Jupyter notebook.
- **TCLabModel** An embedded model of the temperature control lab for off-line and faster-than-realtime simulation of process control experiments. No hardware needs to be attached to use TCLabModel.

The companion Arduino firmware for device operation is available at the TCLab-Sketch repository.

The Arduino Temperature Control Lab is a modular, portable, and inexpensive solution for hands-on process control learning. Heat output is adjusted by modulating current flow to each of two transistors. Thermistors measure the temperatures. Energy from the transistor output is transferred by conduction and convection to the temperature sensor. The dynamics of heat transfer provide rich opportunities to implement single and multivariable control systems. The lab is integrated into a small PCB shield which can be mounted to any Arduino or Arduino compatible microcontroller.

## 1.1 Installation

Install using
pip install tclab

To upgrade an existing installation, use the command

```
pip install tclab --upgrade
```

The development version contains new features, but may be less stable. To install the development version use the command

```
pip install --upgrade https://github.com/jckantor/TCLab/archive/development.zip
```

### 1.2 Hardware setup

1. Plug a compatible Arduino device (UNO, Leonardo, NHduino) with the lab attached into your computer via the USB connection. Plug the DC power adapter into the wall.

2. (optional) Install Arduino Drivers

   * If you are using Windows 10, the Arduino board should connect without additional drivers required.
   * For Arduino clones using the CH340G, CH34G or CH34X chipset you may need additional drivers. Only install these if you see a message saying “No Arduino device found.” when connecting.
     * macOS.
     * Windows.

3. (optional) Install Arduino Firmware

   TCLab requires the one-time installation of custom firmware on an Arduino device. If it hasn’t been pre-installed, the necessary firmware and instructions are available from the TCLab-Sketch repository.

### 1.3 Checking that everything works

Execute the following code

```
import tclab
with tclab.TCLab() as lab:
  print(lab.T1)
```

If everything has worked, you should see the following output message

```
Connecting to TCLab
TCLab Firmware Version 1.2.1 on NHduino connected to port XXXX
21.54
TCLab disconnected successfully.
```

The number returned is the temperature of sensor T1 in °C.

### 1.4 Troubleshooting

If something went wrong in the above process, refer to our troubleshooting guide in TROUBLESHOOTING.md.
1.5 Next Steps

The notebook directory provides examples on how to use the TCLab module. The latest documentation is available at Read the Docs.

1.6 Course Websites

Additional information, instructional videos, and Jupyter notebook examples are available at the following course websites.

- Arduino temperature control lab page on the BYU Process Dynamics and Control course website.
- CBE 30338 for the Notre Dame Chemical Process Control course website.
- Dynamics and Control for notebooks developed at the University of Pretoria.
The library supplies a simple way to diagnose errors with the TCLab device in a function called `diagnose`, which is called as follows:

```python
from tclab import diagnose
diagnose()
```

This function will attempt to find the Arduino device, make a connection and attempt to exercise the full command set of the device to make sure everything is working correctly.

The above code can also be run from a terminal by using

```
python -m tclab
```

### 2.1 Problems and solutions

#### 2.1.1 No Arduino device found

1. First confirm that the device is correctly plugged in.
2. Plug it out and back in
3. Try a different port.
4. If no configuration has worked, you may need to install drivers (see below)

#### 2.1.2 Access denied

A device has been found but you get an error which mentions “Access denied”.

If you are using Windows, this can be resolved by going to Device Manager and selecting a different port for the device. If the device shows up incorrectly in the Device Manager, you may need to install drivers (see below)
2.1.3 Setting heaters makes temperature jump

You may have plugged both of the USB leads into one computer. The device works best when the barrel-ended jack is plugged into a separate power supply or a different computer.

2.1.4 Setting heater to 100 doesn’t raise temperature

You may only have plugged in your device into your computer using one cable. Your device needs to be plugged in to your computer and requires another connection to a power supply to power the heaters.

2.2 Software fixes

2.2.1 Install Drivers

If you are using Windows 10, the Arduino board should connect without additional drivers required.

For Arduino clones using the CH340G, CH34G or CH34X chipset you may need additional drivers. Only install these if you see a message saying “No Arduino device found.” when connecting.

- macOS
- Windows

2.2.2 Update Firmware

It is usually best to use the most recent version of the Arduino firmware, available from the TCLab-Sketch repository.

2.2.3 Update TCLab python library

If you find that the code supplied in the documentation gives errors about functions not being found, or if you installed tclab a long time ago, you need to update the TCLab library. This can be done with the command

```
pip install --update tclab
```
The tclab package provides a set of Python tools for interfacing with the BYU Temperature Control Laboratory. The Temperature Control Laboratory consists of two heaters and two temperature sensors mounted on an Arduino microcontroller board. Together, the tclab package and the Temperature Control Laboratory provide a low-cost experimental platform for implementing algorithms commonly used for process control.
3.1 TCLab Architecture

The tclab package is intended to be used as a teaching tool. The package provides high-level access to sensors, heaters, a pseudo-realtime clock. The package includes the following Python classes and functions:

- TCLab() providing access to the Temperature Control Laboratory hardware.
- TCLabModel() providing access to a simulation of the Temperature Control Laboratory hardware.
- clock for synchronizing with a real time clock.
- Historian for data logging.
- Plotter for realtime plotting.

Using these Python tools, students can create Jupyter notebooks and python codes covering a wide range of topics in process control.

