fluidlab Documentation

Release 0.0.2a0

Pierre Augier

November 05, 2015

Contents

FluidLab is a framework for studying fluid dynamics by experiments using Python. It is part of the wider project FluidDyn.

The project is still in a planning stage so it is still pretty unstable and many of its planned features have not yet been implemented.

CHAPTER 1

User Guide

1.1 Installation

FluidLab is part of the FluidDyn project. Some issues regarding the installation of Python packages are discussed in the main documentation of the project.

1.1.1 Dependencies

Fluidlab uses some very common scientific Python packages. They will be installed automatically during fluidlab install but it can actually be better to install them before.

- Numpy, Matplotlib, h5py

Other dependencies will be installed automatically and you do not have to worry about:

- pyusb, minimalmodbus

1.1.2 Optional dependencies

Some packages are used for some quite particular tasks. Just install them if needed:

- Cython, Scipy

We give some advises for packages that are not very easy to install:

Linux GPIB

Todo

Help install Linux GPIB

1.1.3 Install commands

The simplest way to install fluidlab is by using pip:

```
pip install fluidlab
```

However, since the package is still in alpha phase (testing for developers), it is often useful to install it from source in "develop" mode. To download the source, the easier is to use Mercurial:

```
hg clone https://bitbucket.org/fluiddyn/fluidlab
cd fluidlab
python setup.py develop
```

After the installation, try to run the unit tests by running `make tests` from the root directory or `python -m unittest discover` from the root directory or from any of the "test" directories.

1.2 Tutorials

```
%matplotlib inline
import os
import fluiddyn as fld
import fluidlab
```

1.2.1 Tutorial: working in the laboratory (user perspective)

FluidDyn uses the object-oriented programming concepts. It deals with objects, which is a very natural way to represent and drive experiments since experiments consist in objects interacting with each other.

Regarding the laboratory, each physical object (a pump, a traverse, a probe, an acquisition board, a tank filled with a stratified fluid...) is represented and controlled by an instance of a class. The experimental results can also be represented by other types of objects.

Example of a conductivity probe attached to a moving traverse

Let's consider a real-life example, how to use a conductivity probe attached to a moving traverse. FluidDyn provides the class *fluidlab.objects.probes.MovingConductivityProbe* which can be used like this:

```
# import the class representing the moving conductivity probe
from fluidlab.objects.probes import MovingConductivityProbe

# create an instance of this class
probe = MovingConductivityProbe()

try:
    # set a parameter, the sample rate (in Hz)
    probe.set_sample_rate(2000)

    # just move the probe (in mm and mm/s)
    probe.move(deltaz=-100, speed=50)

    # just measure without moving (in s)
    measurements1 = probe.measure(duration=5)

    # move and measure (in mm and mm/s)
    measurements2 = probe.move_measure(deltaz=100, speed=100)
except AttributeError:
    pass
```

Of course this is a very simple example and there are more options to create the object *probe* and for the functions. Look at the documentation, i.e. in this case here: *fluidlab.objects.probes.MovingConductivityProbe*.

Save and load an object

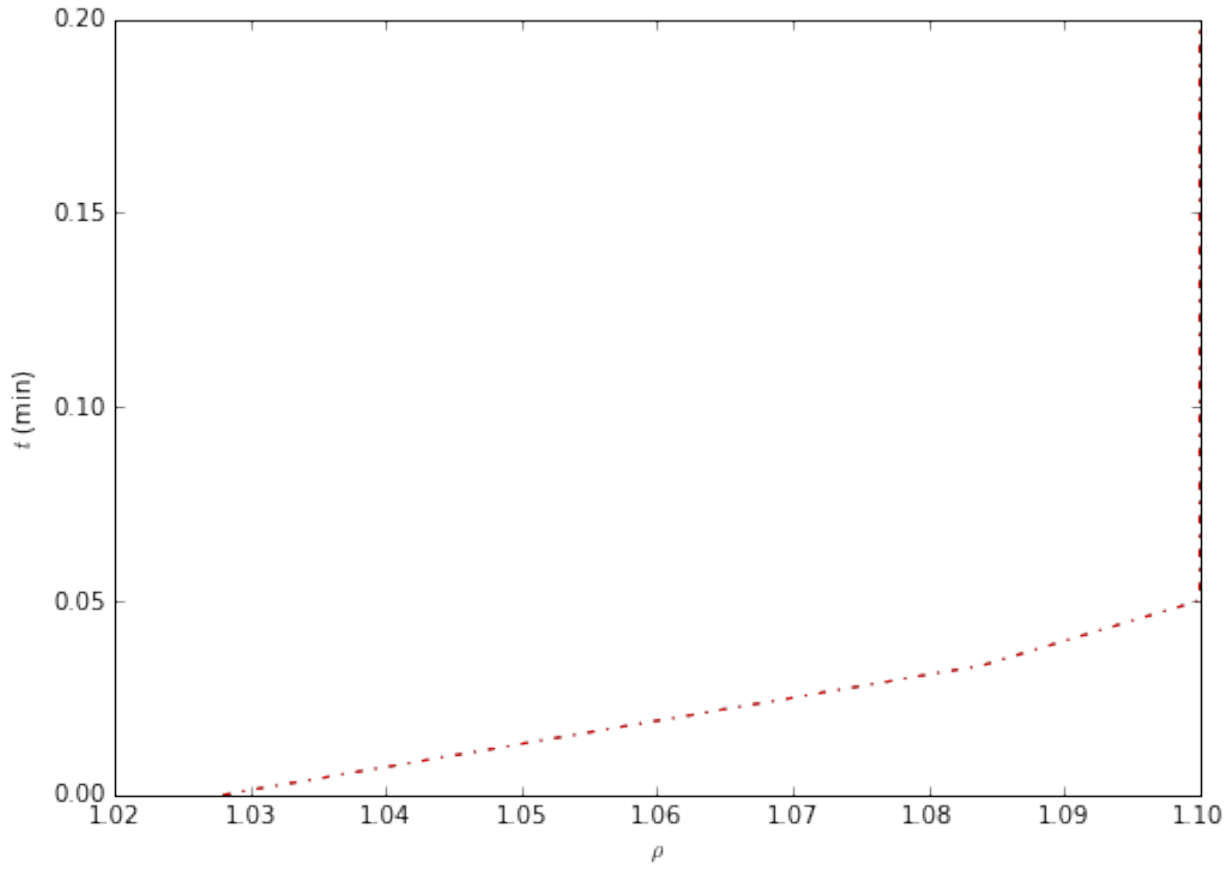
For some classes of FluidDyn, the objects can be saved in a file and loaded afterwards. This is a very useful feature! To see how it works, we can consider the example of a tank filled with a stratified fluid, which is represented by the class *fluidlab.objects.tanks.StratifiedTank*. Let's first see how we create a tank:

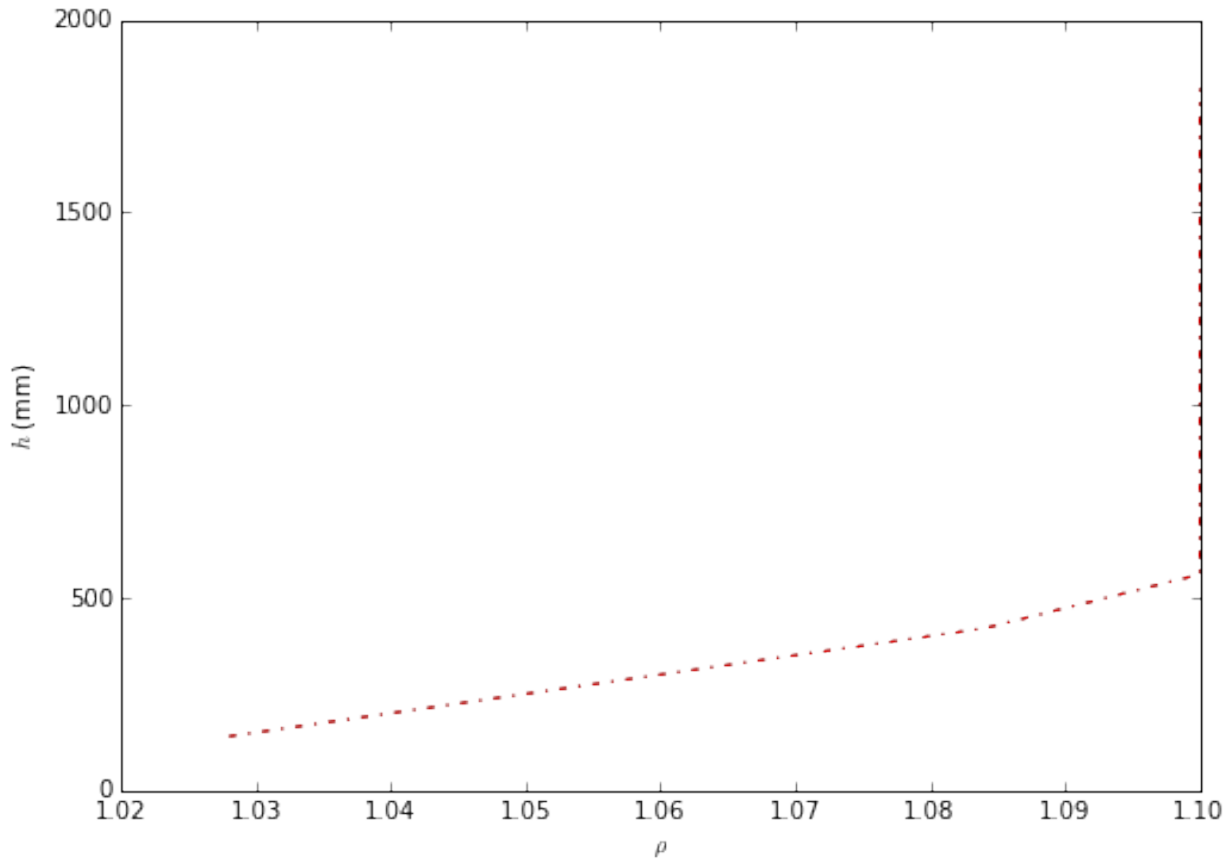
```
from fluidlab.objects.tanks import StratifiedTank

# create a tank with a linear stratification (see the doc of the class)
tank = StratifiedTank(
    H=550, S=100,
    z=[0, 500], rho=[1.1, 1])
```

The numerical object *tank* contains some information and can be use to do useful. We can for example fill the physical tank with the wanted profile (which makes use of some pumps also controlled by FluidDyn, see the class *fluidlab.objects.pumps.MasterFlexPumps*):

```
tank.fill()
```

```
Warning: can not fill without pumps. It will only perform a test of
the filling. To really fill the tank, set argument pumps to True or to
an instance of class MasterFlexPumps.

flowrate_tot: 840.00 ml/min
vol_to_pump: 192.00 ml
time for the filling:  0.23 min
volume pumped / volume to pump = 0.9479
The filling is finished.
```


0.20
0.15
0.10
0.05
0.00
t (min)
1.02
1.03
1.04
1.05
1.06
1.07
1.08
1.09
1.10
ρ



The numerical object *tank* can be saved in a file tank.h5 with its function *save* (the documentation explains how to control where the file is saved):

```
if os.path.exists('/tmp/tank.h5'):
    os.remove('/tmp/tank.h5')
```

```
tank.save('/tmp')
```

If we come back some days later and we want to use again this particular instance of *fluidlab.objects.tanks.StratifiedTank*. Let's assume that the file is in a directory `/tmp/exp0`. If we really know that this file contains the information for loading an object of *fluidlab.objects.tanks.StratifiedTank*, we can obtain the numerical representation of the tank by doing:

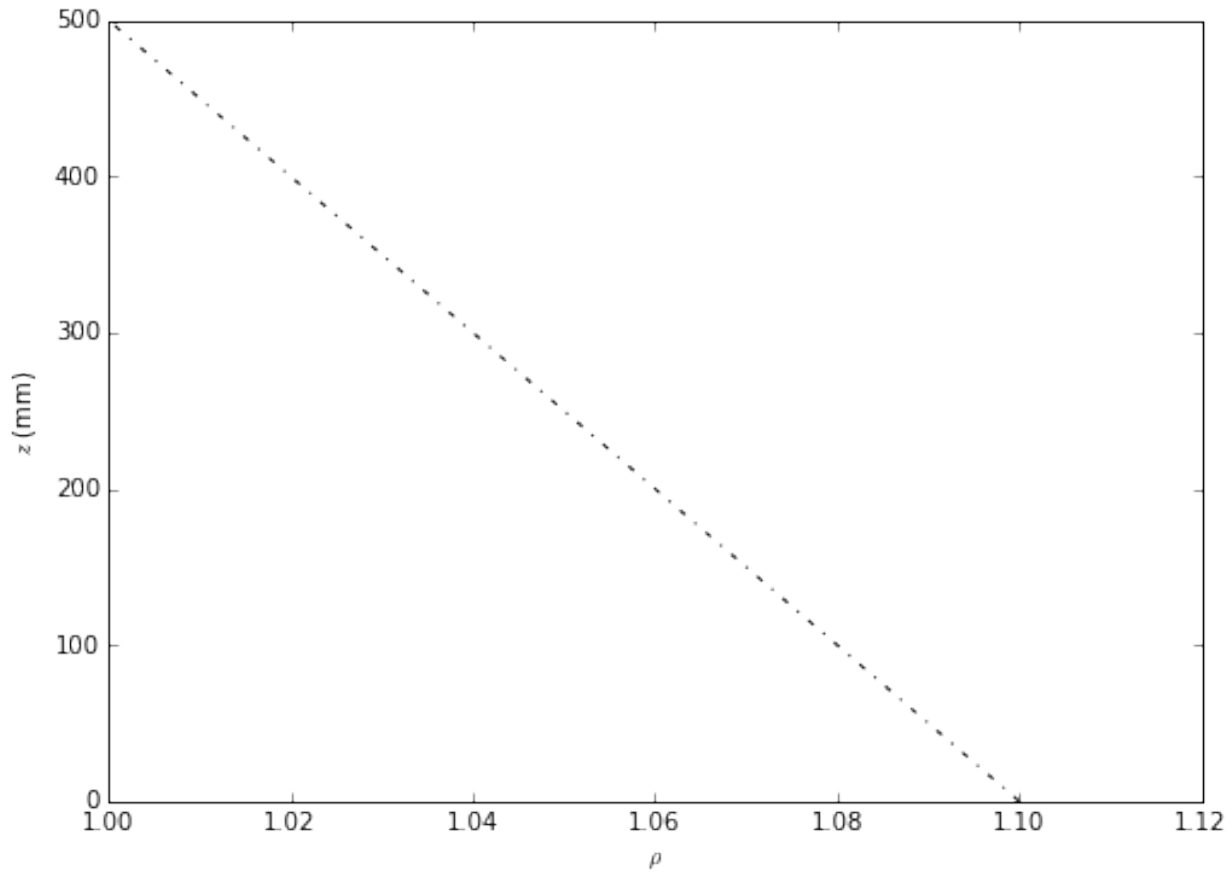
```
del(tank)
tank = StratifiedTank(str_path='/tmp')
```

