Laboratory Documentation

Release 0.1

Sam Jennings

Feb 07, 2019

CONTENTS:

CHAPTER

ONE

MODULE LABORATORY

class laboratory.**Setup**(*filename=None*, *debug=False*)

Sets up the laboratory

Attributes	Description
data	houses lab data during measurements or after parseing
logger	creates a logger for error reporting
plot	contains different plotting tools for data visualisation
command	contains the drivers for controlling instrumentation

Public Methods	Description
device_status	checks the status of all connected devices
get_gas	retrieves and saves data from a single mass flow controller
get_impedance	retrieves and saves complex impedance data from the LCR meter
get_temp	retrieves and saves temperature data from the furnace
get_thermo	retrieves and saves thermopower data from the DAQ
load_data	will parse a datafile and store in 'data' structure for post- processing and visualisation
load_frequencies	loads a set of frequencies into Data object
load_instruments	connects to all available instruments
run	begins a new set of laboratory measurements

Private Methods	Description
_break_loop	determines when the main measurement loop should break and start a new step
_count_down	controls the count down timer displayed between measurements
_measurement_loop	main measurement loop of the program that retrieves and saves data
_progress_bar	controls the progress bar displayed during measurements

Example

```
>>> import Laboratory
>>> lab = Laboratory.Setup()
>>> lab.run('some_controlfile')
```

_break_loop(*step*, *loop_start*)

Checks whether the main measurements loop should be broken in order to proceed to the next step. If temperature is increasing the loop will break once T-indicated exceeds the target temperature. If temperature is decreasing, the loop will break when T-indicated is within 5 degrees of the target. If temperature is holding, the loop will break when the hold time specified by step.hold_length is exceeded.

Parameters

- **step** – current measurement step
- **Tind** (*float*) – current indicated temperature on furnace
- **loop_start** (*float (time.time())*) – start time of the current measurement cycle

_count_down(*start*, *interval*, *time_remaining=1*)

Controls the count down until next measurement cycle

Parameters **interval** (*float/int*) – time in seconds remaining until next measurement

_measurement_loop(*step*)

This is the main measurement loop of the program. All data is accessed and saved from within this loop

Parameters **step** – a single row from the control file

_progress_bar(*iteration*, *message*, *decimals=0*, *bar_length=25*)

Creates a terminal progress bar

Parameters

- **iteration** – iteration number
- **message** (*str*) – message to be displayed on the right of the progress bar

delayed_start

Starts the experiment at a given time the next day. Can be set to any 24 hour time in string format.

Example

```
>>> import Laboratory
>>> lab = Laboratory.Setup()
>>> lab.delayed_start = '0900' #start at 9am the following day
>>> lab.run('some_controlfile')
```

device_status()

Checks the status of all devices. If desired, this function can send an email when something has become disconnected

Returns True if all devices are connected and False if any are disconnected

Return type Boolean

get_gas(*gas_type*)

Gets data from the mass flow controller specified by gas_type and saves to Data structure and file

Parameters **gas_type** (*str*) – type of gas to use when calculating ratio (either ‘h2’ or ‘co’)

Returns [mass_flow, pressure, temperature, volumetric_flow, setpoint]

Return type list

get_impedance()

Sets up the lcr meter and retrieves complex impedance data at all frequencies specified by Data.freq. Data is saved in Data.imp.z and Data.imp.theta as a list of length Data.freq. Values are also saved to the data file.

get_temp(*target*)

Retrieves the indicated temperature of the furnace and saves to Data structure and file

Note: this is the temperature indicated by the furnace, not the temperature of the sample

Parameters **target** (*float*) – target temperature of current step

get_thermopower()

Retrieves thermopower data from the DAQ and saves to Data structure and file

Returns [thermistor, te1, te2, voltage]

Return type list

load_data(*filename*)

loads a previous data file for processing and analysis

Parameters **filename** (*str*) – path to data file

load_frequencies(*min=20*, *max=2000000*, *n=50*, *log=True*, *filename=None*)

Loads an np.array of frequency values specified by either min, max and n or a file containing a list of frequencies specified by filename.