- tclab.py: A Python package providing high-level access to sensors, heaters, a pseudo-realtime clock. The package includes TCLab() providing access to the device, clock for synchronizing with a real time clock, Historian for data logging, and Plotter for realtime plotting.
- TCLab-sketch.ino: Firmware for the intrinsically safe operation of the Arduino board and shield. The sketch is available at https://github.com/jckantor/TCLab-sketch.
- Arduino: Hardware platform for the Temperature Control Laboratory. TCLab is compatible with Arduino Uno, Arduino Leonardo, and compatible clones.

3.2 Getting Started

3.2.1 Installation

Install using
To upgrade an existing installation, use the command

```
pip install tclab --upgrade
```

The development version contains new features, but may be less stable. To install the development version use the command

```
pip install --upgrade https://github.com/jckantor/TCLab/archive/development.zip
```

### 3.2.2 Hardware Setup

1. Plug a compatible Arduino device (UNO, Leonardo, NHduino) with the lab attached into your computer via the USB connection. Plug the DC power adapter into the wall.

2. (optional) Install Arduino Drivers.

*If you are using Windows 10, the Arduino board should connect without additional drivers required.*

Mac OS X users may need to install a serial driver. For Arduino clones using the CH340G, CH34G or CH34X chipset, a suitable driver can be found [here](#) or [here](#).

3. (optional) Install Arduino Firmware;

TCLab requires the one-time installation of custom firmware on an Arduino device. If it hasn’t been pre-installed, the necessary firmware and instructions are available from the TCLab-Sketch repository.

### 3.2.3 Checking that everything works

Execute the following code

```python
import tclab
with tclab.TCLab() as lab:
    print(lab.T1)
```

If everything has worked, you should see the following output message

```
Connecting to TCLab
TCLab Firmware Version 1.2.1 on NHduino connected to port XXXX
21.54
TCLab disconnected successfully.
```

The number returned is the temperature of sensor T1 in °C.

### 3.3 Next Steps

The notebook directory provides examples on how to use the TCLab module. The latest documentation is available at [Read the Docs](#).
3.3.1 Course Web Sites

More information, instructional videos, and Jupyter notebook examples are available at the following course websites.

- Arduino temperature control lab page on the BYU Process Dynamics and Control course website.
- CBE 30338 for the Notre Dame Chemical Process Control course website.
- Dynamics and Control for notebooks developed at the University of Pretoria.
4.1 Importing tclab

Once installed the package can be imported into Python and an instance created with the Python statements

```python
import tclab
lab = tclab.TCLab()
```

TCLab provides access to temperature measurements, heaters, and LED on board the Temperature Control Laboratory. When called with no arguments, attempts to find a device connected to a serial port and returns a connection. An error is generated if no device is found. The connection should be closed with

```python
lab.close()
```

when no longer in use. The following cell demonstrates this process, and uses the tclab LED() function to flash the LED on the Temperature Control Lab for a period of 10 seconds at a 100% brightness level.

```bash
[3]: !pip install tclab
Collecting tclab
  Downloading https://files.pythonhosted.org/packages/b5/5a/751cc39b1b90c8a8fcce83d8a49b34572e6e8a22e6d6e4420bacbaa/tclab-0.4.6-py2.py3-none-any.whl
Collecting pyserial (from tclab)
  Downloading https://files.pythonhosted.org/packages/0d/e4/2a744dd9e3be04a0c0907414e2a01a7c88bb3915cbe3c8cc06e209f59c30/pyserial-3.4-py2.py3-none-any.whl (193kB)
  100% | | 194kB 4.1MB/s Remaining 0:00:01
Installing collected packages: pyserial, tclab
Successfully installed pyserial-3.4 tclab-0.4.6

[4]: import tclab
lab = tclab.TCLab()
```

(continues on next page)
4.1.1 A note on terminology

TCLab is a class. We instantiate the class by calling TCLab(). What is returned is an instance of the TCLab class. So in the above code, we would say lab is an instance of TCLab.

4.2 Using TCLab with Python’s with statement

The Python with statement provides a simple means of setting up and closing a connection to the Temperature Control Laboratory. The with statement establishes a context where a TCLab instance is created, assigned to a variable, and automatically closed upon completion.

```python
import tclab

with tclab.TCLab() as lab:
    lab.LED(100)

Arduino Leonardo connected on port /dev/cu.usbmodemWUART1 at 115200 baud.
TCLab Firmware 1.3.0 Arduino Leonardo/Micro.
TCLab disconnected successfully.
```

The with statement is likely to be the most common way to connect the Temperature Control Laboratory for most uses.

4.3 Reading Temperatures

Once a TCLab instance is created and connected to a device, the temperature sensors on the temperature control lab can be accessed with the attributes .T1 and .T2. For example, given an instance lab, the temperatures are accessed as

```python
T1 = lab.T1
T2 = a.T2
```

Note that lab.T1 and lab.T2 are read-only properties. Any attempt to set them to a value will return a Python error.

```python
import tclab

with tclab.TCLab() as lab:
    print("Temperature 1: {0:0.2f} °C".format(lab.T1))
    print("Temperature 2: {0:0.2f} °C".format(lab.T2))
```
4.4 Setting Heaters

The heaters are controlled by functions \texttt{.Q1()} and \texttt{.Q2()} of a TCLab instance. For example, both heaters can be set to 100% power with the functions

\begin{verbatim}
lab.Q1(100)
lab.Q2(100)
\end{verbatim}

The device firmware limits the heaters to a range of 0 to 100%. The current value of attributes may be accessed via

\begin{verbatim}
Q1 = lab.Q1()
Q2 = lab.Q2()
\end{verbatim}

Note that the retrieved values may be different due to the range-limiting enforced by the device firmware.