But most of the case, it is easier and safer to use the function `fluiddyn.util.util.create_object_from_file()` like this:

```
path_to_tank_h5 = '/tmp/tank.h5'
tank = fld.create_object_from_file(path_to_tank_h5)
```

The function `create_object_from_file()` gets the correct class from information written in the file, calls the constructor of this class and return the object.

```
tank.profile.plot()
```



Representation of an experiment

Physically, an experiment consists in interacting objects. The experimentalist wants to control the actions of the objects with a good control in space and time and in a reproducible way. The results are then measurements produced by the measuring objects. Usually, after the experiment has been set up, it is repeated a number of times in order to vary some parameters.

A experimental set-up is represented in FluidDyn by a class derived from the class *fluidlab.exp.base.Experiment*. The experiment class has attributes that represent the physical objects interacting in the experimental set-up.

Each realisation of the experimental set-up (with a particular set of parameters) is represented by an instance of the experiment class. Each experiment (each realisation) is associated with a particular directory.

In order to create a class, do for example:

```
from fluidlab.exp.taylorcouette.linearprofile import ILSTaylorCouetteExp

exp = ILSTaylorCouetteExp(
    rho_max=1.1, N0=1., prop_homog=0.1,
    Omega1=0.4, Omega2=0, R1=150, R2=200,
    description="""This experiment is for the tutorial.""")

[attr for attr in dir(exp) if not attr.startswith('_')]
```

```
['board',
 'description',
 'first_creation',
 'name_dir',
 'params',
 'path_save',
 'profiles',
 'save_script',
 'tank',
 'time_start']
```

```
print(exp.description)
```

```
Experiment in a Taylor-Couette.

This tank is 520 mm high. The radius of the outer cylinder is
approximately   200 mm.


Initially linear stratification (ILS)...

This experiment is for the tutorial.
```

```
print(exp.path_save)
```

```
/home/users/augier3pi/Exp_data/TaylorCouette/ILS/Exp_Omega1=0.40_N0=1.00_2015-06-24_17-32-49
```

When this experiment has been created, the description files have been automatically saved in the "right" place. This "right" place being defined in the class of the experiment. Then we can easily reload the experiment.

```
path_save = exp.path_save
del(exp)
exp = fluidlab.load_exp(path_save[-20:-5])
print(exp.path_save)
print('R2 = {}'.format(exp.params['R2']))
```

```
/home/users/augier3pi/Exp_data/TaylorCouette/ILS/Exp_Omega1=0.40_N0=1.00_2015-06-24_17-14-39
R2 = 200
```

Note that I deliberately only use the string `path_save[-20:-5]` to show that `fld.load_exp()` is sufficiently clever to find out an experiment that corresponds to this string. Be careful to provide a sufficiently long string to be sure to load the wanted experiment.

1.3 Examples

1.3.1 Simple loop with one power supply

We show how we switch on an off lights powered by a power supply. The driver for the device is defined in *fluidlab.instruments.powersupply.isotech_ips2303s*.

```
from time import time

from fluidlab.instruments.powersupply.isotech_ips2303s import IsoTechIPS2303S
from fluidlab.exp import Timer
```

```
voltage_max = 8

pwrsupply = IsoTechIPS2303S()

print('Initialize the device ' + pwrsupply.query_identification())
pwrsupply.set_output_state(False)
pwrsupply.iset1.set(0.3)
pwrsupply.iset2.set(0.3)
pwrsupply.vset1.set(0.)
pwrsupply.vset2.set(voltage_max)
pwrsupply.set_output_state(True)


def switch(channel):
    volt = channel.get()
    if volt > 0:
        volt = 0
    else:
        volt = voltage_max
    channel.set(volt)

time_step = 2.
total_time = 20.

print('Loop during total_time = {:7.5} s'.format(total_time))
t = 0.
timer = Timer(time_step)
tstart = timer.tstart
while t < total_time:
    print('time till start: {:7.5} s'.format(t))

    switch(pwrsupply.vset1)
    switch(pwrsupply.vset2)

    t = timer.wait_tick()

print('time at the end: {:7.5} s'.format(time() - tstart))
pwrsupply.set_output_state(False)
```

1.3.2 Control a motor

The video demonstrates how a motor (Modbus RTU with RS485 in RJ45) can be controlled with Python and the package fluidlab.

The driver of the motor is coded in the module *fluidlab.instruments.modbus.unidrive_sp*.

The first script with a simple loop:

```
from time import sleep

from fluidlab.exp import Timer
from fluidlab.instruments.modbus.unidrive_sp import UnidriveSP

motor = UnidriveSP()

# set a timer which ticks every 5 s
timer = Timer(time_between_ticks=5)
```

```
print('Enter in a loop for 3 ticks.')
for i in range(3):
    print('  Enter in the block of the loop. i = {}.'.format(i))
    print('  Start rotation with a frequency of 2 Hz.')
    motor.start_rotation(2)

    print('  Sleep 2 s.')
    sleep(2)

    print('  Change target rotation rate to 1 Hz.')
    motor.set_target_rotation_rate(1)

    print('  Sleep 2 s.')
    sleep(2)

    print('  Stop the motor.')
    motor.stop_rotation()

    print('  Wait for the tick.')
    timer.wait_tick()
```

And the code for the mini GUI (sorry, quick and dirty...):

```
"""Example of a tiny generic graphical driver.
================================================

.. autoclass:: GraphicalDriver
   :members:
   :private-members:

"""


import sys
from PySide import QtGui

from serial import SerialException


from fluidlab.instruments.modbus.unidrive_sp import (
    OpenLoopUnidriveSP, example_linear_ramps)


class FalseMotor(object):
    def __init__(self):
        self.rr = 0.

    def unlock(self):
        pass

    def lock(self):
        pass

    def set_target_rotation_rate(self, rr, check=False):
        self.rr = rr

    def get_target_rotation_rate(self):
        return self.rr
```

```
    def start_rotation(self, a):
        pass

    def stop_rotation(self):
        pass


class GraphicalDriver(QtGui.QWidget):
    def __init__(self, class_motor=OpenLoopUnidriveSP):
        super(GraphicalDriver, self).__init__()

        # initialization of the motor driver
        try:
            self.motor = class_motor()
        except (OSError, SerialException):
            self.motor = FalseMotor()

        # initialization of the window
        self.initUI()

    def create_btn(self, name, function, x, y):
        button = QtGui.QPushButton(name, self)
        self.grid.addWidget(button, x, y)
        button.clicked.connect(function)
        return button

    def initUI(self):
        self.grid = QtGui.QGridLayout()
        self.grid.setSpacing(20)

        # create few basic buttons
        self.create_btn('Unlock', self.motor.unlock, 0, 0)
        self.create_btn('Lock', self.motor.lock, 0, 1)
        self.create_btn('Start', self.motor.start_rotation, 1, 0)
        self.create_btn('Stop', self.motor.stop_rotation, 1, 1)
        self.create_btn('Triangle', self.triangle_dialog, 2, 1)

        # create a layout for the speed with one button and one "lcd"
        speed_layout = QtGui.QVBoxLayout()
        speed_layout.setSpacing(5)

        set_speed_btn = QtGui.QPushButton('Set speed', self)
        set_speed_btn.clicked.connect(self.show_set_speed_dialog)
        self.set_speed_btn = set_speed_btn

        self.lcd = QtGui.QLCDNumber(self)
        self.lcd.display(self.motor.get_target_rotation_rate())

        speed_layout.addWidget(set_speed_btn)
        speed_layout.addWidget(self.lcd)

        self.grid.addLayout(speed_layout, 2, 0)

        # global setting
        self.setLayout(self.grid)
        self.move(400, 300)
        self.setWindowTitle('Leroy Somer driver')
        self.show()
```

```
    def show_set_speed_dialog(self):
        speed, ok = QtGui.QInputDialog.getText(
            self, 'Input Dialog',
            'Enter the motor speed in Hz:')
        if ok:
            self.motor.set_target_rotation_rate(float(speed))
            self.lcd.display(self.motor.get_target_rotation_rate())

    def triangle_dialog(self):
        max_speed, ok_a = QtGui.QInputDialog.getText(
            self, 'Input Dialog 1',
            'Enter the maximum motor speed in Hz:')
        duration, ok_b = QtGui.QInputDialog.getText(
            self, 'Input Dialog 2',
            'Enter the duration in s:')
        steps, ok_c = QtGui.QInputDialog.getText(
            self, 'Input Dialog 3',
            'Enter the number of steps:')
        if ok_a and ok_b and ok_c:
            example_linear_ramps(self.motor, max_speed, duration, steps)


def main():
    app = QtGui.QApplication(sys.argv)
    ex = GraphicalDriver()
    sys.exit(app.exec_())


if __name__ == '__main__':
    main()
```

1.3.3 Experimental session and instruments

The first script does totally useless things but it shows how FluidLab can be used for very simple scripts to control experiments. An "experimental session" with one "data table" is created or loaded if it already exists. A figure to plot the two quantities *U0* and *U1* is defined. Data is saved at each time step of a short time loop.

Since there is no physical instruments, you should be able to try this script. You can modify the argument *email_to* to a real email address and uncomment the lines starting with "email" in the instantiation of the session. You can also modify the flag *raise_error* to see what it gives with the logger and the emails.

```
from __future__ import print_function, division

import numpy as np

from fluidlab.exp import Session, Timer
from fluidlab.output import show

raise_error = False

# unless explicitly mentioned, FluidLab uses SI units, so the times are is s
total_time = 5.
time_step = 0.2
omega = 2*np.pi/2.

# conversion volt into temperature (nothing physical, just for the example)
```

```
alpha = 2.

# initialize session, log, saving and emails
session = Session(
    path='Tests',
    name='False_exp',
    # email_to='experimentalist@lab.earth',
    email_title='False experiment without instrument',
    email_delay=30  # time in s
)

print = session.logger.print_log

data_table = session.get_data_table(
    fieldnames=['U0', 'U1', 'T0', 'T1'])

data_table.init_figure(['U0', 'U1'])
data_table.init_figure(['T0', 'T1'])

# initialization of the time loop
t_last_print = 0.
t = 0.
timer = Timer(time_step)

# start the time loop
while t < total_time:

    U0 = np.cos(omega*t) + 0.1*np.random.rand()
    U1 = U0 * np.random.rand()

    data_table.save({'U0': U0, 'U1': U1,
                     'T0': alpha*U0, 'T1': alpha*U1})

    if t - t_last_print > 1 - time_step/2.:
        t_last_print = t
        print('time till start: {:8.5} s'.format(t))
        data_table.update_figures()
        session.logger.send_email_if_has_to(figures=data_table.figures)

    t = timer.wait_tick()

if raise_error:
    print("let's raise a ValueError to see what it gives.")
    raise ValueError('The flag raise_error is True...')

print('Time end: {:8.5} s'.format(t))

show()
```