Parameters

- **filename** (*str*) – name of file containing frequencies
- **n** (*int*) – number of desired frequencies
- **min** (*int, float*) – minimum frequency (Hz) - may not be below default value of 20 Hz
- **max** (*int, float*) – maximum frequency (Hz) - may not exceed default value of 2*10^6 Hz
- **log** (*boolean*) – specifies whether array is created in linear or log space. default to logspace

Example

```
>>> lab = Laboratory.Setup()
>>> lab.load_frequencies(min=1000, max=10000, n=10)
>>> print(lab.data.freq)
[1000 2000 3000 4000 5000 6000 7000 8000 9000 10000]
>>> lab.load_frequencies(min=1000, max=10000, n=10, log=True)
>>> print(lab.data.freq)
[1000 1291.55 1668.1 2154.43 2782.56 3593.81 4641.59 5994.84 7742.64 10000]
```

load_instruments()

Loads all the laboratory instruments. Called automatically when calling Setup() without a filename specified.

Returns lcr, daq, mfc, furnace, motor

Return type instrument objects

preflight_checklist(*controlfile*)

Conducts necessary checks before running an experiment abs

Parameters **controlfile** (*string*) – name of control file for the experiment

reconnect()

Attempts to reconnect to any instruments that have been disconnected

restart_from_backup()

TODO - reload an aborted experiment and pick up where it left off

run(*controlfile=False*)

starts a new set of measurements. requires a control file that contains specific instruction for the instruments to follow. see the tutorial section for help setting up a control file.

Parameters **controlfile** (*str*) – path to control file

save_data(*val_type*, *vals*, *gastype=None*)

Takes input values and saves to both the current Data object and an external file

Parameters

- **val_type** (*str*) – type of measurement being saved
- **vals** (*list*) – the required values suitable to that specified by val_type
- **gastype** (*str*) – [optional] required when saving gas data

set_fugacity(*buffer*, *offset*, *gas_type*)

Sets the correct gas ratio for the given buffer. Percentage offset from a given buffer can be specified by 'offset'. Type of gas to be used for calculations is specified by gas_type.

Parameters

- **buffer** (*str*) – buffer type (see table for input options)
- **offset** (*float, int*) – percentage offset from specified buffer
- **gas_type** – gas type to use for calculating ratio - can be either 'h2' or 'co'

shut_down()

Returns the furnace to a safe temperature and closes ports to both the DAQ and LCR. (TODO need to close ports to motor and furnace)

CHAPTER

TWO

MODULE DRIVERS

Contains drivers for each instrument in use in the laboratory.

class drivers.**AlicatController**(*port*, *address*)

Driver for an individual Mass Flow Controller.

> **Note:** not called directly - access is from within :class:'~Drivers.MFC'

Methods	message
get_massflow	gets massflow from controller
set_massflow	sets massflow on controller
get_pressure	gets pressure from controller
set_pressure	sets massflow on controller
get_temp	gets pressure from controller
get_vol_flow	gets volumetric flow from controller
get_setpoint	gets current set point from controller
reset	resets the device

*see FlowController for more methods

massflow(*value=None*)

Get or set the massflow of the appropriate flowmeter

Parameters **value** (*float*) – desired massflow value

pressure(*value=None*)

Get or set pressure of the appropriate flowmeter

Parameters **value** (*float*) – desired massflow value

reset()

sets the massflow to 0

setpoint()

Gets the current set point of the appropriate flowmeter

temperature()

Gets the temperature of the appropriate flowmeter :returns: gas temperature :rtype: float

volume_flow()

Gets the volumetric flow of the appropriate flowmeter

class drivers.**DAQ**

Driver for the 34970A Data Acquisition / Data Logger Switch Unit

Attributes	message
maxtry	max number to attempt command
status	whether the instrument is connected
therm	specifies type of thermistor
tref	‘101’ - channel for thermistor
te1	‘104’ - channel for electrode 1
te2	‘105’ - channel for electrode 2
volt	‘103’ - channel for voltage measurements
switch	‘205’,‘206’ - channels for switch between LCR and temp measurements
address	computer port address

Methods	message
connect	attempt to connect to the LCR meter
configure	configures device for measurements
get_temp	gets temperature from te1,te2 and tref
get voltage	gets voltage measurement
read_errors	reads errors stored in the DAQ
reset	resets the device
shut_down	shuts down the device
toggle_switch	switches configuration between temp and voltage

Note: do not change class attributes unless the physical wiring has been changed within the DAQ

`_config_temp``()`

Configures the thermistor (‘tref’) as 10,000 Ohm Configures both electrodes (‘te1’ and ‘te2’) as S-type thermocouples Sets units to degrees celsius

`_config_volt``()`

Configures the voltage measurements

`configure``()`

Configures the DAQ according to the current wiring

`connect``()`

Connects to the DAQ

`get_temp``()`

Scans the thermistor and thermocouples for temperature readings

Returns [tref,te1,te2]

Return type list of floats (degrees Celsius)

`get_thermopower``()`

Collects both temperature and voltage data and returns a list

`get_voltage``()`

Gets voltage across the sample from the DAQ

Returns voltage

Return type float

`read_errors``()`

Reads errors from the DAQ (unsure if working or not)

reset()

Resets the device

shutdown()

Shuts down the DAQ

toggle_switch(*command*)

Opens or closes the switch to the lcr. Must be closed for impedance measurements and open for thermopower measurements.