Alternatively, the heaters can also be specified with the properties \texttt{.U1} and \texttt{.U2}. Thus setting

\begin{verbatim}
lab.U1 = 100
lab.U2 = 100
\end{verbatim}

would set both heaters to 100% power. The current value of the heaters can be accessed as

\begin{verbatim}
print("Current setting of heater 1 is", lab.U1, ")")
print("Current setting of heater 2 is", lab.U2, ")")
\end{verbatim}

The choice to use a function (i.e., \texttt{.Q1()} and \texttt{.Q2()}) or a property (i.e., \texttt{.U1} or \texttt{.U2}) to set and access heater settings is a matter of user preference.

[4]:
\begin{verbatim}
import tclab
import time

with tclab.TCLab() as lab:
    print("Starting Temperature 1: {0:0.2f} °C".format(lab.T1), flush=True)
    print("Starting Temperature 2: {0:0.2f} °C".format(lab.T2), flush=True)

    lab.Q1(100)
    lab.Q2(100)
    print("Set Heater 1:", lab.Q1(), ")", flush=True)
    print("Set Heater 2:", lab.Q2(), ")", flush=True)

    t_heat = 20
    print("Heat for", t_heat, "seconds")
    time.sleep(t_heat)

    print("Turn Heaters Off")
    lab.Q1(0)
    lab.Q2(0)
    print("Set Heater 1:", lab.Q1(), ")", flush=True)
    print("Set Heater 2:", lab.Q2(), ")", flush=True)
\end{verbatim}

(continues on next page)
print("Final Temperature 1: {0:0.2f} °C".format(lab.T1))
print("Final Temperature 2: {0:0.2f} °C".format(lab.T2))

Arduino Leonardo connected on port /dev/cu.usbmodemFWAR1 at 115200 baud.
TCLab Firmware 1.3.0 Arduino Leonardo/Micro.

Starting Temperature 1: 27.67 °C
Starting Temperature 2: 27.03 °C

Set Heater 1: 100.0 %
Set Heater 2: 100.0 %

Heat for 20 seconds

Turn Heaters Off

Set Heater 1: 0.0 %
Set Heater 2: 0.0 %

Final Temperature 1: 28.96 °C
Final Temperature 2: 29.29 °C
TCLab disconnected successfully.

4.5 Setting Maximum Heater Power

The control inputs to the heaters power is normally set with functions .Q1() and .Q2() (or properties .U1 and .U2) specifying a value in a range from 0 to 100% of maximum heater power.

The values of maximum heater power are specified in firmware with values in the range from 0 to 255. The default values are 200 for heater 1 and 100 for heater 2. The maximum heater power can be retrieved and set by properties P1 and P2. The following code, for example, sets both heaters to a maximum power of 100.

[5]: import tclab
    with tclab.TCLab() as lab:
        print("Maximum power of heater 1 = ", lab.P1)
        print("Maximum power of heater 2 = ", lab.P2)
    
        print("Adjusting the maximum power of heater 1.")
        lab.P1 = 100
    
        print("Maximum power of heater 1 = ", lab.P1)
        print("Maximum power of heater 2 = ", lab.P2)

Arduino Leonardo connected on port /dev/cu.usbmodemFWAR1 at 115200 baud.
TCLab Firmware 1.3.0 Arduino Leonardo/Micro.
Maximum power of heater 1 = 200.0
Maximum power of heater 2 = 100.0
Adjusting the maximum power of heater 1.
Maximum power of heater 1 = 100.0
Maximum power of heater 2 = 100.0
TCLab disconnected successfully.

The actual power supplied to the heaters is a function of the power supply voltage applied to the Temperature Control Lab shield,
The maximum power applied to the heaters is a product of the settings \((P_1, P_2)\) and of the power supply used with the TCLab hardware. The TCLab hardware is normally used with a 5 watt USB power supply capable of supply up to 1 amp at 5 volts.

The TCLab hardware actually draws more than 1 amp when both \(P_1\) and \(P_2\) are set to 255 and \(Q_1\) and \(Q_2\) are at 100\%. This situation will overload the power supply and result in the power supply shutting down. Normally the power supply will reset itself after unplugging from the power mains.

Experience with the device shows keeping the sum \(P_1\) and \(P_2\) to a value less than 300 will avoid problems with the 5 watt power supply. If you have access to larger power supplies, then you can adjust \(P_1\) and \(P_2\) accordingly to achieve a wider range of temperatures.

4.6 tclab Sampling Speed

There are limits to how quickly the board can be sampled. The following examples show values for a particular type of board. You can run them to see how quick your board is.