The next script also corresponds to a false experiment but two real instruments, a function generator (*fluidlab.instruments.funcgen.tektronix_afg3022b*) and an oscilloscope (*fluidlab.instruments.scope.agilent_dsox2014a*), are used:

```
from __future__ import print_function, division

import numpy as np

from fluidlab.exp import Session, Timer
```

```
from fluidlab.output import show

from fluidlab.instruments.scope.agilent_dsox2014a import AgilentDSOX2014a
from fluidlab.instruments.funcgen.tektronix_afg3022b import TektronixAFG3022b

total_time = 5.
time_step = 1.
omega = 2*np.pi/2.

# conversion volt into temperature (nothing physical, just for the example)
alpha = 0.01

# initialize session, log, saving and emails
session = Session(
    path='Tests',
    name='Exp_funcgen_scope',
    # email_to='experimentalist@lab.earth',
    # email_title='False experiment with function generator and oscilloscope',
    # email_delay=2*3600
)

print = session.logger.print_log
send_email_if_has_to = session.logger.send_email_if_has_to

data_table = session.get_data_table(
    fieldnames=['U0', 'U1', 'T0', 'T1'])

data_table.init_figure(['T0', 'T1'])

# setup the function generator
funcgen = TektronixAFG3022b('USB0::1689::839::C034062::0::INSTR')
offset = 1.
funcgen.function_shape.set('sin')
funcgen.frequency.set(1e4)
funcgen.voltage.set(0.)
funcgen.offset.set(offset)
funcgen.output1_state.set(True)

# setup the oscilloscope
scope = AgilentDSOX2014a('USB0::2391::6040::MY51450715::0::INSTR')
scope.channel1_coupling.set('DC')
scope.channel1_range.set(5.)
scope.timebase_range.set(1e-3)
scope.trigger_level.set(offset)

# initialization of the time loop
t_last_print = 0.
t = 0.
timer = Timer(time_step)

# start the time loop
while t < total_time:

    volts_out = np.cos(omega*t) + 0.1*np.random.rand()

    funcgen.voltage.set(volts_out)

    time, volts = scope.get_curve(nb_points=200)
```

```
    U0 = volts.min()
    U1 = volts.max()

    data_table.save({'U0': U0, 'U1': U1,
                     'T0': alpha*U0, 'T1': alpha*U1})

    if t - t_last_print > 1 - time_step/2.:
        t_last_print = t
        print('time till start: {:8.5} s'.format(t))
        data_table.update_figures()
        send_email_if_has_to()

    t = timer.wait_tick()

print('Time end: {:8.5} s'.format(t))
show()
```

1.3.4 Parallel tasks

```
"""This example demonstrates how we can launch independent daemons to
perform tasks in parallel.

It produces the following output (all times are in seconds):

Daemons launched!
loop dt = 0.100, t = 0.00000
loop dt = 0.025, t = 0.00000
loop dt = 0.025, t = 0.02514
loop dt = 0.025, t = 0.05015
loop dt = 0.025, t = 0.07510
loop dt = 0.100, t = 0.10018
loop dt = 0.025, t = 0.10010
loop dt = 0.025, t = 0.12509
loop dt = 0.025, t = 0.15013
loop dt = 0.025, t = 0.17510
end of loop dt = 0.100, t = 0.20019
end of loop dt = 0.025, t = 0.20009
End of the script

"""
from time import sleep

from fluidlab.exp import Timer

# We can use processes or threads...
from fluiddyn.util.daemons import DaemonProcess as Daemon
# from fluiddyn.util.daemons import DaemonThread as Daemon


def make_loop_function(dt, total_time):
    def loop():
        timer = Timer(dt)
        t = 0
        while t < total_time:
            print('loop dt = {:5.3f}, t = {:7.5f}'.format(dt, t))
            t = timer.wait_tick()
```

```
        print('end of loop dt = {:5.3f}, t = {:7.5f}'.format(dt, t))
    return loop


total_time = 0.2

daemons = []
for dt in [0.1, 0.025]:
    loop = make_loop_function(dt, total_time)
    daemons.append(Daemon(loop))

for daemon in daemons:
    daemon.start()

print('Daemons launched!')
sleep(total_time + 0.02)
print('End of the script')
```

CHAPTER 2

Modules Reference

fluidlab.instruments	Represent and communicate with instruments
fluidlab.objects	Laboratory objects
fluidlab.exp	Experiment classes
fluidlab.postproc	Post-processing

2.1 fluidlab.instruments

2.1.1 Represent and communicate with instruments

We call "instruments" all devices that can be more or less directly plug to a computer. This packages provides instrument drivers and utilities to write them.

In FluidLab, the drivers have an attribute *interface* that contains relatively low-level functions to communicate with the device. Some base interface class are defined in the module *fluidlab.instruments.interfaces*.

The drivers have also attributes representing values that are saved or can be set in or get from the instruments. Some of these "features" are defined in *fluidlab.instruments.features*.

For many instruments, the communication with the computer is made directly through simple messages understandable by humans. For these instruments, the communication can be done using VISA (Virtual Instrument Software Architecture), independently of how the communication is done in practice (e.g. with GPIB, RS232, USB, Ethernet).

A mechanism to easily write such drivers is implemented in:

drivers	Instrument drivers (*fluidlab.instruments.drivers*)
interfaces	Interfaces with the instruments (*fluidlab.instruments.interfaces*)
features	Features for defining drivers (*fluidlab.instruments.features*)
iec60488	IEC60488 drivers (*fluidlab.instruments.iec60488*)

fluidlab.instruments.drivers

Instrument drivers (`fluidlab.instruments.drivers`)

Todo

Verify potential bug due to the fact that values are class attributes.

Provides:

class fluidlab.instruments.drivers.**Driver**(*interface=None*)
Bases: object

Instrument driver (base class).

Parameters **interface** : fluidlab.instruments.interface.Interface

The interface used to communicate with the instrument.

set(*name*, **args*, ***kargs*)
Set a value.

get(*name*, **args*, ***kargs*)
Get a value.

class fluidlab.instruments.drivers.**VISADriver**(*interface=None*, *backend='@py'*)
Bases: *fluidlab.instruments.drivers.Driver*

A VISA driver.

Parameters **interface** : {str or interface}

A VISA interface or a string defining a VISA interface.

backend : str

Defines the backend used by pyvisa ("@py", "@ni", "@sim"...)

Classes

Driver([interface])	Instrument driver (base class).
VISADriver([interface, backend])	A VISA driver.

fluidlab.instruments.interfaces

Interfaces with the instruments (fluidlab.instruments.interfaces)

Provides some classes:

class fluidlab.instruments.interfaces.**Interface**
Bases: object

class fluidlab.instruments.interfaces.**QueryInterface**
Bases: *fluidlab.instruments.interfaces.Interface*

class fluidlab.instruments.interfaces.**FalseInterface**
Bases: *fluidlab.instruments.interfaces.QueryInterface*

Provides some modules:

visa	
linuxgpib	Interfaces with linux-gpib (*fluidlab.instruments.interfaces.linuxgpib*)
serial_inter	Interfaces with serial (*fluidlab.instruments.interfaces.serial_inter*)

fluidlab.instruments.interfaces.linuxgpib

Interfaces with linux-gpib (`fluidlab.instruments.interfaces.linuxgpib`) Provides:

class `fluidlab.instruments.interfaces.linuxgpib.`**`GPIBInterface`**(*board_adress*, *instrument_adress*)

Bases: *fluidlab.instruments.interfaces.QueryInterface*

Classes

GPIBInterface(board_adress, instrument_adress)

fluidlab.instruments.interfaces.serial_inter

Interfaces with serial (`fluidlab.instruments.interfaces.serial_inter`) Provides:

class `fluidlab.instruments.interfaces.serial_inter.`**`SerialInterface`**(*port*, *baudrate=9600*, *bytesize=8*, *parity='N'*, *stopbits=1*, *timeout=1*, *xonxoff=False*, *rtscts=False*, *dsrdtr=False*)

Bases: *fluidlab.instruments.interfaces.QueryInterface*

Functions

Classes

SerialInterface(port[, baudrate, bytesize, ...])

Classes

FalseInterface
Interface
QueryInterface

fluidlab.instruments.features

Features for defining drivers (`fluidlab.instruments.features`)

Todo

Work on the documentation of *fluidlab.instruments.features*.

Provides:

class fluidlab.instruments.features.**Feature**(*name*, *doc=''*)
Bases: object

class fluidlab.instruments.features.**WriteCommand**(*name*, *doc=''*, *command_str=''*)
Bases: *fluidlab.instruments.features.Feature*

_build_driver_class(*Driver*)
Add a "write function" to the driver class

class fluidlab.instruments.features.**QueryCommand**(*name*, *doc=''*, *command_str=''*, *parse_result=None*)
Bases: *fluidlab.instruments.features.Feature*

_build_driver_class(*Driver*)
Add a "query function" to the driver class

class fluidlab.instruments.features.**Value**(*name*, *doc=''*, *command_set=None*, *command_get=None*)
Bases: fluidlab.instruments.features.SuperValue

get()
Get the value from the instrument.

set(*value*, *warn=True*)
Set the value in the instrument.

class fluidlab.instruments.features.**BoolValue**(*name*, *doc=''*, *command_set=None*, *command_get=None*)
Bases: *fluidlab.instruments.features.Value*

class fluidlab.instruments.features.**StringValue**(*name*, *doc=''*, *command_set=None*, *command_get=None*, *valid_values=None*)
Bases: *fluidlab.instruments.features.Value*

class fluidlab.instruments.features.**NumberValue**(*name*, *doc=''*, *command_set=None*, *command_get=None*, *limits=None*)
Bases: *fluidlab.instruments.features.Value*

class fluidlab.instruments.features.**IntValue**(*name*, *doc=''*, *command_set=None*, *command_get=None*, *limits=None*)
Bases: *fluidlab.instruments.features.NumberValue*

class fluidlab.instruments.features.**FloatValue**(*name*, *doc=''*, *command_set=None*, *command_get=None*, *limits=None*)
Bases: *fluidlab.instruments.features.NumberValue*

class fluidlab.instruments.features.**RegisterValue**(*name*, *doc=''*, *command_set=None*, *command_get=None*, *keys=None*, *default_value=0*)
Bases: *fluidlab.instruments.features.NumberValue*

get_as_number()
Get the register as number.

get()
Get the register as dictionary.

set(*value*)
Set the register.

Parameters **value** : {dict, int}

The value as a dictionnary or an integer.

Functions

custom_formatwarning(message, category, ...)

Classes

BoolValue(name[, doc, command_set, command_get])
Feature(name[, doc])
FloatValue(name[, doc, command_set, ...])
IntValue(name[, doc, command_set, ...])
NumberValue(name[, doc, command_set, ...])
QueryCommand(name[, doc, command_str, ...])
RegisterValue(name[, doc, command_set, ...])
StringValue(name[, doc, command_set, ...])
SuperValue(name[, doc])
Value(name[, doc, command_set, command_get])
WriteCommand(name[, doc, command_str])

fluidlab.instruments.iec60488

IEC60488 drivers (`fluidlab.instruments.iec60488`)

Todo

Verify if IEC60488 drivers work and are convenient (which names for the functions and values?)...

[This module is inspired by the module *slave.iec60488*. Part of the documentation is taken from it. Thank you to the Slave authors.]

This module implements an IEC 60488-2:2004(E) compliant interface.

The IEC 60488-2 describes a standard digital interface for programmable instrumentation. It is used by devices connected via the IEEE 488.1 bus, commonly known as GPIB. It is an adoption of the *IEEE std. 488.2-1992* standard.

The IEC 60488-2 requires the existence of several commands which are implemented in the class:

class fluidlab.instruments.iec60488.**IEC60488**(*interface=None*)

Bases: *fluidlab.instruments.drivers.VISADriver*

Instrument driver with IEC 60488-2 interface

The IEC 60488-2 requires the existence of several commands which are logically grouped.

Reporting Commands

- **CLS* - Clears the data status structure.
- **ESE* - Write the event status enable register.
- **ESE?* - Query the event status enable register.
- **ESR?* - Query the event status register.
- **SRE* - Write the status enable register.
- **SRE?* - Query the status enable register.

- **STB* - Query the status register.

Internal operation commands

- **IDN?* - Identification query.
- **RST* - Perform a device reset.
- **TST?* - Perform internal self-test.

Synchronization commands

- **OPC* - Set operation complete flag high.
- **OPC?* - Query operation complete flag.
- **WAI* - Wait to continue.

Parameters **interface** : {str or interface}

A VISA interface or a string defining a VISA interface.

backend : str

Defines the backend used by pyvisa ("@py", "@ni", "@sim"...)

clear_status()
Clears the data status structure

event_status_enable_register = <fluidlab.instruments.features.RegisterValue object>
Event status enable register

Used in the status and events reporting system.

get_operation_complete_flag()
Get operation complete flag

perform_internal_test()
Perform internal self-test

query_esr()
Query the event status register

query_identification()
Identification query

query_stb()
Query the status register

reset_device()
Perform a device reset

status_enable_register = <fluidlab.instruments.features.RegisterValue object>
Status enable register

Used in the status reporting system.

wait()
Wait to continue

wait_till_completion_of_operations()
Return "1" when all operation are completed

Despite the required commands, there are several optional command groups defined. The standard requires that if one command is used, it's complete group must be implemented. These are implemented in the following mixin classes.

class fluidlab.instruments.iec60488.**PowerOn**(*interface=None*, *backend='@py'*)
Bases: *fluidlab.instruments.drivers.VISADriver*

A mixin class, implementing the optional power on commands.

Power on common commands

- **PSC* - Set the power-on status clear bit.
- **PSC?* - Query the power-on status clear bit.

power_on_status = <fluidlab.instruments.features.BoolValue object>
Power-on status.

class fluidlab.instruments.iec60488.**ParallelPoll**(*interface=None*, *backend='@py'*)
Bases: *fluidlab.instruments.drivers.VISADriver*

A mixin class, implementing the optional parallel poll commands.

Parallel poll common commands

- **IST?* - Query the individual status message bit.
- **PRE* - Set the parallel poll enable register.
- **PRE?* - Query the parallel poll enable register.

parallel_poll = <fluidlab.instruments.features.RegisterValue object>
Parallel poll enable register.

query_status_message_bit()
Query the individual status message bit

class fluidlab.instruments.iec60488.**ResourceDescription**(*interface=None*, *backend='@py'*)
Bases: *fluidlab.instruments.drivers.VISADriver*

A mixin class, implementing the optional resource description common commands.