Parameters **command** (*str*) – either 'thermo' to make thermopower measurements or 'impedance' for impedance measurements

class drivers.**Furnace**(*ports=None*)

Driver for the Eurotherm 3216 Temperature Controller

Note: units are in °C

Attributes	message
maxtry	max number to attempt command
default_temp	revert to this temperature when resetting
status	whether the instrument is connected
address	computer port address

Methods	message
connect	attempt to connect to the LCR meter
set_temp	set temperature of furnace
get_temp	get temperature from furnace
set_heatrate	set heatrate of furnace
get_heatrate	get heatrate from furnace
set_other	set another parameter on furnace
get_other	get another parameter from furnace
reset	resets the device

_connect(*ports*)

Attempts connection to the furnace through each port in ports. Stops searching when connection is successful

Parameters **ports** (*list*) – names of available serial ports

heating_rate(*heat_rate=None*, *address=35*)

Sets the desired heating rate of furnace. Modbus address - 35

Parameters **heat_rate** (*float, int*) – heating rate in °C/min

Returns True if succesful, False if not

Return type Boolean

indicated(*address=1*)

Query current temperature of furnace. Modbus address - 1

Returns Temperature in °C if succesful, else False

Return type float/boolean

other (*address*, *value=None*)
set value at specified modbus address.

Parameters

- **modbus_address** (`float, int`) – see furnace manual for adresses
- **val** (`float, int`) – value to be sent to the furnace

Returns True if succesful, False if not

Return type Boolean

reset ()
resets the furnace to default temperature

setpoint_1 (*temperature=None*, *address=24*)
Sets target temperature of furnace. Modbus address - 24

Parameters **temp** (`float, int`) – temperature in °C

Returns True if succesful, False if not

Return type Boolean

setpoint_2 (*temperature=None*, *address=25*)
Sets target temperature of furnace. Modbus address - 24

Parameters **temp** (`float, int`) – temperature in °C

Returns True if succesful, False if not

Return type Boolean

class drivers.**LCR**
Driver for the E4980A Precision LCR Meter, 20 Hz to 2 MHz

Attributes	message
maxtry	max number to attempt command
status	whether the instrument is connected
address	port name

Methods	message
connect	attempt to connect to the LCR meter
configure	configures device for measurements
write_freq	transfers desired frequencies to the LCR meter
trigger	gets impedance for one specified frequency
get_complexZ	retrieves complex impedance from the device
reset	resets the device

_connect ()
Connects to the LCR meter

_set_continuous (*mode='ON'*)
Allows the LCR meter to auto change state from idle to 'wait for trigger'

_set_format (*mode='ascii'*)
Sets the format type of the LCR meter. Defaults to ascii. TODO - allow for other format types

_set_source (*mode='remote'*)
Sets up the LCR meter to expect a trigger from a remote source

configure(*freq*)
Appropriately configures the LCR meter for measurements

display(*mode=None*)
Sets the LCR meter to display frequencies as a list

function(*mode='impedance'*)
Sets up the LCR meter for complex impedance measurements

get_complexZ()
Collects complex impedance from the LCR meter

list_mode(*mode=None*)
Instructs LCR meter to take a single measurement per trigger

reset()
Resets the LCR meter

shutdown()
Resets the LCR meter and closes the serial port

trigger()
Triggers the next measurement

write_freq(*freq*)
Writes the desired frequencies to the LCR meter

Parameters **freq** (*np.ndarray*) – array of frequencies

class drivers.**MFC**
Global driver for the Mass Flow Controllers

Note: see AlicatController for methods to control individual gases

Attributes	message
maxtry	max number to attempt command
status	whether the instrument is connected
co2	controls for the Carbon Dioxide (CO2) controller
co_a	controls for the coarse Carbon Monoxide (CO) controller
co_b	controls for the fine Carbon Monoxide (CO) controller
h2	controls for the Hydrogen (H2) controller
address	computer port address