4.6.1 Temperature Sampling Speed

```python
[6]: import time
    import tclab

    TCLab = tclab.setup(connected=True)

    N = 100
    meas = []
    with TCLab() as lab:
        tic = time.time()
        for k in range(0,N):
            meas.append(lab.T1)
        toc = time.time()

    print('Reading temperature at', round(N/(toc-tic),1), 'samples per second."

Arduino Leonardo connected on port /dev/cu.usbmodem1234 at 115200 baud.
TCLab Firmware 1.3.0 Arduino Leonardo/Micro.
TCLab disconnected successfully.
Reading temperature at 12.3 samples per second.
```

4.6.2 Heater Sampling Speed

```python
[7]: import time
    import tclab

    TCLab = tclab.setup(connected=True)

    N = 100
    with TCLab() as lab:
        tic = time.time()
        for k in range(0,N):
            lab.Q1(100)
        toc = time.time()
```

(continues on next page)
print('Setting heater at', round(N/(toc-tic),1), 'samples per second.\n')

Arduino Leonardo connected on port /dev/cu.usbmodemFWAR1 at 115200 baud.
TCLab Firmware 1.3.0 Arduino Leonardo/Micro.
TCLab disconnected successfully.
Setting heater at 8.2 samples per second.
Synchronizing with Real Time

5.1 Simple use of tclab.clock()

The tclab module includes a function clock for synchronizing calculations with real time. clock(period) is an iterator that generates a sequence of equally spaced time steps from zero to period separated by one second intervals. For each step clock returns time since start rounded to the nearest 10th of a second.

```python
import tclab

period = 5
for t in tclab.clock(period):
    print(t, "sec.")
```

0 sec.
1.0 sec.
2.0 sec.
3.0 sec.
4.0 sec.

tclab.clock() is implemented as a Python generator. A consequence of this implementation is that tclab.clock() is ‘blocking’ which limits its use for creating interactive demonstrations. See later sections of this user’s guide for non-blocking alternatives that can be used for interactive demonstrations or GUI’s.

5.2 Optional Parameters

5.2.1 step: Clock time step

An optional parameter step specifies a time step different from one second.

```python
import tclab

period = 5
for t in tclab.clock(period, step=0.5):
    print(t, "sec.")
```

0.5 sec.
1.0 sec.
1.5 sec.
2.0 sec.
2.5 sec.
3.0 sec.
3.5 sec.
4.0 sec.
4.5 sec.

The step parameter defines the time step to be used instead of one second. This allows for creating time steps that are more suitable for different applications.
period = 5
step = 2.5
for t in tclab.clock(period, step):
    print(t, "sec.")
0 sec.
2.5 sec.

5.2.2 tol: clock tolerance

There are some considerations when using `clock`. First, by its nature Python is not a real-time environment. `clock` makes a best effort to stay in sync with the wall clock but there can be no guarantees. The default behavior of `clock` is to maintain long-term synchronization with the real time clock.

The `tol` argument specifies the allowable error on time steps. By default it is 0.5 seconds.

The following cell demonstrates the effect of an intermittent calculation that exceeds the time step specified by `step`. In this instance, a `sleep` timeout of 2 seconds occurs at t=2. The default behaviour is to raise an error when desynchronization occurs.

```python
[3]: import tclab
import time
period = 5
step = 1
for t in tclab.clock(period, step):
    print(t, "sec.")
    if 1.9 < t < 2.5:
        time.sleep(2)
```

```sql
0 sec.
1.0 sec.
2.0 sec.
```

```
RuntimeError Traceback (most recent call last)
<ipython-input-3-10f7d48b2c2a> in <module>()
      5 step = 1
      6 --> 7 for t in tclab.clock(period, step):
            8     print(t, "sec.")
            9     if 1.9 < t < 2.5:
~/Documents/Development/TCLab/tclab/labtime.py in clock(period, step, tol, adaptive)
      102     "Step size was {} s, but {:.2f} s elapsed “
      103     "((:.2f) too long). Consider increasing step.’)
--> 104     raise RuntimeError(message.format(step, elapsed, elapsed-step))
      105     labtime.sleep(step - (labtime.time() - start) % step)
      106     now = labtime.time() - start

RuntimeError: Labtime clock lost synchronization with real time. Step size was 1 s, but 2.01 s elapsed (1.01 too long). Consider increasing step.
```

We can avoid the error above by specifying a larger value of `step` as advised, or we can specify a larger value for `tol`. Note that now time steps are skipped.
5.3 Using `tclab.clock()` with TCLab

An important use of the `tclab.clock()` generator is to implement and test control and estimation algorithms. The following cell shows how the `clock` generator can be used within the context defined by the Python `with` statement.

```python
import tclab

period = 20
step = 2

with tclab.TCLab() as lab:
    lab.Q1(100)
    lab.Q2(100)

    print("\nSet Heater 1 to {0:f} %".format(lab.Q1()))
    print("Set Heater 2 to {0:f} %\n".format(lab.Q2()))

    sfmt = "  {0:5.1f} sec: T1 = {1:0.1f} °C  T2 = {2:0.1f} °C"

    for t in tclab.clock(period, step):
        print(sfmt.format(t, lab.T1, lab.T2), flush=True)
```

Arduino Leonardo connected on port /dev/cu.usbmodemFWAR1 at 115200 baud. TCLab Firmware 1.3.0 Arduino Leonardo/Micro.

Set Heater 1 to 100.000000 %
Set Heater 2 to 100.000000 %

0.0 sec:  T1 = 30.9 °C  T2 = 33.1 °C
2.0 sec:  T1 = 30.9 °C  T2 = 33.5 °C
4.0 sec:  T1 = 30.9 °C  T2 = 33.1 °C
6.0 sec:  T1 = 30.6 °C  T2 = 32.2 °C
8.0 sec:  T1 = 30.9 °C  T2 = 33.1 °C
10.0 sec: T1 = 30.9 °C  T2 = 33.1 °C
12.0 sec: T1 = 30.9 °C  T2 = 33.5 °C
14.0 sec: T1 = 30.9 °C  T2 = 33.8 °C
16.0 sec: T1 = 31.2 °C  T2 = 32.8 °C
18.0 sec: T1 = 31.2 °C  T2 = 34.4 °C
20.0 sec: T1 = 31.5 °C  T2 = 34.8 °C

TCLab disconnected successfully.
TCLab Model replaces TCLab for occasions where the TCLab hardware might not be available. To use, include the import

```
from tclab import TCLabModel as TCLab
```

The rest of your code will work without change. Be advised the underlying model used to approximate the behavior of the Temperature Control Laboratory is an approximation to the dynamics of the actual hardware.

```
[1]: from tclab import TCLabModel as TCLab

with TCLab() as a:
    print("Temperature 1: {0:0.2f} °C"\n          .format(a.T1))
    print("Temperature 2: {0:0.2f} °C"\n          .format(a.T2))
```

TCLab version 0.4.5dev
Simulated TCLab
Temperature 1: 20.95 °C
Temperature 2: 20.95 °C
TCLab Model disconnected successfully.

6.1 Choosing Real or Emulation mode with setup()

The `tclab.setup()` function provides a choice of using actual hardware or an emulation of the TCLab device by changing a single line of code. When emulating TCLab, a second parameter `speedup` allows the emulation to run at a multiple of real time.

```
# connect to TCLab mounted on arduino
TCLab = tclab.setup(connected=True)

# Emulate the operation of TCLab using TCLabModel
TCLab = tclab.setup(connected=False)
```

(continues on next page)
The next cell demonstrates emulation of the TCLab device at 5× real time.

```python
[2]: %matplotlib inline
import tclab

TCLab = tclab.setup(connected=False, speedup=5)

with TCLab() as lab:
    for t in tclab.clock(20):
        lab.Q1(100 if t < 10 else 0)
        print("t = {0:4.1f} Q1 = {1:3.0f} % T1 = {2:5.2f}".format(t, lab.Q1(), lab.T1))
```

TCLab version 0.4.5dev
Simulated TCLab

<table>
<thead>
<tr>
<th>t</th>
<th>Q1</th>
<th>T1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>100%</td>
<td>20.95</td>
</tr>
<tr>
<td>1.0</td>
<td>100%</td>
<td>20.95</td>
</tr>
<tr>
<td>2.0</td>
<td>100%</td>
<td>20.95</td>
</tr>
<tr>
<td>3.0</td>
<td>100%</td>
<td>20.95</td>
</tr>
<tr>
<td>4.0</td>
<td>100%</td>
<td>20.95</td>
</tr>
<tr>
<td>5.0</td>
<td>100%</td>
<td>21.27</td>
</tr>
<tr>
<td>6.1</td>
<td>100%</td>
<td>21.27</td>
</tr>
<tr>
<td>7.0</td>
<td>100%</td>
<td>21.27</td>
</tr>
<tr>
<td>8.1</td>
<td>100%</td>
<td>21.59</td>
</tr>
<tr>
<td>9.0</td>
<td>100%</td>
<td>21.59</td>
</tr>
<tr>
<td>10.1</td>
<td>0</td>
<td>21.92</td>
</tr>
<tr>
<td>11.2</td>
<td>0</td>
<td>21.92</td>
</tr>
<tr>
<td>12.2</td>
<td>0</td>
<td>22.24</td>
</tr>
<tr>
<td>13.2</td>
<td>0</td>
<td>22.24</td>
</tr>
<tr>
<td>14.0</td>
<td>0</td>
<td>22.56</td>
</tr>
<tr>
<td>15.2</td>
<td>0</td>
<td>22.88</td>
</tr>
<tr>
<td>16.2</td>
<td>0</td>
<td>22.88</td>
</tr>
<tr>
<td>17.2</td>
<td>0</td>
<td>22.88</td>
</tr>
<tr>
<td>18.2</td>
<td>0</td>
<td>23.21</td>
</tr>
<tr>
<td>19.0</td>
<td>0</td>
<td>23.21</td>
</tr>
<tr>
<td>20.1</td>
<td>0</td>
<td>23.21</td>
</tr>
</tbody>
</table>

TCLab Model disconnected successfully.
CHAPTER 7

TCLab Historian

7.1 Basic logging

The tclab.Historian class provides data logging. Given an instance of a TCLab object, an historian is created with the commands

```python
import tclab
lab = tclab.TCLab()
h = tclab.Historian(lab.sources)
```

The historian initializes a data log. The sources for the data log are specified in the argument to tclab.Historian. A default set of sources for an instance lab is given by lab.sources. The specification for sources is described in a later section.

The data log is updated by issuing a command

```python
h.update(t)
```

Where \( t \) is the current clock time. If \( t \) is omitted the historian will calculate its own time.

```python
[1]: import tclab
   TCLab = tclab.setup(connected=False, speedup=10)
   with TCLab() as lab:
       h = tclab.Historian(lab.sources)
       for t in tclab.clock(20):
           lab.Q1(100 if t <= 10 else 0)
           print("Time:", t, \'seconds\')
           h.update(t)
```

TCLab version 0.4.7dev
Simulated TCLab
Time: 0 seconds
Time: 1.1 seconds

(continues on next page)
7.2 Accessing the Data Log from the Historian

Historian maintains a data log that is updated on each encounter of the .update() function. The list of variables logged by an Historian is given by

```
[2]: h.columns
```

```
[2]: ['Time', 'T1', 'T2', 'Q1', 'Q2']
```

Individual time series are available as elements of Historian.fields. For the default set of sources, the time series can be obtained as

```
t, T1, T2, Q1, Q2 = h.fields
```

For example, here's how to plot the history of temperature T1 versus time from the example above.

```
%matplotlib notebook

import matplotlib.pyplot as plt

t, T1, T2, Q1, Q2 = h.fields
plt.plot(t, T1)
plt.xlabel('Time / seconds')
plt.ylabel('Temperature / °C')
plt.grid()

<IPython.core.display.Javascript object>
<IPython.core.display.HTML object>
```

7.2.1 Accessing fields by name

To enable easy access of a particular field by name, Historian allows individual field access via the logdict dictionary. It also has a t property for quick access of the time field, so the above code could also have been written
this way:

```python
%matplotlib notebook

import matplotlib.pyplot as plt

column = 'T1'
plt.plot(h.t, h.logdict['T1'])
plt.xlabel('Time / seconds')
plt.ylabel('Temperature / °C')
plt.grid()
```

Here is a simple code using these access methods to plot all the fields an Historian is logging:

```python
def plotlog(historian):
    t = historian.t
    fig, axes = plt.subplots(nrows=len(historian.fields) - 1, ncols=1, sharex='col')
    for axis, column in zip(axes, historian.columns[1:]):
        axis.step(t, historian.logdict[column], where='post')
        axis.grid()
        axis.set_ylabel(column)
    plt.xlabel('Time / Seconds')

plotlog(h)
```

### 7.2.2 Tabular access

The entire data history is available from the historian as the attribute `.log`. Here we show the first three rows from the log. The columns are returned in the same order as `.columns`.

```python
h.columns
# ['Time', 'T1', 'T2', 'Q1', 'Q2']

h.log[:3]
# [(0, 20.949499999999997, 20.6272, 100, 0),
#  (1.1, 20.949499999999997, 20.949499999999997, 100, 0),
#  (2.0, 20.949499999999997, 20.949499999999997, 100, 0)]
```

### 7.3 Saving to a file

The log can be saved to a CSV file using `.to_csv()`. This is useful for spreadsheet access.

```python
h.to_csv('saved_data.csv')
```
7.4 Accessing log data using Pandas

Pandas is a widely use Python library for manipulation and analysis of data sets. Here we show how to access the tclab.Historian log using Pandas.

The log can be converted to a Pandas dataframe.

```python
import pandas as pd

df = pd.DataFrame.from_records(h.log, columns=h.columns, index='Time')
df.head()
```

<table>
<thead>
<tr>
<th>Time</th>
<th>T1</th>
<th>T2</th>
<th>Q1</th>
<th>Q2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>20.9495</td>
<td>20.6272</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>1.1</td>
<td>20.9495</td>
<td>20.9495</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>2.0</td>
<td>20.9495</td>
<td>20.9495</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>3.1</td>
<td>20.9495</td>
<td>20.9495</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>4.1</td>
<td>20.9495</td>
<td>20.9495</td>
<td>100</td>
<td>0</td>
</tr>
</tbody>
</table>

The following cells provide examples of plots that can be constructed once the data log has been converted to a pandas dataframe.

```python
df.plot()
```

```python
df[['T1','T2']].plot(grid=True)
```

7.5 Specifying Sources for tclab.Historian

To create an instance of tclab.Historian, a set of sources needs to be specified. For many cases the default sources created for an instance of TCLab is sufficient. However, it is possible to specify additional sources which can be useful when implementing more complex algorithms for process control.

sources is specified as a list of tuples. Each tuple as two elements. The first element is a label for the source. The second element is a function that returns a value.

The following cell shows how to create a source with the label Power with a value equal to the estimated heater power measured in watts. This is created on the assumption that 100% of a maximum power of 200 corresponds to 4.2 watts.

```python
import tclab

tclab = tclab.setup(connected=False, speedup=10)

with tclab() as lab:
    sources = [
        ('T1', lambda: lab.T1),
    ]
```
('Power', lambda: lab.P1*lab.U1*4.2/(200*100))
]
h = tclab.Historian(sources)
for t in tclab.clock(20):
    lab.Q1(100 if t <= 10 else 0)
    print("Time:", t, 'seconds')
    h.update(t)

[13]: import pandas as pd

    df = pd.DataFrame.from_records(h.log, columns=h.columns, index='Time')
    df.head()

    T1 Power
    Time
    0.0 20.9495 4.2
    1.0 20.9495 4.2
    2.0 20.9495 4.2
    3.0 20.9495 4.2
    4.1 20.9495 4.2

7.5.1 Functions with multiple returns

In some cases it is easier to calculate a number of different variables to be logged in one function, especially if intermediate results are used in following calculations. This can be accommodated by Historian by passing None as the function for subsequent values if a previous value returned a list of values.

[14]: import tclab

    TCLab = tclab.setup(connected=False, speedup=10)
```python
def log_values():
    T1 = lab.T1
    T1Kelvin = T1 + 273.15
    power = lab.P1*lab.U1*4.2/(200*100)
    return T1, T1Kelvin, power

with TCLab() as lab:
    h = tclab.Historian([('T1', log_values),
                         ('T1Kelvin', None),
                         ('Power', None)])
    for t in tclab.clock(20):
        lab.Q1(100 if t <= 10 else 0)
        print("Time": t, 'seconds')
        h.update(t)
```

TCLab version 0.4.7dev
Simulated TCLab
Time: 0 seconds
Time: 1.0 seconds
Time: 2.0 seconds
Time: 3.0 seconds
Time: 4.0 seconds
Time: 5.0 seconds
Time: 6.0 seconds
Time: 7.2 seconds
Time: 8.0 seconds
Time: 9.2 seconds
Time: 10.1 seconds
Time: 11.0 seconds
Time: 12.2 seconds
Time: 13.0 seconds
Time: 14.0 seconds
Time: 15.0 seconds
Time: 16.1 seconds
Time: 17.0 seconds
Time: 18.1 seconds
Time: 19.1 seconds
Time: 20.0 seconds
TCLab Model disconnected successfully.

```
[15]: import pandas as pd

df = pd.DataFrame.from_records(h.log, columns=h.columns, index='Time')
df.head()

[15]:
       T1   T1Kelvin  Power
    Time
    0.0  20.9495  294.0995  4.2
    1.0  20.9495  294.0995  4.2
    2.0  20.9495  294.0995  4.2
    3.0  20.9495  294.0995  4.2
    4.0  20.9495  294.0995  4.2

[16]: h.log
```
It is possible to run multiple experiments using the same Historian and page back to them after running them:

```python
import tclab

TCLab = tclab.setup(connected=False, speedup=10)

with TCLab() as lab:
    h = tclab.Historian(lab.sources)
    for t in tclab.clock(20):
        lab.Q1(100 if t <= 10 else 0)
        h.update(t)
    h.new_session()
    for t in tclab.clock(20):
        lab.Q1(0 if t <= 10 else 100)
        h.update(t)

TCLab version 0.4.7dev
Simulated TCLab
TCLab Model disconnected successfully.

To see the stored sessions, use get_sessions:

```python
h.get_sessions()
```

```python
[(1, '2018-03-18 08:10:02', 21), (2, '2018-03-18 08:10:03', 21)]
```

The historian log shows the data for the last session, where the heater started open

```python
h.log[:2]
```

```python
[(0, 23.205599999999997, 20.949499999999997, 0, 0),
 (1.1, 23.205599999999997, 20.949499999999997, 0, 0)]
```
To roll back to a different session, use `load_session()`:

```
[20]: h.load_session(1)
[21]: h.log[:2]
[21]: [(0, 20.949499999999997, 20.949499999999997, 100, 0), (1.0, 20.949499999999997, 20.949499999999997, 100, 0)]
```

### 7.7 Persistence

Historian stores results in a SQLite database. By default, it uses the special file `:memory:` which means the results are lost when you shut down or restart the kernel. To retain values across runs, you can specify a filename (by convention SQLite databases have the `.db` extension).

```
[22]: import tclab
    TCLab = tclab.setup(connected=False, speedup=10)
with TCLab() as lab:
    h = tclab.Historian(lab.sources, dbfile='test.db')
    for t in tclab.clock(20):
        lab.Q1(100 if t <= 10 else 0)
        h.update(t)
```

TCLab version 0.4.7dev
Simulated TCLab
TCLab Model disconnected successfully.

Every time you run this cell, new sessions will be added to the file. These sessions can be loaded as shown in the Sessions section above. There is currently no support for managing sessions. If you want to remove the old sessions, delete the database file.
When operating in a Jupyter Notebook, a `Plotter` can be used together with the `Historian`.

```python
h = Historian(lab)
p = Plotter(h, tfinal)
```

where `lab` is a TCLab instance as before and the optional parameter `tfinal` provides an initial scaling of the time axes. Each call to `p.update()` will automatically update both the historian and the plot.

```python
%matplotlib notebook
from tclab import TCLab, clock, Historian, Plotter, setup

TCLab = setup(connected=False, speedup=10)

with TCLab() as lab:
    h = Historian(lab.sources)
    p = Plotter(h, twindow=200)
    for t in clock(200):
        lab.Q1(100 if t < 100 else 0)
        p.update(t)

Simulated TCLab

<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>

TCLab Model disconnected successfully.

### 8.1 Specifying layout

The layout of values can be specified when creating a `Plotter`. Layout is given as a tuple of tuples. Each of the first level tuples creates a new axis, and each of the elements in the tuple is plotted on that same axis.
import time

tic = time.time()
TCLab = setup(connected=False, speedup=10)

with TCLab() as lab:
    h = Historian(lab.sources)
    p = Plotter(h, 200, layout=((T1', 'T2'), ('Q1', 'Q2')))
    for t in clock(200):
        lab.U1 = 80
        p.update(t)

toc = time.time()

print(toc-tic, 'seconds')

Simulated TCLab

<TIpython.core.display.Javascript object>
<TIpython.core.display.HTML object>

TCLab Model disconnected successfully.
20.255380153656006 seconds
The `tclab.gui` module supplies a graphical interface to the Temperature Control Laboratory.

```python
[1]: from tclab.gui import NotebookUI
    %matplotlib notebook
    interface = NotebookUI()
```

```python
[2]: interface.gui

A Jupyter Widget

Simulated TCLab

<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>
```

[ ]:

[ ]:
The `Experiment` class provides an all-in-one interface for running simple experiments. It combines the effect of instantiating a TCLab, a Historian and optionally a Plotter.

```python
[8]: import tclab
   with tclab.Experiment(connected=False, plot=True, speedup=10) as experiment:
     for t in experiment.clock():
       experiment.lab.Q1(0 if experiment.lab.T1 > 40 else 10)
```

10.1 Accessing results

`experiment.historian` contains the historian instance which was created by `Experiment`:

```python
[14]: experiment.historian.log[:5]

[14]: [(0, 20.812700940382776, 20.788384634613717, 0, 0),
      (1.0, 20.905995814379526, 20.978928790222188, 100, 0),
      (2.0, 21.245346825807175, 21.23492347886724, 100, 0),
      (3.0, 21.071910253650344, 21.1795991571187, 100, 0),
      (4.0, 21.11707641600787, 20.841145497515168, 100, 0)]
```
10.2 Even simpler experiments

Sometimes you can use a simple function to describe the experiment you are running. In this case, it is simpler to use runexperiment. Experiment and runexperiment share arguments.

```
[9]: from tclab import runexperiment

[10]: %matplotlib notebook

[11]: def onoffcontroller(t, lab):
    lab.Q1(0 if lab.T1 > 40 else 100)

[12]: experiment = runexperiment(onoffcontroller, connected=False, time=100, speedup=10)
```

Simulated TCLab

<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>

TCLab Model disconnected successfully.
The Labtime class is a tool for speeding up the simulation of process control experiments. With this tool you can more quickly develop control algorithms through simulation, then apply the algorithms to the Temperature Control Lab device with minimal changes to your code.

In most cases you do not need to directly invoke Labtime. For example, setup with the optional parameter speedup (described in the chapter TCLab Simulation for Offline Use) uses Labtime to adjust the operation of the clock iterator. This is sufficient for many applications.

## 11.1 Basic Usage

### 11.1.1 .time()

Labtime provides a replacement for the time.time() function from the Python standard library. The basic usage is demonstrated in the following cell. Note that import brings in an instance of Labtime. labtime.time() returns the lab time elapsed since first imported into the Python kernal.

```python
from tclab import labtime

tic = labtime.time()
labtime.sleep(2)
toc = labtime.time()

print("Time since first imported = ", round(tic, 2), " labtime seconds.")
print("Time since first imported = ", round(toc, 2), " labtime seconds.")
```

Time since first imported = 0.0 labtime seconds.
Time since first imported = 2.01 labtime seconds.

By default, labtime.time() progresses at the same rate at real time as measured by the Python time package. The following cell demonstrates the default correspondence of labtime and real time.
11.1.2 .set_rate(rate) and .get_rate(rate)

Lab time can proceed at a rate faster or slower than real time. The relative rate of lab time to real time is set with the labtime.set_rate(rate). The default value is one. The current value of the rate is returned by the get_rate().
As demonstrated, conceptually you can think of lab time as a piecewise linear function of real time with the following properties:

- monotonically increasing
- continuous
- shared by all functions using `labtime`.

### 11.1.3 `sleep(delay)`

The `labtime.sleep()` function suspends execution for a period `delay` in lab time units. This is used, for example, in the `clock` iterator to speed up execution of a control loop when used in simulation mode.

### 11.2 Advanced Usage

An additional set of functions are available in `Labtime` to facilitate construction of GUI’s, and for programmatically creating code to simulate the behavior of more complex control systems.

#### 11.2.1 `reset(t)`

The `labtime.reset(t)` method resets lab time to `t` (default 0). The function `setnow(t)` provides an equivalent service, and is included to provide backward compatibility early versions of `tclab`. This function is typically used within a GUI for repeated testing and tuning of a control algorithm.

```
[4]: from tclab import labtime

print("Resetting lab time to zero.")
labtime.reset(0)
print("labtime =", labtime.time(),"\n")

print("Resetting lab time to ten.")
labtime.reset(10)
print("labtime =", labtime.time(),"\n")

Resetting lab time to zero.
labtime = 3.2901763916015625e-05

Resetting lab time to ten.
labtime = 10.000032901763916
```

#### 11.2.2 `stop()` / `start()` / `running`

`labtime.stop()` freezes the labtime clock at its current value. A Runtime warning is generated if there is an attempt to sleep while the labtime is stopped.

`labtime.start()` restarts the labtime clock following a stoppage.

`labtime.running` is a Boolean value that is `True` if the labtime clock is running, otherwise it is `False`. 
from tclab import labtime

print("Is labtime running?", labtime.running)
print("labtime =", labtime.time(), ")

print("Now we'll stop the labtime.")
labtime.stop()
print("Is labtime running?", labtime.running, ")

print("We'll pause for 2 seconds in real time.")
time.sleep(2)

print("We'll restart labtime and pick up where we left off.")
labtime.start()
print("labtime =", labtime.time())

Is labtime running? True
labtime = 10.015635967254639

Now we’ll stop the labtime.
Is labtime running? False

We’ll pause for 2 seconds in real time.

We’ll restart labtime and pick up where we left off.
labtime = 10.015910148620605

11.3 Auxiliary Functions

11.3.1 clock(tperiod, tstep)

The clock iterator was introduced in an earlier section on synchronizing tclab with real time. In fact, clock uses the Labtime class to coordinate with real time, and to provide faster than real time operation in simulation mode.

from tclab import labtime, clock

import time
time_start = time.time()
labtime_start = labtime.time()

def do(n):
    print("\nRate =", labtime.get_rate())
    for t in clock(n):
        t_real = time.time() - time_start
        t_lab = labtime.time() - labtime_start
        print("real time = {0:4.1f} lab time = {1:4.1f}".format(t_real, t_lab))

labtime.set_rate(1)
do(5)
labtime.set_rate(10)
do(5)
11.3.2 setnow(t)

setnow(t) performs the same function as labtime.reset(t). This function appeared in an early version of tclab, and is included here for backwards compatibility.
CHAPTER 12

Search Page

- search