Resource description common commands

- **RDT* - Store the resource description in the device.
- **RDT?* - Query the stored resource description.

resource_description = <fluidlab.instruments.features.StringValue object>
Resource description.

class fluidlab.instruments.iec60488.**ProtectedUserData**(*interface=None*, *backend='@py'*)
Bases: *fluidlab.instruments.drivers.VISADriver*

A mixin class, implementing the protected user data commands.

Protected user data commands

- **PUD* - Store protected user data in the device.
- **PUD?* - Query the protected user data.

user_data = <fluidlab.instruments.features.StringValue object>
Protected user data.

class fluidlab.instruments.iec60488.**Calibration**(*interface=None*, *backend='@py'*)
Bases: *fluidlab.instruments.drivers.VISADriver*

A mixin class, implementing the optional calibration command.

Calibration command

- **CAL?* - Perform internal self calibration.

perform_calibration()
Perform internal self calibration

class fluidlab.instruments.iec60488.**Trigger**(*interface=None*, *backend='@py'*)
Bases: *fluidlab.instruments.drivers.VISADriver*

A mixin class, implementing the optional trigger command.

Trigger command

- **TRG* - Execute trigger command.

trigger()
Execute trigger command.

class fluidlab.instruments.iec60488.**TriggerMacro**(*interface=None*, *backend='@py'*)
Bases: *fluidlab.instruments.drivers.VISADriver*

A mixin class, implementing the optional trigger macro commands.

Trigger macro commands

- **DDT* - Define device trigger.
- **DDT?* - Define device trigger query.

define_device_trigger = <fluidlab.instruments.features.StringValue object>
Define device trigger.

class fluidlab.instruments.iec60488.**Macro**(*interface=None*, *backend='@py'*)
Bases: *fluidlab.instruments.drivers.VISADriver*

A mixin class, implementing the optional macro commands.

Macro Commands

- **DMC* - Define device trigger.
- **EMC* - Define device trigger query.
- **EMC?* - Define device trigger.
- **GMC?* - Define device trigger query.
- **LMC?* - Define device trigger.
- **PMC* - Define device trigger query.

dmc()
Define device trigger (???).

emc = <fluidlab.instruments.features.StringValue object>
Define device trigger (???).

gmc()
Define device trigger (???).

lmc()
Define device trigger (???).

pmc()
Define device trigger (???).

class fluidlab.instruments.iec60488.**ObjectIdentification**(*interface=None*, *backend='@py'*)

Bases: *fluidlab.instruments.drivers.VISADriver*

A mixin class, implementing the optional object identification command.

Option Identification command

- **OPT?* - Option identification query.

opt_identification()
Option identification query.

class fluidlab.instruments.iec60488.**StoredSetting**(*interface=None*, *backend='@py'*)

Bases: *fluidlab.instruments.drivers.VISADriver*

A mixin class, implementing the optional stored setting commands.

Stored settings commands

- **RCL* - Restore device settings from local memory.
- **SAV* - Store current settings of the device in local memory.

restore_device_settings()
Restore device settings from local memory.

store_current_settings()
Store current settings of the device in local memory.

class fluidlab.instruments.iec60488.**Learn**(*interface=None*, *backend='@py'*)

Bases: *fluidlab.instruments.drivers.VISADriver*

A mixin class, implementing the optional learn command.

Learn command

- **LRN?* - Learn device setup query.

learn_device_setup()
Learn device setup query.

class fluidlab.instruments.iec60488.**SystemConfiguration**(*interface=None*, *backend='@py'*)

Bases: *fluidlab.instruments.drivers.VISADriver*

A mixin class, implementing the optional system configuration commands.

System configuration commands

- **AAD* - Accept address command.
- **DLF* - Disable listener function command.

accept_address_command()
Accept address command.

disable_listener()
Disable listener function command.

class fluidlab.instruments.iec60488.**PassingControl**(*interface=None*, *backend='@py'*)

Bases: *fluidlab.instruments.drivers.VISADriver*

A mixin class, implementing the optional passing control command.

Passing control command

- **PCB* - Pass control back.

pass_control_back()
Pass control back.

Classes

Calibration([interface, backend])	A mixin class, implementing the optional calibration command.
IEC60488([interface])	Instrument driver with IEC 60488-2 interface
Learn([interface, backend])	A mixin class, implementing the optional learn command.
Macro([interface, backend])	A mixin class, implementing the optional macro commands.
ObjectIdentification([interface, backend])	A mixin class, implementing the optional object identification command.
ParallelPoll([interface, backend])	A mixin class, implementing the optional parallel poll commands.
PassingControl([interface, backend])	A mixin class, implementing the optional passing control command.
PowerOn([interface, backend])	A mixin class, implementing the optional power on commands.
ProtectedUserData([interface, backend])	A mixin class, implementing the protected user data commands.
ResourceDescription([interface, backend])	A mixin class, implementing the optional resource description common comma
StoredSetting([interface, backend])	A mixin class, implementing the optional stored setting commands.
SystemConfiguration([interface, backend])	A mixin class, implementing the optional system configuration commands.
Trigger([interface, backend])	A mixin class, implementing the optional trigger command.
TriggerMacro([interface, backend])	A mixin class, implementing the optional trigger macro commands.

Todo

Write drivers for some “VISA instruments”...

Some drivers of particular “VISA instruments” are organized in the packages:

scope	Oscilloscopes (*fluidlab.instruments.scope*)
funcgen	Function generators (*fluidlab.instruments.funcgen*)
powersupply	Power supply (*fluidlab.instruments.powersupply*)
multimeter	Multimeters (*fluidlab.instruments.multimeter*)
multiplexer	Multiplexers (*fluidlab.instruments.multiplexer*)

fluidlab.instruments.scope

Oscilloscopes (fluidlab.instruments.scope)

Provides:

agilent_dsox2014a	agilent_dsox2014a

fluidlab.instruments.scope.agilent_dsox2014a

agilent_dsox2014a

class fluidlab.instruments.scope.agilent_dsox2014a.**AgilentDSOX2014a**(*interface=None*)
Bases: *fluidlab.instruments.iec60488.IEC60488*, *fluidlab.instruments.iec60488.Trigger*, *fluidlab.instruments.iec60488.ObjectIdentification*, *fluidlab.instruments.iec60488.Stored*, *fluidlab.instruments.iec60488.Learn*

Driver for the oscilloscope Agilent DSOX2014a.

get_curve(*nb_points=1000*, *format_output='byte'*)
Acquire and return two Numpy arrays (time and data).

Parameters **nb_points** : int

The number of points that have to be returned.

format_output : string

The format of the data that is sent from the scope. Has to be in ['ascii', 'byte'].

autoscale()
Autoscale the oscilloscope.

channel1_coupling = <fluidlab.instruments.features.StringValue object>
The type of input coupling for the channel.

It can be set to "AC" or "DC".

channel1_display = <fluidlab.instruments.features.BoolValue object>
A boolean setting the display of the channel.

channel1_probe_attenuation = <fluidlab.instruments.features.FloatValue object>
The probe attenuation ratio.

channel1_range = <fluidlab.instruments.features.FloatValue object>
The vertical full-scale range value (in volt).

channel1_scale = <fluidlab.instruments.features.FloatValue object>
The number of units per division of the channel.

channel2_coupling = <fluidlab.instruments.features.StringValue object>
The type of input coupling for the channel.

It can be set to "AC" or "DC".

channel2_display = <fluidlab.instruments.features.BoolValue object>
A boolean setting the display of the channel.

channel2_probe_attenuation = <fluidlab.instruments.features.FloatValue object>
The probe attenuation ratio.

channel2_range = <fluidlab.instruments.features.FloatValue object>
The vertical full-scale range value (in volt).

channel2_scale = <fluidlab.instruments.features.FloatValue object>
The number of units per division of the channel.

nb_points = <fluidlab.instruments.features.IntValue object>
The number of points returned.

timebase_range = <fluidlab.instruments.features.FloatValue object>
The time for 10 division in seconds.

trigger_level = <fluidlab.instruments.features.FloatValue object>
The trigger level voltage for the active trigger source.

Classes

AgilentDSOX2014a([interface])	Driver for the oscilloscope Agilent DSOX2014a.

fluidlab.instruments.funcgen

Function generators (fluidlab.instruments.funcgen)

Provides:

tektronix_afg3022b	tektronix_afg3022b
agilent_33220a	agilent_33220a
tti_tsx3510p	tti_tsx3510p

fluidlab.instruments.funcgen.tektronix_afg3022b

tektronix_afg3022b

class fluidlab.instruments.funcgen.tektronix_afg3022b.**TektronixAFG3022b**(*interface=None*)

Bases: *fluidlab.instruments.iec60488.IEC60488*, *fluidlab.instruments.iec60488.PowerOn*, *fluidlab.instruments.iec60488.Calibration*, *fluidlab.instruments.iec60488.Trigger*, *fluidlab.instruments.iec60488.ObjectIdentification*, *fluidlab.instruments.iec60488.Stored*

A driver for the function generator Tektronix AFG 3022 B.

frequency = <fluidlab.instruments.features.FloatValue object>
The function frequency.

function_shape = <fluidlab.instruments.features.StringValue object>
The function shape (str).

Has to be in ['sine', 'sin', 'ramp', 'square'] (not case sensitive)

offset = <fluidlab.instruments.features.FloatValue object>
The function offset (in volt).

voltage = <fluidlab.instruments.features.FloatValue object>
Peak to peak voltage (in volt).

Warning: The voltage depends on the impedance of the receiver of the signal. If its impedence is very large, the actual output voltage is twice what it should be. If its impedance is 50 ohm, there is no problem.

Classes

TektronixAFG3022b([interface])	A driver for the function generator Tektronix AFG 3022 B.

fluidlab.instruments.funcgen.agilent_33220a

agilent_33220a

class fluidlab.instruments.funcgen.agilent_33220a.**Agilent33220a**(*interface=None*)

Bases: *fluidlab.instruments.iec60488.IEC60488*, *fluidlab.instruments.iec60488.PowerOn*, *fluidlab.instruments.iec60488.Calibration*, *fluidlab.instruments.iec60488.Trigger*, *fluidlab.instruments.iec60488.ObjectIdentification*, *fluidlab.instruments.iec60488.Stored*

A driver for the function generator Agilent 33220A

frequency = <fluidlab.instruments.funcgen.agilent_33220a.Agilent33220a_Frequency object>
Wave frequency

get_generator_state()
Get the current configuration of the funcgen

vdc = <fluidlab.instruments.funcgen.agilent_33220a.Agilent33220a_Vdc object>
DC voltage

vrms = <fluidlab.instruments.funcgen.agilent_33220a.Agilent33220a_Vrms object>
RMS voltage

Functions

parse_agilent33220a_configuration_str(str)	Parse the Agilent 33220A configuration string.

Classes

Agilent33220a([interface])	A driver for the function generator Agilent 33220A
Agilent33220a_Frequency()	
Agilent33220a_Vdc()	
Agilent33220a_Vrms()	

fluidlab.instruments.funcgen.tti_tsx3510p

tti_tsx3510p

class fluidlab.instruments.funcgen.tti_tsx3510p.**TtiTsx3510p**(*interface=None*)
Bases: *fluidlab.instruments.iec60488.IEC60488*, *fluidlab.instruments.iec60488.PowerOn*, *fluidlab.instruments.iec60488.Calibration*, *fluidlab.instruments.iec60488.Trigger*, *fluidlab.instruments.iec60488.ObjectIdentification*, *fluidlab.instruments.iec60488.Stored*

A driver for the function generator Thurlby Thandar Instruments (TTI) TSX3510P.

idc = <fluidlab.instruments.funcgen.tti_tsx3510p.TTIFloatValue object>
set current setup value in Amps, reads the current output

onoff = <fluidlab.instruments.funcgen.tti_tsx3510p.TTIBoolValue object>
set on or off the power supply

vdc = <fluidlab.instruments.funcgen.tti_tsx3510p.TTIFloatValue object>
set voltage setup value in Volts, reads the voltage output

vmax = <fluidlab.instruments.funcgen.tti_tsx3510p.TTIFloatValue object>
set over-voltage value in Volts

wdc = <fluidlab.instruments.funcgen.tti_tsx3510p.TTIFloatValue object>
reads the power output

Classes

TTIBoolValue(name[, doc, command_set])	
TTIFloatValue(name[, doc, unit_str, ...])	
TtiTsx3510p([interface])	A driver for the function generator Thurlby Thandar Instruments (TTI) TSX3510P.

fluidlab.instruments.powersupply

Power supply (fluidlab.instruments.powersupply)

Provides:

isotech_ips2303s	Iso-tech IPS 2303S

fluidlab.instruments.powersupply.isotech_ips2303s

Iso-tech IPS 2303S

class fluidlab.instruments.powersupply.isotech_ips2303s.**IsoTechIPS2303S**(*baudrate=115200*)

Bases: *fluidlab.instruments.drivers.Driver*

Driver for the power supply IPS 2303S.

Parameters **baudrate** : {9600, 57600, 115200}

The baud rate (symbol per second). Warning: it is written in the documentation of the device that the baudrate has to be equal to 57600 or 115200. Actually, before I modified this parameter I was able to communicate with the device only with a baud rate of 9600 Bd.

set_beep(*ok_for_beep=True*)

Set beep on or off.

set_output_state(*on=True*)

Set output state on or off.

If the output state is off, there is no output.

set_operation_mode(*tracking='independent'*)

Set the operation mode (tracking).

Parameters **tracking** : {'independent', 'series', 'parallel'}

The tracking mode.

set_baud(*baud=115200*)

Set baud rate for the serial communication.

Parameters **baud** : {115200, 57600}

The baud rate (symbol per second).

print_device_help()

Print internal help.

_query_help()

Query help (list of commands).

get_iout1()

Get the actual current for channel 1 (A).

get_iout2()

Get the actual current for channel 2 (A).

get_vout1()

Get the actual voltage for channel 1. (V)

get_vout2()
Get the actual voltage for channel 2. (V)

iset1 = <fluidlab.instruments.powersupply.isotech_ips2303s.FloatValueIPS object>
Target output current for channel 1 (A).

iset2 = <fluidlab.instruments.powersupply.isotech_ips2303s.FloatValueIPS object>
Target output current for channel 2 (A).

query_error_message()
Query error message.

query_identification()
Identification query

query_status()
Query status

Return a dictionary containing information of the device.

vset1 = <fluidlab.instruments.powersupply.isotech_ips2303s.FloatValueIPS object>
Target output voltage for channel 1 (V).

vset2 = <fluidlab.instruments.powersupply.isotech_ips2303s.FloatValueIPS object>
Target output voltage for channel 2 (V).

Functions

for_dev_idn_with_serial()
for_dev_idn_with_visa()

Classes

FloatValueIPS(name[, doc, command_set])	Particular value for the driver IsoTechIPS2303S.
IsoTechIPS2303S([baudrate])	Driver for the power supply IPS 2303S.

fluidlab.instruments.multimeter

Multimeters (fluidlab.instruments.multimeter)

Provides:

hp_34401a	hp_34401a

fluidlab.instruments.multimeter.hp_34401a

hp_34401a
class fluidlab.instruments.multimeter.hp_34401a.**HP34401a**(*interface=None*)
Bases: *fluidlab.instruments.iec60488.IEC60488*

Driver for the multimeter HP 34401a.

ohm = <fluidlab.instruments.features.FloatValue object>

2-wire Ohm measurement

ohm_4w = <fluidlab.instruments.features.FloatValue object>
4-wire Ohm measurement

vdc = <fluidlab.instruments.features.FloatValue object>
Voltage measurement

Classes

HP34401a([interface])	Driver for the multimeter HP 34401a.

fluidlab.instruments.multiplexer

Multiplexers (fluidlab.instruments.multiplexer)

Provides:

agilent_34970a	agilent_34970a

fluidlab.instruments.multiplexer.agilent_34970a

agilent_34970a

class fluidlab.instruments.multiplexer.agilent_34970a.**Agilent34970a**(*interface=None*)
Bases: *fluidlab.instruments.iec60488.IEC60488*

Driver for the multiplexer Agilent 34970A.

scan(*channelList*, *functionName*, *samplesPerChan*, *sampleRate*)
Initiates a scan

write_vdc(*channel*, *value*)
Write DC-Voltage on specified AO channel.

Todo

Solve an evident bug in this function (should be easy).

idc = <fluidlab.instruments.multiplexer.agilent_34970a.Agilent34970aValue object>
DC Current

ohm = <fluidlab.instruments.multiplexer.agilent_34970a.Agilent34970aValue object>
2-wire resistance

ohm_4w = <fluidlab.instruments.multiplexer.agilent_34970a.Agilent34970aValue object>
4-wire resistance

temperature = <fluidlab.instruments.multiplexer.agilent_34970a.Agilent34970aValue object>
Temperature

vdc = <fluidlab.instruments.multiplexer.agilent_34970a.Agilent34970aValue object>
DC Voltage

vrms = <fluidlab.instruments.multiplexer.agilent_34970a.Agilent34970aValue object>
RMS Voltage

Classes

Agilent34970a([interface])	Driver for the multiplexer Agilent 34970A.
Agilent34970aValue(name[, doc, function_name])	

Other very common communication standards are Modbus and Firewire:

modbus	Modbus instruments (*fluidlab.instruments.modbus*)
firewire	Firewire instruments (*fluidlab.instruments.firewire*)

fluidlab.instruments.modbus

Modbus instruments (fluidlab.instruments.modbus)

Provides some base modules:

driver	Modbus driver (*fluidlab.instruments.modbus.driver*)
features	Modbus features (*fluidlab.instruments.modbus.features*)
interfaces	Modbus interfaces (*fluidlab.instruments.modbus.interfaces*)

fluidlab.instruments.modbus.driver

Modbus driver (fluidlab.instruments.modbus.driver) Provides:

class fluidlab.instruments.modbus.driver.**ModbusDriver**(*port*, *method='rtu'*, *timeout=1*, *module='minimalmodbus'*)

Bases: *fluidlab.instruments.drivers.Driver*

Driver for instruments communicating with Modbus.

Classes

ModbusDriver(port[, method, timeout, module])	Driver for instruments communicating with Modbus.

fluidlab.instruments.modbus.features

Modbus features (fluidlab.instruments.modbus.features) Provides:

class fluidlab.instruments.modbus.features.**Value**(*name*, *doc=''*, *adress=0*)

Bases: fluidlab.instruments.features.SuperValue

class fluidlab.instruments.modbus.features.**ReadOnlyBoolValue**(*name*, *doc=''*, *adress=0*)

Bases: *fluidlab.instruments.modbus.features.Value*

class fluidlab.instruments.modbus.features.**BoolValue**(*name*, *doc=''*, *adress=0*)

Bases: *fluidlab.instruments.modbus.features.Value*

class fluidlab.instruments.modbus.features.**ReadOnlyInt16Value**(*name*, *doc=''*, *adress=0*)

Bases: *fluidlab.instruments.modbus.features.Value*

class fluidlab.instruments.modbus.features.**Int16Value**(*name*, *doc=''*, *adress=0*)
Bases: *fluidlab.instruments.modbus.features.Value*

class fluidlab.instruments.modbus.features.**ReadOnlyFloat32Value**(*name*, *doc=''*, *adress=0*)
Bases: *fluidlab.instruments.modbus.features.Value*

class fluidlab.instruments.modbus.features.**Float32Value**(*name*, *doc=''*, *adress=0*)
Bases: *fluidlab.instruments.modbus.features.Value*

Classes

BoolValue(name[, doc, adress])
DecimalInt16Value(name[, doc, address, ...])
Float32Value(name[, doc, adress])
Int16StringValue(name[, doc, int_dict, adress])
Int16Value(name[, doc, adress])
ReadOnlyBoolValue(name[, doc, adress])
ReadOnlyFloat32Value(name[, doc, adress])
ReadOnlyInt16Value(name[, doc, adress])
Value(name[, doc, adress])

fluidlab.instruments.modbus.interfaces

Modbus interfaces (fluidlab.instruments.modbus.interfaces) Provides:

class fluidlab.instruments.modbus.interfaces.**ModbusInterface**
Bases: *fluidlab.instruments.interfaces.Interface*

class fluidlab.instruments.modbus.interfaces.**MinimalModbusInterface**(*port*, *method='rtu'*, *slave_address=1*, *timeout=1*)
Bases: *fluidlab.instruments.modbus.interfaces.ModbusInterface*

class fluidlab.instruments.modbus.interfaces.**PyModbusInterface**(*port*, *method='rtu'*, *timeout=1*)
Bases: *fluidlab.instruments.modbus.interfaces.ModbusInterface*

Classes

MinimalModbusInterface(port[, method, ...])
ModbusInterface
PyModbusInterface(port[, method, timeout])

and some Modbus drivers for particular instruments:

unidrive_sp	Unidrive SP motor (Leroy Somer)

fluidlab.instruments.modbus.unidrive_sp

Unidrive SP motor (Leroy Somer)

How to setup and control the motor Pad, parameters and menus

The power drive has to be setup using its pad. There are arrow keys and two important buttons (a red button for reset, validate and stop the motor and a green one to start it). Using the arrow key, you can access many parameters organized in 23 menus.

Menu 0 gathers important parameters from other menus. For a simple usage, it's the only one that matters.

Terminals

The power drive cannot be controlled only with the pad and terminals ("bornes" in french) have to be linked. In particular, we have to use:

- Terminal 22 gives 24 V.
- Terminal 31 has to be plugged to 24 V to give a "drive enable signal".
- Terminal 26 has to be plugged to 24 V to give a "run signal".

Indications written on the drive

- "inh" stands for inhibited, it means the motor is locked.
- "rdY" stands for ready.
- "trip" means there is a problem, check section K of the manual for solutions.

Control the motor with a computer

The value of the parameter 0.05 controls how the motor is driven.

- 0.05 -> PAd : controlled by the pad on the power drive.
- 0.05 -> Pr : controlled by other parameters (that can be set by the computer). In particular the rotating rate of the motor in proportional to the value of parameter 0.24. The "run signal" can be given with the parameter 6.34.

Modes

The power drive can drive the motor in three modes,

- Open loop,
- close loop,
- servo.

The parameter 0.48 correspond to the modes. In order to change the mode, one need to change the parameter 0.48, to change the parameter 0.00 and to reset the drive by pressing the red "reset" button. Then the user has to manually launch an auto-calibration process. Therefore, it is not possible to change mode only from the computer. Since some parameters have different meanings in the different modes, we provide one class for each mode.

The setup procedures for the different modes are described in the docstring of the classes.

class `fluidlab.instruments.modbus.unidrive_sp.`**`BaseUnidriveSP`**(*port=None*, *timeout=1*, *module='minimalmodbus'*)

Bases: *fluidlab.instruments.modbus.driver.ModbusDriver*

Base class for the driver for the motor driver Unidrive SP

Parameters **port** : {None, str}

The port where the motor is plugged.

timeout : {1, number}

Timeout for the communication with the motor (in s).

module : {'minimalmodbus', str}

Module used to communicate with the motor.

Notes

This class can be used to write other classes for drivers of the Unidrive SP.

mode = <fluidlab.instruments.modbus.unidrive_sp.StringValue object>
The operating mode.

unlock()
Unlock the motor (then rotation is possible).

lock()
Lock the motor (then rotation is not possible).

start_rotation(*speed=None*, *direction=None*)
Start the motor rotation.

Parameters **speed** : {None, number}

Rotation rate in Hz. If speed is None, start the rotation with the speed that the motor has in memory.

direction : {None, number}

Direction (positive or negative).

stop_rotation()
Stop the rotation.

set_target_rotation_rate(*rotation_rate*, *check=False*)
Set the target rotation rate in Hz.

get_target_rotation_rate()
Get the target rotation rate in Hz.

_number_of_pairs_of_poles = <fluidlab.instruments.modbus.unidrive_sp.Value object>
The number of pairs of poles of the motor.

_rated_current_open_loop = <fluidlab.instruments.modbus.unidrive_sp.Value object>
Rated current of the motor. Used in open loop.

_rated_voltage = <fluidlab.instruments.modbus.unidrive_sp.Value object>
The Rated voltage of the motor (V).

_reference_selection = <fluidlab.instruments.modbus.unidrive_sp.StringValue object>
Defines how the rotation speed is given to the motor.

- "preset" is what we want here,
- "pad" means it can be entered with the arrow keys of the motor pad

_rotate = <fluidlab.instruments.modbus.unidrive_sp.Value object>
Set this to 1 to give an order of rotation

_unlocked = <fluidlab.instruments.modbus.unidrive_sp.Value object>
When this variable is equal to 0, the motor is inhibited and displays "inh". When it is equal to 1, the motor is ready to run and displays "rdY".

acceleration_time = <fluidlab.instruments.modbus.unidrive_sp.Value object>
The time to go from 0 Hz to 100 Hz (s).

deceleration_time = <fluidlab.instruments.modbus.unidrive_sp.Value object>
The time to go from 100 Hz to 0 Hz (s).

class fluidlab.instruments.modbus.unidrive_sp.**OpenLoopUnidriveSP**(*port=None*, *timeout=1*, *module='minimalmodbus'*)

Bases: *fluidlab.instruments.modbus.unidrive_sp.BaseUnidriveSP*

Driver for the motor driver Unidrive SP setup in open loop mode.

Parameters **port** : {None, str}

The port where the motor is plugged.

timeout : {1, number}

Timeout for the communication with the motor (in s).

module : {'minimalmodbus', str}

Module used to communicate with the motor.

Notes

Setup of the power drive in "open loop" mode

See short guide (section 7.2) and long guide (chapter H1). Follow the instructions.

Example for LEGI

Reset in open loop mode:

- 0.00 -> 1253,
- 0.48 -> OPEn.LP + reset.

For this to work, the parameter 0.48 has to be changed. If it is already in open loop, you have to first to reset the motor in another mode and then reset it in open loop.

In case of error br.th, 0.51 -> 8 + reset.

Main parameters:

- 0.02 -> 200 (Hz, 50 * 4 pairs of poles),
- 0.03 -> 5 (s, time of acceleration 0 to 100 Hz),
- 0.04 -> 10 (s, time of deceleration 100 to 0 Hz),
- 0.21 -> th.

Motor parameters (read on the motor):

- 0.44 -> 400 (V),
- 0.45 -> 3000 (rpm, max (?) rotation rate),
- 0.46 -> 1 (A, current),
- 0.47 -> 200 (Hz, 3000/60 (Hz) * 4 pairs of poles).

Warning: the parameters 0.45 (motor rated speed, min-1) and 0.47 (rated frequency, Hz) must be proportional: Rated frequency = motor rated speed / 60 * number of pairs of poles.

Autocalibration

- 0.40 -> 2 (for rotating calibration, 1 for stationary calibration),
- Plug the terminals to send "drive enable signal" (link terminals 22 and 31) and "run signal" (link terminals 22 and 26),
- Remove the terminals,
- 0.00 - > 1000 (memorization of the parameters),
- Send "drive enable signal" (link terminals 22 and 31).

Other useful parameters:

- 6.15 -> 1 (unlock) or 0 (lock),
- 6.34 -> 1 (order of rotation) or 0 (no rotation).

set_target_rotation_rate(*rotation_rate*, *check=False*)
Set the target rotation rate in Hz.

get_target_rotation_rate()
Get the target rotation rate in Hz.

_min_frequency = <fluidlab.instruments.modbus.unidrive_sp.Value object>
Minimum limit of frequency (Hz).

_rated_frequency = <fluidlab.instruments.modbus.unidrive_sp.Value object>
Rated frequency of the driving signal of the motor (Hz).

It has to be equal to [rated speed / 60 * number of pairs of poles].

_rated_speed = <fluidlab.instruments.modbus.unidrive_sp.Value object>
Rated speed of the motor (rpm).

_speed = <fluidlab.instruments.modbus.unidrive_sp.Value object>
Frequency of the driving signal (Hz).

Warning: the actual rotation rate in Hz is equal to this value divided by the number of poles.

class fluidlab.instruments.modbus.unidrive_sp.**ServoUnidriveSP**(*port=None*, *timeout=1*, *module='minimalmodbus'*)

Bases: *fluidlab.instruments.modbus.unidrive_sp.BaseUnidriveSP*

Driver for the motor driver Unidrive SP setup in "servo" mode.

Warning: NotImplemented the content of the class has to be adapted to the mode.

Parameters **port** : {None, str}

The port where the motor is plugged.

timeout : {1, number}

Timeout for the communication with the motor (in s).

module : {'minimalmodbus', str}

Module used to communicate with the motor.

Notes

Setup of the power drive in "servo" mode

See short guide (section 7.2) and long guide (chapter H1). Follow the instructions.

Example for LEGI

Drive enable signal and run signal are not given

Reset in open loop mode:

- 0.00 -> 1253,
- 0.48 -> ServO + reset.

In case of error br.th, 0.51 -> 8 + reset.

Motor parameters (read on the motor and on 4146 documentation): - 0.02 -> 3000 (rpm, maximum velocity),,

- 0.21 -> th.
- 0.41 -> 12 (kHz, switching frequency)
- 0.42 -> 8 (number of poles)
- 0.45 -> 42 (s, thermal time constant),
- 0.46 -> 1 (A, stalling current),
- 0.47 -> 200 (Hz, 3000/60 (Hz) * 4 pairs of poles).

Coder parameters:

- 0.49 -> L2
- 3.34 -> 2048 (ppr)
- 3.36 -> 5 (V)
- 3.38 -> Ab.SErvo

Autocalibration

- 0.40 -> 2 (for rotating calibration, 1 for stationary calibration),
- Plug the terminals to send "drive enable signal" (link terminals 22 and 31) and "run signal" (link terminals 22 and 26),
- Unplug the terminals when calibration is over,
- connect the motor to the load
- 0.40 -> 3
- Plug the terminals to send "drive enable signal" (link terminals 22 and 31) and "run signal" (link terminals 22 and 26)
- Unplug the terminals when calibration is over,
- 0.00 - > 1000 (memorization of the parameters),
- Send "drive enable signal" (link terminals 22 and 31).

Other useful parameters:

- 6.15 -> 1 (unlock) or 0 (lock),
- 6.34 -> 1 (order of rotation) or 0 (no rotation).

set_target_rotation_rate(*rotation_rate*, *check=False*)
Set the target rotation rate in rpm.

get_target_rotation_rate()
Get the target rotation rate in rpm.

_min_frequency = <fluidlab.instruments.modbus.unidrive_sp.Value object>
Minimum limit of frequency (rpm).

_speed = <fluidlab.instruments.modbus.unidrive_sp.Value object>
Rotation rate of the motor (rpm).

How to read the commercial designation

At LEGI, we have a motor "055U2C300BAMRA063110". This name can be decomposed as 055-U-2-C-30-0-B-A-MR-A-063-110. The different part of the names mean:

- 055: frame size
- U: voltage (400 V)
- 2: torque selection (std)
- C: stator length
- 30: winding speed (3000 rpm)
- 0: brake (no brake)
- B: connection type
- A: output shaft (std)
- MR: Feedback device (Incremental encoder 2048 ppr)
- A: inertial (std)
- 063: PCD (std)
- 110: Shaft diameter.

Functions

example_linear_ramps(motor[, max_speed, ...])	

Classes

BaseUnidriveSP([port, timeout, module])	Base class for the driver for the motor driver Unidrive SP
OpenLoopUnidriveSP([port, timeout, module])	Driver for the motor driver Unidrive SP setup in open loop mode.
ServoUnidriveSP([port, timeout, module])	Driver for the motor driver Unidrive SP setup in "servo" mode.
StringValue(name[, doc, int_dict, parameter_str])	
Value(name[, doc, parameter_str, ...])	

Exceptions

ModeError	Some values are only useable in one mode (open_loop, closed_loop, servo) When a value is used, a function checks t

fluidlab.instruments.firewire

Firewire instruments (`fluidlab.instruments.firewire`)

Todo

Firewire interface and drivers.

Provides:

For other instruments, the communication is done with libraries:

sound	Data acquisition using sound input (*fluidlab.instruments.sound*)

fluidlab.instruments.sound

Data acquisition using sound input (`fluidlab.instruments.sound`)

Todo

Sound interface and drivers...

Provides:

The drivers for the data acquisition boards are also gather in this package:

daq	Data acquisition boards (*fluidlab.instruments.daq*)

fluidlab.instruments.daq

Data acquisition boards (`fluidlab.instruments.daq`)

Todo

Data acquisition board drivers.

Provides:

daqmx	DAQmx (National Instruments, *fluidlab.instruments.daq.daqmx*)

fluidlab.instruments.daq.daqmx

DAQmx (National Instruments, `fluidlab.instruments.daq.daqmx`)

Todo

DAQmx interface and drivers (using Comedi API?)...

Provides:

fluidlab.instruments.daq.daqmx.**read_analog**(*resource_names*, *terminal_config*, *volt_min*, *volt_max*, *samples_per_chan=1*, *sample_rate=1*, *coupling_types='DC'*, *output_filename=None*)

Read from the analog input subdevice.

Parameters **resource_names: {str or iterable of str}**

Analogic input identifier(s), e.g. 'Dev1/ai0'.

terminal_config: {'Diff', 'PseudoDiff', 'RSE', 'NRSE'}

A type of configuration (apply to all terminals).

volt_min : {number or iterable of numbers}

Minima for the channels.

volt_max : {number or iterable of numbers}

Maxima for the channels.

samples_per_chan: number

Number of samples per channel to read.

sample_rate: number

Sample rate for all channels (Hz).

coupling_types : {'DC', 'AC', 'GND', list of str}

Type of coupling for each resource.

output_filename: {None, str}

If specified data is output into this file instead of output arrays.

Functions

measure_freq(resource_name[, freq_min, freq_max])	Write analogic output
read_analog(resource_names, terminal_config, ...)	Read from the analog input subdevice.
write_analog(resource_names, sample_rate, ...)	Write analogic output
write_analog_end_task(task[, timeout])	End task.

Classes

2.2 fluidlab.objects

2.2.1 Laboratory objects

Provides some modules to represent and control devices:

tanks	Tanks (*fluidlab.objects.tanks*)
pumps	Pumps (*fluidlab.objects.pumps*)

Continued on next page

Table 2.41 – continued from previous page

probes	Probes (*fluidlab.objects.probes*)
pinchvalve	Pinching valve (*fluidlab.objects.pinchvalve*)
traverse	Traverses (*fluidlab.objects.traverse*)
rotatingobjects	

fluidlab.objects.tanks

Tanks (fluidlab.objects.tanks)

Todo

Solve a bug loading the tank (about values Rin, Rout...).

Provides:

class fluidlab.objects.tanks.**DensityProfile**(*z*, *rho*)
Bases: fluiddyn.util.signal.FunctionLinInterp

Represent the density profile as a function of the height.

Parameters **z** : array_like

Position.

rho : array_like

Surface.

Attributes

z: *numpy.ndarray*	Position.
rho: *numpy.ndarray*	Density profile.

plot()
Plot the density profile.

class fluidlab.objects.tanks.**Surface**(*z*, *S*)
Bases: fluiddyn.util.signal.FunctionLinInterp

Represent the surface as a function of the height.

Parameters **z** : array_like

Position.

S : array_like

Surface.

Attributes

z: *numpy.ndarray*	Position.
S: *numpy.ndarray*	Surface.

class fluidlab.objects.tanks.**StratifiedTank**(*H=520*, *S=100*, *z=None*, *rho=None*, *dico_profile=None*, *pumps=None*, *str_path=None*)

Bases: object

Represent a tank with a density profile.

Parameters **H** : {520, number}, optional

Height (in mm).

S : {100, number}, optional

Surface (in mm).

dico_profile : dict, optional

Characteristics of the profile.

pumps : *fluidlab.objects.pumps.MasterFlexPumps*

Represent the pumps.

str_path : str

Related to the path of the associated directory.

fill(*dt=1*, *pumps=False*, *hastoplot=True*, *vol_tube=142.0*)

Fill the tank.

Parameters **dt** : {2, number}, optional

Time interval (in s) between the change of flow rate.

pumps : {False, *fluidlab.objects.pumps.MasterFlexPumps*}, optional

If False, an instance is created.

hastoplot : {True, False}, optional

if True, plot the density profile.

Notes

Warning: I would advice not to use the "quick-edit option" of the command prompt in Windows since when you click on the command prompt window, a charactere is selected and when any characteres are selected, the output (and the program!) is frozen. This is unbelivable how bug-generating is it! Attaboy Windows!

save(*path_save*)

Save in a file tank.h5

class fluidlab.objects.tanks.**TaylorCouette**(*Rin=100*, *Rout=240*, *H=520*, *z=None*, *rho=None*, *dico_profile=None*, *str_path=None*)

Bases: *fluidlab.objects.tanks.StratifiedTank*

Represent a Taylor-Couette tank.

Functions

test_load()

Continued on next page

Table 2.42 – continued from previous page

`test_profiles()`
`test_save()`

Classes

DensityProfile(z, rho)	Represent the density profile as a function of the height.
StratifiedTank([H, S, z, rho, dico_profile, ...])	Represent a tank with a density profile.
Surface(z, S)	Represent the surface as a function of the height.
TaylorCouette([Rin, Rout, H, z, rho, ...])	Represent a Taylor-Couette tank.

fluidlab.objects.pumps

Pumps (`fluidlab.objects.pumps`)

Provides:

class `fluidlab.objects.pumps.`**`MasterFlexPumps`**(*nb_pumps=2*, *verbose=False*)

Bases: `object`

Represent some Masterflex pumps.

We use Masterflex L/S (model number 7551.00).

Parameters **nb_pumps** : {2, int}, optional

Number of pumps

verbose : {False, bool}, optional

If True, more verbose.

Notes

The pumps and this class are often used with the class *fluidlab.objects.tanks.StratifiedTank*.

Attributes

flow_rates_max: *numpy.ndarray*	Maximum flow rates of the pumps.
pumps: list	List of the pump indexes.
serial: serial.Serial	Object representing the serial port connected to the pumps.

`_command`(*command*, *pumps=None*, *verbose=False*)

Send a command to some pumps.

Parameters **command** : str

The command that has to be send to the pumps.

pumps : int or array_like

The index of one pump or an array_like containing the indexes of pumps.

verbose : bool

More verbose If equal to `True`.

Returns **results** : list

The results of the commands.

Notes

The command can be for example ‘’, ‘G0’, ‘I’, ‘z’...

go(*pumps=None*)
Start some pumps.

Send the command ‘G0’ to the pumps.

Parameters **pumps** : {None, int, array_like}, optional

The index of one pump or an array_like containing the indexes of pumps. If None, the function uses self.pumps.

set_rot_per_min(*rots_per_min=0*, *pumps=None*)
Set the number of rotations per min for some pumps.

Parameters **rots_per_min** : number or array_like

The rotation rate(s) in rotations per mimute.

pumps : {None, int, array_like}, optional

The index of one pump or an array_like containing the indexes of pumps. If None, the function uses self.pumps.

stop(*pumps=None*)
Stop some pumps.

Parameters **pumps** : {None, int, array_like}, optional

The index of one pump or an array_like containing the indexes of pumps. If None, the function uses self.pumps.

_give_list_pumps(*pumps=None*)
Return a list of pump indexes.

Parameters **pumps** : {None, int, array_like}, optional

The index of one pump or an array_like containing the indexes of pumps. If None, the function uses self.pumps.

calibrate(*pumps=None*, *nb_mins=4*)
Calibrate the pumps.

Parameters **pumps** : {None, int, array_like}, optional

The index of one pump or an array_like containing the indexes of pumps. If None, the function uses self.pumps.

nb_mins : {4, number}

Number of minutes for calibrating one pumps.

set_flow_rate(*flow_rates=0*, *pumps=None*)
Set the flow rates.

Parameters **flow_rates** : {number, array_like}

flow rates in ml/min.

pumps : {None, int, array_like}, optional

The index of one pump or an array_like containing the indexes of pumps. If None, the function uses self.pumps.

test_one_pump(*pump=1*, *vol_to_pump=2000.0*, *flow_rate_test=None*)

Test one pump and print actual maximum flowrate.

Pump with the pump with the index *pump* an approximate volume *vol_to_pump*. Ask for a measure of the volume actually pumped. Print the "actual" maximum flowrate for the tested pump (which can be written in the function __init__).

The "actual" maximum flowrate is the one that which has give a more accurate result, computed from the volume actually pumped.

Parameters **pump** : {1, int}, optional

A pump index.

vol_to_pump : {2000., number}

The volume to pump.

flow_rate_test : {None, number}, optional

The flow rate used for testing. If None, use 2/3 of the maximum flow rate for the tested pump.

Functions

modif_calib_file(pump, flow_rate)

Classes

MasterFlexPumps([nb_pumps, verbose])	Represent some Masterflex pumps.

fluidlab.objects.probes

Probes (fluidlab.objects.probes)

Provides:

class fluidlab.objects.probes.**ConductivityProbe**(*board=None*, *channel=1*, *sample_rate=1000*, *has_to_config_board=True*, *VALVE=True*)

Bases: *fluidlab.objects.boards.ObjectUsingBoard*

Represent a conductivity probe.

This is an example of modification.

Parameters **board** : ???

Object representing a board.

channel : number

The channel index.

prepare_calibration(*rho_min=1*, *rho_max=1.18*, *nb_solutions=6*)
Gives indications to prepare a calibration.

calibrate(*rhos*, *duration_1measure=4.0*)
Calibrates the probe.

Parameters **rhos** : array_like

The density ρ of the samples (in kg/l).

duration_1measure : number

The duration of one measurement (in s).

Notes

$\rho(C)$ and $U(C,T)$, where C the concentration, U the voltage and T the temperature.

plot_calibrations(*rhos=None*, *volts=None*, *rho_real=None*)
Plots the measurements of the saved calibrations.

save_calibration(*rhos*, *voltages*)
Saves the results of a calibration.

load_calibrations()
Loads the data from the previous calibrations.

create_function_from_data(*rhos*, *volts*)
Creates a function from data.

set_sample_rate(*sample_rate*)
Sets the sample rate.

measure_volts(*duration*, *sample_rate=None*, *return_time=False*, *verbose=False*)
Measure and return the times and voltages.

Parameters **duration :**

(in s)

measure(*duration*, *sample_rate=None*, *return_time=False*, *verbose=False*)
Measure and return the times and density.

Parameters **duration** : number

(in s)

sample_rate : number

(in Hz)

test_measure(*duration=2*, *hastoplot=True*, *rho_real=None*)
Test the measurement.

class fluidlab.objects.probes.**MovingConductivityProbe**(*board=None*, *channel=1*, *sample_rate=100*, *has_to_config_board=True*, *position_start=300.0*, *position_max=None*, *Deltaz=400.0*)
Bases: *fluidlab.objects.probes.ConductivityProbe*, *fluidlab.objects.traverse.Traverse*

Represent a conductivity probe that can be deplaced by a traverse.

Parameters (for the __init__ method)

board : , optional

Acquisition board.

channel : {1, int}, optional

Indice of the channel in the acquisition board.

sample_rate : {100, number}, optional

Sample rate of the probe.

has_to_config_board : {True, bool}, optional

Has to configure the board.

position_start : {300., number}, optional

Position when created (in mm).

position_max : {None, number}, optional

???

Deltaz : {400, number}, optional

Distance between extremal points.

move_measure(*deltaz=100*, *speed=100*, *sample_rate=None*, *return_time=False*)

Move the probe while measuring.

Parameters deltaz: number

Distance to move (in mm).

speed: number

(in mm/s)

sample_rate: number

(in Hz)

Functions

Classes

ConductivityProbe([board, channel, ...])	Represent a conductivity probe.
MovingConductivityProbe([board, channel, ...])	Represent a conductivity probe that can be deplaced by a traverse.

fluidlab.objects.pinchvalve

Pinching valve (*fluidlab.objects.pinchvalve*)

Provides:

class fluidlab.objects.pinchvalve.**PinchValve**(*board=None*, *channel=0*)

Bases: *fluidlab.objects.boards.ObjectUsingBoard*

A class handling the pinch valve.

class fluidlab.objects.pinchvalve.**ContextManagerOpenedValve**(*valve*)
Bases: object

class fluidlab.objects.pinchvalve.**FalseContextManager**
Bases: object

fluidlab.objects.pinchvalve.**tube_as_opened_as_possible**(*valve*)

Functions

tube_as_opened_as_possible(valve)	

Classes

ContextManagerOpenedValve(valve)	
FalseContextManager	
PinchValve([board, channel])	A class handling the pinch valve.

fluidlab.objects.traverse

Traverses (fluidlab.objects.traverse)

Provides:

class fluidlab.objects.traverse.**Traverse**(*board=None*, *position_start=300.0*, *position_max=None*, *Deltaz=400.0*)
Bases: *fluidlab.objects.boards.ObjectUsingBoard*

Represent a traverse.

move_nb_steps(*direction='up'*, *nb_steps=200*, *steps_per_second=500*, *bloquing=False*)
Moves *nb_steps* in the direction *direction*.

Parameters **direction** : str

The direction.

nb_steps : int

The number of steps.

steps_per_second : number

The number of steps per second (fixing the speed).

bloquing : bool

Whether or not the function should be bloquing.

move(*deltaz=100*, *speed=100*, *bloquing=False*)
Move by a particular distance with a particular speed.

Parameters **deltaz: number**

Distance to move (in mm).

speed: number

(in mm/s)

gotopos(*position*, *speed=100*, *bloquing=True*)
Go as close as possible of a position

Classes

Traverse([board, position_start, ...])	Represent a traverse.

and two packages to control acquisition boards and Raspberry Pi:

boards	Data Acquisition boards (*fluidlab.objects.boards*)
raspberrypi	Using a Raspberry Pi (*fluidlab.objects.raspberrypi*)

fluidlab.objects.boards

Data Acquisition boards (fluidlab.objects.boards)

Provides two small classes, *ObjectUsingBoard* and the modules:

nidaqnx	National Instrument board (*fluidlab.objects.boards.nidaqnx*)

fluidlab.objects.boards.nidaqnx

National Instrument board (fluidlab.objects.boards.nidaqnx) Provides:

class fluidlab.objects.boards.nidaqnx.**NIDAQBoard**
Bases: object

Handle for a National Instrument board (NIDAQNX).

class fluidlab.objects.boards.nidaqnx.**AnalogicOutput**
Bases: object

Analogic output.

class fluidlab.objects.boards.nidaqnx.**AnalogicInput**
Bases: object

Analogic input.

Classes

AnalogicInput()	Analogic input.
AnalogicOutput()	Analogic output.
NIDAQBoard()	Handle for a National Instrument board (NIDAQNX).

Remark: Can not import the module fluidlab.objects.boards.powerdaq in Linux... So no doc!

The classes for data acquisition boards should be obtained from this package. If the boards are not available, no error are raised and the classes are replaced by the class *fluidlab.objects.boards.FalseBoard*.

For example, with a computer without PowerDAQ board:

```
from fluidlab.objects.boards import PowerDAQBoard
board = PowerDAQBoard() # no error
assert(board.works == False) # no error
assert(not board) # no error
board.ain.configure(sample_rate=100)
# AttributeError: You tried to use a false acquisition board.
```

class fluidlab.objects.boards.**ObjectUsingBoard**(*board=None*, *VERBOSE=False*)
Bases: object

Useful to write classes for objects using a board.

class fluidlab.objects.boards.**FalseBoard**
Bases: object

Represent a false acquisition board.

This object tested as a boolean is False. It has an attribute *works* equal to False and an AttributeError on it returns an understandable message.

Classes

FalseBoard	Represent a false acquisition board.
ObjectUsingBoard([board, VERBOSE])	Useful to write classes for objects using a board.
PowerDAQBoard	alias of *FalseBoard*

fluidlab.objects.raspberrypi

Using a Raspberry Pi (fluidlab.objects.raspberrypi)

A Raspberry Pi is a very small minimalist computer. It can be useful in the laboratory.

This package provides:

daq	RPi acquisition board MCP3008SPI (*fluidlab.objects.raspberrypi.daq*)
remotecontrol	Remote control of RPi (*fluidlab.objects.raspberrypi.remotecontrol*)
server	Server (*fluidlab.objects.raspberrypi.server*)
torque	Torque measurements (*fluidlab.objects.raspberrypi.torque*)

fluidlab.objects.raspberrypi.daq

RPi acquisition board MCP3008SPI (fluidlab.objects.raspberrypi.daq)

Classes

MCP3008SPI([differential])	Analogic / digital conversion with the MCP3008 SPI ADC chip.

fluidlab.objects.raspberrypi.remotecontrol

Remote control of RPi (`fluidlab.objects.raspberrypi.remotecontrol`)

Classes

`RaspBerryPi()`	Remote control of the Raspberry pi.

fluidlab.objects.raspberrypi.server

Server (`fluidlab.objects.raspberrypi.server`) Defines a rpyc server.

Classes

`RaspberryPiService`

fluidlab.objects.raspberrypi.torque

Torque measurements (`fluidlab.objects.raspberrypi.torque`) Handle measurement, saving, loading and plotting of torque measured by a gain strain using a Raspberry Pi.

Functions

Classes

`Torque`([path_exp, name_exp])	A *Torque* object handles torque measurements.
`TorqueClient`([path_exp, name_exp, connect])	
`TorqueRaspberryPi`([path_exp, name_exp, ...])	

2.3 fluidlab.exp

2.3.1 Experiment classes

session	Experiment session (*fluidlab.exp.session*)
octavesession	Read older Octave-based experiment session (*fluidlab.exp.octavesession*)

fluidlab.exp.session

Experiment session (`fluidlab.exp.session`)

Todo

Improve *fluidlab.exp.session.Session* to produce a nice file *session.h5*.

Provides:

class fluidlab.exp.session.**Session**(*path=None*, *name=None*, *info=None*, *save_in_dir=True*, *email_to=None*, *email_title=None*, *email_delay=None*, *email_server='localhost'*)

Bases: object

Experimental session

Base class representing an experimental session. A session automatically creates or loads files containing data. It contains an object *logger* for printing with logging (and possibly sending emails).

It can create managers of data tables for saving, loading and plotting data time series (see *fluidlab.exp.session.DataTable*).

Parameters **path** : {None, str}

name : {None, str}

info : {None, str}

save_in_dir : {True, False}

email_to : {None, str}

email_title : {None, str}

email_delay : {None, int}

Time is second between two emails.

get_data_table(*name=None*, ***kargs*)

Create or get a data table.

See *fluidlab.exp.session.DataTable*.

class fluidlab.exp.session.**SessionWithDefaultParams**(*params*)

Bases: *fluidlab.exp.session.Session*

Not implemented

class fluidlab.exp.session.**DataTable**(*name=None*, *path=None*, *session=None*, *extension=None*, *fieldnames=None*, *add_time=True*, *add_clock=True*)

Bases: object

Data table for time series

Parameters **name** : {None, str}, optional

Name of the date table.

path : {None, str}, optional

Path of a directory or of a file.

session : {None, fluidlab.exp.session.Session}, optional

A session used to get its path.

extension : {None, 'csv'}, optional

An extension defining in which format the data is saved.

fieldnames : {None, array_like}, optional

An array_like of strings.

add_time : {True, False}, optional

add_clock : {True, False}, optional

init_figure(*varnames=None*)
Initialize a figure to follow the evolution of variables.

update_figures()
Update all active figures of the data table.

save(*dict_to_save*)
Save the data contained in the dict *dict_to_save*.

load(*fieldnames=None*, *skiptimes=0*)
Load the data contained in the files as a dict.

plot_vs_time(*varnames=None*)
Plot the evolution of variables.

Functions

Classes

DataTable([name, path, session, extension, ...])	Data table for time series
Session([path, name, info, save_in_dir, ...])	Experimental session
SessionWithDefaultParams(params)	Not implemented

fluidlab.exp.octavesession

Read older Octave-based experiment session (fluidlab.exp.octavesession)

Provides:

read_OctMI_session(sessionName, verbose=True)

Functions

colored(string, color)	
read_OctMI_session(sessionName[, verbose])	
read_cell_var(f, verbose)	Reads one Cell variable from Octave binary file and returns it as a list
read_header(f, verbose)	Reads Octave binary file header (one per file)
read_matrix_var(f, verbose)	Reads one variable of type 'matrix' from Octave binary file
read_octave_binary(path[, verbose])	Reads an Octave binary file.
read_scalar_struct_var(f, verbose)	Reads one scalar structure variable from Octave binary file and returns it as a d
read_scalar_var(f, verbose)	Reads one variable of type 'scalar' from Octave binary file
read_string_var(f, verbose)	Reads one sq_string variable from Octave binary file
read_var(f, verbose)	Reads one variable from Octave binary file

Exceptions

`OctaveReaderError`	OctaveReader error

Warning: Beware, most of the modules of this package (those listed below) are depreciated and will be at least rewritten with other base class.

Physically, an experiment consists in interacting objects. The experimentalist wants to control the actions of the objects with a good control in space and time and in a reproducible way. The results are then some measurements about the studied physical phenomenon produced by the measuring objects. Usually, after the experiment has been set up, it is repeted a number of times in order to vary some parameters.