Methods	message
close_all	closes all controllers
connect	attempt to connect to the LCR meter
flush_all	flushes data from the input/output buffer of all devices
fugacity_co	returns a ratio of CO2/CO to achieve desired oxygen fugacity
fugacity_h2	returns a ratio of H2/CO2 to achieve desired oxugen fugacity
reset	resets the device

Example

```
>>> import Drivers
>>> mfc = Drivers.MFC()
>>> mfc.co2.get_massflow()
```

_connect()
Connects to the mass flow controllers

close_all()
Closes all flow controllers

flush_all()
Flushes the input? buffer of all flow controllers

fo2_buffer(*temp*, *buffer*, *pressure=1.01325*)
Calculates oxygen fugacity at a given temperature and fo2 buffer

input options	type of fo2 buffer
'QFM'	quartz-fayalite-magnetite
'IW'	iron-wustite
'WM'	wustite-magnetite
'MH'	magnetite-hematite
'QIF'	quartz-iron-fayalite
'NNO'	nickel-nickel oxide
'MMO'	molyb
'CCO'	cobalt-cobalt oxide

Parameters

- **temp** (*float, int*) – Temperature in u'°C'
- **buffer** (*str*) – buffer type (see table for input options)
- **pressure** (*float, int*) – pressure in bar (default: surface pressure)

Returns log10 oxygen fugacity

Return type float

fugacity_co(*fo2p*, *temp*)
Calculates the ratio CO2/CO needed to maintain a constant oxygen fugacity at a given temperature.

Parameters

- **fo2p** (*float, int*) – desired oxygen fugacity (log Pa)
- **temp** (*float, int*) – temperature (u'°C)

Returns CO2/CO ratio

Return type float

fugacity_h2(*fo2p*, *temp*)
Calculates the ratio CO2/H2 needed to maintain a constant oxygen fugacity at a given temperature.

Parameters

- **fo2p** (*float, int*) – desired oxygen fugacity (log Pa)
- **temp** (*float, int*) – temperature (u'°C)

Returns CO2/H2 ratio

Return type float

reset()

Resets all connected flow controllers to 0 massflow

class drivers.**Motor**(*ports=None*)

Driver for the motor controlling the linear stage

Attributes	message
maxtry	max number to attempt command
status	whether the instrument is connected
home	approximate xpos where te1 == te2
max_xpos	maximum x-position of the stage
address	computer port address

Methods	message
home	move to the center of the stage
connect	attempt to connect to the LCR meter
move	moves the stage the desired amount in mm
get_xpos	get the absolute position of the stage
set_xpos	moves the stage the desired amount in steps
get_speed	get the current speed of the stage
set_speed	sets the movement speed of the stage
reset	resets the device
test	sends stage on a test run

_connect(*ports*)

attempts connection to the motor

Parameters **ports** (`list, string`) – list of available ports

_convertdisplacement(*displacement*)

Converts a positive or negative displacement (in mm) into a command recognisable by the motor

_convertspeed(*speed*, *default=True*)

Converts a speed given in mm/s into a command recognisable by the motor

center()

Moves stage to the absolute center

get_speed()

Gets the current speed of the motor

Returns speed of motor

Return type float

get_xpos()

Gets the current position of the stage

Returns x-position of stage

Return type str

home()

Moves furnace to the center of the stage (x = 5000)

move(*displacement*)

Moves the stage in the positive or negative direction

Parameters **`displacement`** (*`float, int`*) – positive or negative displacement [in mm]

`reset`()

Resets the stage position so that xPos=0

`set_speed`(*motor_speed*)

Sets the speed of the motor

Parameters **`motor_speed`** (*`float, int`*) – speed of the motor in mm/s

`set_xpos`(*xpos*)

Moves the linear stage to an absolute x position

Parameters **`xpos`** (*`float, int`*) – desired absolute position of stage in controller pulses

`test`()

Sends the motorised stage on a test run to ensure everything is working

`drivers.`**`get_ports`**()

Returns a list of available serial ports for connecting to the furnace and motor

Returns list of available ports

Return type list, str

CHAPTER

THREE

MODULE DATA

class data.**Data** (*freq=None*, *filename=None*)

Storage for all data collected during experiments. Data file are loaded into this object for processing and plotting

Attributes	Description
freq	array of frequencies for use by the LCR meter
filename	name of the file being used
time	times for each measurement
thermo	stroes thermopower data
gas	stores gas and fugacity data
temp	stores temperature data
imp	stores impedance data
xpos	stores stage x position at each measurement