A experimental set-up is represented in FluidDyn by a class derived from the class *fluidlab.exp.base.Experiment*. The experiment class has attributes that represent the physical objects interacting in the experimental set-up (composition).

Each realisation of the experimental set-up (with a particular set of parameters) is represented by an instance of the experiment class. Each experiment (each realisation) is associated with a directory.

This package provides:

- some modules defining classes to represent base experiments:

base	Base Experiments (*fluidlab.exp.base*)
withtank	Experiments with a tank (*fluidlab.exp.withtank*)
withconductivityprobe	

fluidlab.exp.base

Base Experiments (`fluidlab.exp.base`)

Provides:

class fluidlab.exp.base.**Experiment**(*params=None*, *description=None*, *str_path=None*)

Bases: object

Base class for classes representing an experiment.

Parameters (for the __init__ method)

params : dict, optional

Contain parameters.

description : str, optional

A description of the experiment.

str_path : str, optional

A string related to the path where the experiment is saved or will be saved.

Notes

There are two modes of creating an *Experiment*:

1. if *str_path* doesn't point on an already saved experiment, a new experiment is created (the instance variable *first_creation* is True).

2.if *str_path* points on an already saved experiment, the instance variable *first_creation* is False and the experiment is loaded.

Note that if you want to load an already saved experiment, it is more convenient to use the function `fluiddyn.load_exp()` like so:

```
exp = fld.load_exp('Omega=0.73_N0=1.83_2014-03-25_12-43-48')
```

Attributes

first_creation	(bool) False if the experiment has not been loaded from the disk.
params	(dict) Containing parameters.
description	(str) A description of the experiment..
path_save	(str) The absolute path of the directory associated with the experiment.
name_dir	(str) Name of the directory associated with the experiment.
time_start	(str) Coding the time of creation.

_init_from_str(*str_path*)

Basic initialisation (begin of path_save and first_creation).

This function can be run at the beginning of the __init__ function of all classes representing experiments. If the instance has no attribute first_creation, it finds out if it is the first creation of the experiment or if it has to be loaded from the disk.

Parameters **str_path** : str

A string related to the path where the experiment is saved or will be saved.

_create_self_params(*params*)

Initialise the dictionary params.

This function can be overridden in children classes. Then, the function should call ... ???

_verify_params_first_creation(*params*, *keys_needed*)

Verify the parameters during the first creation.

Parameters **params** : dict

A dictionary with parameters.

keys_needed : list

The keys needed for a class.

_init_name_dir()

Initialise the name of the directory where the data are saved.

Initialises *name_dir* as *begin*+*end*, with *begin* = *'Exp_'* and *end* codes the time of creation, then returns (*begin*, *end*).

This function can be overridden in children classes.

Returns **begin** : str

equal to *'Exp_'*.

end : str

coding the time of creation.

_complete_description(*description_class*, *description=None*)

Complete or create a descrition.

If *description* is None, returns *description_class*; otherwise, retruns the concatenation of the two strings.

Parameters **description_class** : str

A description.

description : str, optional

Another description.

_save_basic_infos()
Save some basic information on the experiment.

_load_basic_infos(*verbose=False*)
Load some basic information on the experiments.

save_script()
Save the file from where this function is called.

Functions

Classes

Experiment([params, description, str_path])	Base class for classes representing an experiment.
NumpyAwareJSONEncoder([skipkeys, ...])	

fluidlab.exp.withtank

Experiments with a tank (fluidlab.exp.withtank)

Provides:

class fluidlab.exp.withtank.**ExperimentWithTank**(*rhos=None*, *zs=None*, *params=None*, *description=None*, *str_path=None*)

Bases: *fluidlab.exp.base.Experiment*

Represent an experiment with a tank.

Parameters **(for the __init__ method)**

rhos : array_like, optional

Density array.

zs : array_like, optional

Position array.

params : dict, optional

Contain parameters (*rhos* and *zs* can be given in this dictionary. Other parameters can be added and will also be saved.)

description : str, optional

A description of the experiment.

str_path : str, optional

A string related to the path where the experiment is saved or will be saved.

Notes

There are two modes of creating an *ExperimentWithTank*:

1. if *str_path* doesn't point on an already saved experiment, a new experiment is created (the instance variable *first_creation* is True). In this case, the parameters *rhos* or *zs* have to be given (either directy or through the dictionary *params*).
2. if *str_path* points on an already saved experiment, the instance variable *first_creation* is False and the experiment is loaded.

Note that if you want to load an already saved experiment, it is more convenient to use the function `fluiddyn.load_exp()` like so:

```
exp = fld.load_exp('2014-03-25_12-43-48')
```

Attributes

tank	(*fluidlab.objects.tanks.StratifiedTank*) Contains the informations on the tank and the density profile.
first_creation	(bool) False if the experiment has not been loaded from the disk.
params	(dict) Containing parameters.
description	(str) A description of the experiment..
path_save	(str) The absolute path of the directory associated with the experiment.
name_dir	(str) Name of the directory associated with the experiment.
time_start	(str) Coding the time of creation.

_create_self_params(*params*)

Calculate some parameters.

First, call the function *_create_self_params* of the inherited class. Then, update the instance variable *params* with the viscosity ν (m^2/s) and the maximum density difference $\Delta\rho = \max\rho - \min\rho$ (g/cm^3).

Parameters **params** : dict

Containing parameters.

_create_tank()

Create the instance variable representing the tank.

Here, *tank* represents a simple stratified tank (*fluidlab.objects.tanks.StratifiedTank*).

_save_tank()

Save the object representing the tank.

_load_tank(*verbose=False*)

Load the object representing the tank.

Functions

Classes

ExperimentWithTank([rhos, zs, params, ...])	Represent an experiment with a tank.

- a package with classes representing Taylor-Couette experiments:

taylorcouette

- other very simple classes derived from *fluidlab.exp.withtank.ExperimentWithTank* and fluidlab.exp.withconductivityprobe.ExpWithConductivityProbe, respectively:

doublediffusion	Experiments on double diffusion (*fluidlab.exp.doublediffusion*)
vertduct	

fluidlab.exp.doublediffusion

Experiments on double diffusion (fluidlab.exp.doublediffusion)

Provides:

class fluidlab.exp.doublediffusion.**DoubleDiffusion**(*zs=None*, *rhos=None*, *params=None*, *description=None*, *str_path=None*)

Bases: *fluidlab.exp.withtank.ExperimentWithTank*

Represent an experience on the double diffusion instability.

See the documentation of the inherited class.

Classes

DoubleDiffusion([zs, rhos, params, ...])	Represent an experience on the double diffusion instability.

Classes

2.4 fluidlab.postproc

2.4.1 Post-processing

serieofarrays	Serie of arrays (*fluidlab.postproc.serieofarrays*)
uvmat	Uvmat interface (*fluidlab.postproc.uvmat*)

fluidlab.postproc.serieofarrays

Serie of arrays (fluidlab.postproc.serieofarrays)

Provides:

class fluidlab.postproc.serieofarrays.**SerieOfArrays**(*path*)
Bases: object

Serie of arrays used for post-processing.

Parameters (for the __init__ method)

path : str

The path of the base directory or of a file example.

Notes

???

Attributes

path_dir	(str) The path of the base directory.

Functions

Classes

SerieOfArrays(path)	Serie of arrays used for post-processing.
SerieOfArraysFromFiles(path[, index_slices])	Serie of arrays saved in files (images, netcdf, etc.).
SeriesOfArrays(serie_arrays, ...)	

fluidlab.postproc.uvmat

Uvmat interface (fluidlab.postproc.uvmat)

From matlab, something like:

```
system(['python -m fluidlab.postproc.uvmat ' path_to_instructions_xml])
```

Functions

main()

Classes

ActionAverage(instructions)	Compute the average and save as a png file.
ActionBase(instructions)	
InstructionsUVMAT(**kargs)	

CHAPTER 3

Scripts and examples

FluidLab also comes with scripts and examples. They are organised in the following directories:

bin	Scripts for command-line utilities
scripts	Example scripts

3.1 bin

3.1.1 Scripts for command-line utilities

fluid_stop_pumps.py: Quickly stop the pumps!

3.2 scripts

3.2.1 Example scripts

taylor_couette	Scripts for preparing and carrying out Taylor-Couette experiments

scripts.taylor_couette

Scripts for preparing and carrying out Taylor-Couette experiments

Some of these scripts should be modified to do useful things. Thus, I would strongly advise to open and understand them before running them!

These scripts can be used for the different steps that have to be done to run particular types of experiments. Typically, the steps can be:

1. Chose consistent parameters in one of the files params_creation_*.py, which defines some parameters that can be modified. You should run it in order to verify the consistency of the parameters.
2. Create an experiment object with the parameters contains in the corresponding params_creation_*.py file (create_exp_*.py).
3. Chose a "working" experiment by modifying str_path in str_path_working_exp.py, which is used for loading the corresponding experiment by the other scripts.

4. Fill the tank with the wanted profile (fill_tank.py).
5. Move the traverse and test the probe (traverse_and_probe.py).
6. Run the experiment (run_exp.py).
7. Plot some results (plot_profiles.py)

The scripts are now described in more details:

params_creation_TC_lin.py: parameters for creation of an ILSTaylorCouetteExp This file is imported by create_exp_TC_lin.py and should be modified. It should be run to test the consistency of the parameters.

str_path_working_exp.py: just define a string *str_path*. This string is used by other scripts in the same directory for loading the corresponding experiment (denoted as the "working" experiment).

fill_tank.py: fill the tank of the "working" experiment. See *fluidlab.objects.tanks*.

traverse_and_probe.py: for moving the traverse and testing the probe. See *fluidlab.objects.traverse* and *fluidlab.objects.probes*.

plot_profiles.py: plot profiles of the "working" experiment. See `fluidlab.exp.withconductivityprobe.ExpWithCor`

CHAPTER 4

More

4.1 To do list

FluidLab is still in a planning stage so there is still A LOT to do!!

4.1.1 Long term

- interface for Arduino boards (see the wikipedia article and the Python module Arduino).

4.1.2 Questions

- String, unicode and byte in Python 2 and 3.

4.1.3 Inline to do items

Todo

Improve `fluidlab.exp.session.Session` to produce a nice file *session.h5*.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/fluidlab/envs/latest/local/lib/python2.7/site-packages/fluidlab-0.0.2a0-py2.7.egg/fluidlab/exp/session.py:docstring of fluidlab.exp.session, line 4.)

Todo

Write drivers for some "VISA instruments"...

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/fluidlab/envs/latest/local/lib/python2.7/site-packages/fluidlab-0.0.2a0-py2.7.egg/fluidlab/instruments/__init__.py:docstring of fluidlab.instruments, line 36.)

Todo

Data acquisition board drivers.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/fluidlab/envs/latest/local/lib/python2.7/site-packages/fluidlab-0.0.2a0-py2.7.egg/fluidlab/instruments/daq/__init__.py:docstring of fluidlab.instruments.daq, line 4.)

Todo

DAQmx interface and drivers (using Comedi API?)...

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/fluidlab/envs/latest/local/lib/python2.7/site-packages/fluidlab-0.0.2a0-py2.7.egg/fluidlab/instruments/daq/daqmx.py:docstring of fluidlab.instruments.daq.daqmx, line 4.)

Todo

Verify potential bug due to the fact that values are class attributes.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/fluidlab/envs/latest/local/lib/python2.7/site-packages/fluidlab-0.0.2a0-py2.7.egg/fluidlab/instruments/drivers.py:docstring of fluidlab.instruments.drivers, line 4.)

Todo

Work on the documentation of *fluidlab.instruments.features*.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/fluidlab/envs/latest/local/lib/python2.7/site-packages/fluidlab-0.0.2a0-py2.7.egg/fluidlab/instruments/features.py:docstring of fluidlab.instruments.features, line 4.)

Todo

Firewire interface and drivers.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/fluidlab/envs/latest/local/lib/python2.7/site-packages/fluidlab-0.0.2a0-py2.7.egg/fluidlab/instruments/firewire/__init__.py:docstring of fluidlab.instruments.firewire, line 4.)

Todo

Verify if IEC60488 drivers work and are convenient (which names for the functions and values?)...

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/fluidlab/envs/latest/local/lib/python2.7/site-packages/fluidlab-0.0.2a0-py2.7.egg/fluidlab/instruments/iec60488.py:docstring of fluidlab.instruments.iec60488, line 4.)

Todo

Solve an evident bug in this function (should be easy).

(The original entry is located in [u'Agilent34970a.write_vdc(channel, value)', u':module: fluidlab.instruments.multiplexer.agilent_34970a', u'', u'Write DC-Voltage on specified AO channel.', u'', u'.. todo::', u'', u' Solve an evident bug in this function (should be easy).'], line 5.)

Todo

Sound interface and drivers...

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/fluidlab/envs/latest/local/lib/python2.7/site-packages/fluidlab-0.0.2a0-py2.7.egg/fluidlab/instruments/sound/__init__.py:docstring of fluidlab.instruments.sound, line 4.)

Todo

Solve a bug loading the tank (about values Rin, Rout...).

(The original entry is located in docstring of fluidlab.objects.tanks, line 7.)

Todo

Help install Linux GPIB

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/fluidlab/checkouts/latest/doc/linuxgpib.rst, line 4.)

4.2 Changes

4.2.1 0.0.1a

- Split the package fluiddyn between one base package and specialized packages.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

b

f

s

b

f

s

Symbols

A

B

C

D

E

F

G

H

I

T

U

V

W