Example

```
>>> lab = Laboratory.Setup('somefile.dat')
>>> print(lab.data.temp.indicated)
[100,105,110,115,120]
```

class data.**Gas**

Stores the seperate gas data under one roof

Attributes	Description
h2	hydrogen flow rate
co2	carbon dioxide flow rate
co_a	carbon monoxide corase flow rate
co_b	carbon monoxide corase flow rate

class data.**Impedance**

Stores complex impedance data

Attributes	Description
Z	impedance
theta	phase angle

class data.**MFC_data**

Stores gas data for an individual mass flow controller

class data.**Temp**

Stores furnace temperature data

Attributes	Description
target	target temperature of current cycle
indicated	temperature indicated by furnace

class data.**Thermo**

Stores thermopower data

Attributes	Description
tref	temperature of the internal thermistor
te1	temperature of electrode 1
te2	temperature of electrode 2
volt	voltage across the sample

data.**parse_datafile**(*filename*)

Parses a text file and stores the data in the Lab.Data object

Parameters **filename** (*str*) – name of the file to be parsed

CHAPTER

FOUR

MODULE PLOTTING

Created on Sat Apr 14 16:09:52 2018 @author: Sam

class plotting.**LabPlots**(*data*)

Contains an assortment of useful plots for visualising self.data

arrhenius()

Plots inverse temperature versus conductivity

cole(*temp_list*, *start=0*, *end=None*, *fit=False*)

Creates a Cole-Cole plot (imaginary versus real impedance) at a given temperature. Finds the available self.data to the temperature specified by 'temp'. A linear least squares circle fit can be added by setting fit=True.

Parameters **temp** (*float/int*) – temperature in degrees C

cond_fug()

Plots inverse temperature versus conductivity

cond_time()

Plots conductivity versus time

gas()

Plots mass_flow self.data for all gases versus time elapsed

imp_diameter()

Plots the impedance diameter versus time_elapsed

temperature()

Plots furnace indicated and target temperature, thermocouple temperature and thermistor self.data versus time elapsed

voltage()

Plots voltage versus time

plotting.**_calculate_conductivity**(*Z*, *theta*)

conductivity = length / area * resistance

CHAPTER

FIVE

MODULE UTILS

utils.**check_controlfile**(*controlfile*)

Checks to make sure the specified controlfile is a valid file that can be used by this program

Parameters **controlfile** (*pd.DataFrame*) – a loaded control file

utils.**data_logger**()

Sets up the data file in much the same way as the log file. Data cannot be output to the console. Data file can be found in /datafiles/

utils.**find_center**(*self*)

TODO - Attempts to place the sample at the center of the heat source such that te1 = te2. untested.

utils.**furnace_profile**(*self*)

Records the temperature of both electrodes (te1 and te2) as the sample is moved from one end of the stage to the other. Used to find the center of the stage or the xpos of a desired temperature gradient when taking thermopower measurements.

utils.**lab_logger**(*name*)

Sets up logging messages for the laboratory. Sends to both a file and the console by default. Levels for both the file and console can be set to anything defined by the python logging package (DEBUG,INFO,WARNING,ERROR,CRITICAL). Specified log level AND GREATER will be included. Logfiles can be found in /logfiles/

utils.**load_frequencies**(*min*, *max*, *n*, *log*, *filename*)

Creates an np.array of frequency values specified by either min, max and n or a file containing a list of frequencies specified by filename

utils.**load_obj**(*filename*)

Loads a .pkl file

Parameters **filename** (*str*) – full path to file. must be a .pkl

utils.**save_obj**(*obj*, *filename*)

Saves an object instance as a .pkl file for later retrieval. Can be loaded again using :meth:’Utils.load_obj’

Parameters

- **obj** (*class*) – the object instance to be saved
- **filename** (*str*) – name of file

utils.**send_email**(*toaddr*, *message*, *cc=False*, *logfile=False*, *datafile=False*)

Sends an email to the specified email address. logfile or datafile can be attached if desired. used mainly for email updates on progress during long measurement cycles. mailer is geophysicslabnotifications@gmail.com.

Parameters

- **toaddr** (*str*) – full email address of intended recipient

- **`message`** (*`str`*) – message to include in email
- **`cc`** (*`str, list`*) – email can be carbon copied to additional adresses in cc
- **`logfile`** (*`boolean`*) – whether to attach the current logfile
- **`datafile`** (*`boolean`*) – whether to attach the current datafile

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX