pvlab

Release 0.1.0.dev7

Silva J.P.

Dec 09, 2021

TABLE OF CONTENTS

CHAPTER

ONE

LICENSE

1.1 License: bsd-3-clause

CHAPTER

TWO

THE PVLAB PROJECT

2.1 Introduction

PVLAB is a project devoted to the development and improvement of scientific software for the measurement, calibration and modeling of the performance of photovoltaic devices and solar sensors. PVLAB package born from the efforts in data treatment performed during the calibration of pyranometers at the Laboratory of Photovoltaic Solar Energy (PVLab) of the Research Center for Energy, Environment and Technology (CIEMAT) in Madrid, Spain. In next releases, `pvlab` will provide sets of tools, mainly consisting in classes and functions, to perform the data treatment for the calibration of pyranometers and other type of solar sensors and photovoltaic devices. Eventually, `pvlab` will try to widen its scope to further calibration procedures of solar sensors and photovoltaic devices.

2.2 History

The origin of `pvlab` is a python tool, named `calibration`, which is being developed since 2019 in PVLab-CIEMAT for its own use. It was originally designed to manage the big amount of data generated during the outdoor measurements, while performing the routine calibration of pyranometers.

Soon, both the *Python programming language* and the `calibration` tool themselves proved to be quick and reliable methods for data treatment. Gradually, the code grew in complexity, whereas new functionalities were being enabled. Indeed, to the basic requirements of data *I/O* and a first block of core calculations, some others joined, like fine data-filtering, determination of error sources and total uncertainty, tools for generation of reports, graphics and further calibration records.

Finally, when it was concluded the development of the version 2.0.0 of the application `calibration`, it became clear that a formal package should be released, separately from the former tool. By doing so, some of the resources created are now at disposal of the scientific community, under a 3-clause BSD License.

2.3 Development

One procedure chosen for the early development of `pvlab` is that, as functions and classes created for its use at the lab are being adapted from their specific purpose to address more general cases, and their robustness and performance is considered sufficiently tested, they will be progresively incorporated to the `pvlab` library.

In order to clarify the features and abilities of the objects created, docstrings of relevant functions or classess contain examples, which have been verified with the python built-in package `doctest`. In addition, there is a *test_[module]* ready for each one, checked by using the *unittest* built-in package.

On the other hand, author's hope is that `pvlab` will eventually turn into a **community-developed library**, so contributions and constructive comments are welcome. At this respect, `pvlab` adopts the aim of providing resources in the

context of measurement, calibration, determination of uncertainty, validation techniques and potentially, many other utilities for the improvement of data treatment for solar sensors and photovoltaic devices.

In the long term, a more general purpose lies in the background, which is the advance of data science and the development of software projects for scientific purposes, even "knocking at the doors" of data mining, machine learning and deep learning techniques.

CHAPTER

THREE

RELEASE HISTORY

See what's new in the latest release of `pvlab` project.

(check README.rst for a general description of the project and its content).

3.1 Release 0.1.0.dev1

Wellcome to the first release of `pvlab` software package!!

In the present release, this library provides a set of tools in order to facilitate the treatment of data for the outdoor calibration of pyranometers.

It is inspired in the calibration procedure corresponding to the *outdoor calibration* sections in ISO 9847 International Standard.

Mainly, it contains convenient functions and classes that help to perform the data treatment. These functions can be grouped together in order to achieve complex tasks like data filtering, determination of responsivities of sensors or the calculation of the total uncertainty.

Eventually, `pvlab` will try to widen its scope to further calibration procedures of solar sensors and photovoltaic devices.

3.2 Release 0.1.0.dev2

Minor changes and bug corrections.

3.3 Release 0.1.0.dev3

Minor changes and bug corrections.

3.4 Release 0.1.0.dev4

Minor changes and bug corrections.

3.5 Release 0.1.0.dev5

1. Added module `dictmarker` (a unique file `dictmaker.py`) containining **two new functions**:

```
get_dict(file:  TypeVar('File', str, io.StringIO), dtype:  str, sep:  str = ':',
isStringIO: bool = False) -> dict:
```

and:

```
get_dicts_list(filelist:  Iterable[str], dtypelist:  Iterable[str], isStringIO:
Iterable[bool], sep:  str = ':')  -> dict:
```

New functions are intended to read files of parameters and convert each data file into a python dictionary.

(see description and examples of use in `pvlab`'s documentation).

2. A new section `I/O Management` has been added to documentation.
3. Some parts of Documentation in section **Math operations** have been rewritten, including titles.
4. Some parts of Documentation in section **Create Dataframes** have been rewritten, including titles.

3.6 Release 0.1.0.dev6

1. Added test cases for module `dictmaker` (file test_dictmaker.py). Test classes derive from `unittest.TestCase` class.

3.7 Release 0.1.0.dev7

1. Added module `channels` (file `channels.py`). It contains two fuctions:
    1. Function `set_channels(numbers:  Iterable[int], names:  Iterable[str], nameafter: bool = True) -> Iterable[str]`. See **`set_channels`_** documentation for further information.
    2. Function `set_channels_grouped(numbergroups:  Iterable[list], namegroups: Iterable[list], nameafter:  bool = True, unify:  bool = True, init_channels: list = []) -> Iterable[str]`. See **`set_channels_grouped`_** for further documentation.

_set_channels: https://pvlab.readthedocs.io/en/pvlab-0.1.0.dev7/usage/en/tools_io.html#function-set-channels
_Set_channels_grouped: https://pvlab.readthedocs.io/en/pvlab-0.1.0.dev7/usage/en/tools_io.html#function-set-channels-grouped

1. Added license files for third-party packages.
2. Added test module for functions in `channels` (file `test_channels.py`).

CHAPTER

FOUR

INSTALLATION

4.1 From Python installer

From pip, proceed as follows:

First of all, if a virtual environment (e.g. *myvenv*) has been previously created, and it is the desired working environment, it should be activated. You can do it by typing from **Terminal** (in MacOS©/Unix):

```
source myenv/bin/activate
```

or, from a **system console** in Windows™ platforms:

```
.\myvenv\Scripts\activate
```

Also, the python installer `pip` itself should be updated:

```
python -m pip install --upgrade pip
```

Note: Be sure the python root folder is present in the system path.

Finally, the `pvlab` package can be installed:

```
python -m pip install --upgrade pvlab
```

4.2 From Anaconda IDE

When using the Anaconda IDE, `pvlab` can be installed both by typing:

```
conda install pvlab
```

or, alternatively, from the Anaconda™ navigator, following the sequence: *Environments* > *Search Packages*, and selecting the `pvlab` package.

CHAPTER

FIVE

CREATE DATAFRAMES

Provide tools to manage dataframes in the context of calibration.

It contains the following python modules:

5.1 DataFrames from dicts

Perform type-conversion and *pretty-print* operations for dictionaries.

It contains the following functions:

5.1.1 Function dict_to_df

`pvlab.dataframes.dfdicts.`**`dict_to_df`**(*dictionary: dict*, *columns: list*) → pandas.core.frame.DataFrame
Re-arrange a dictionary to become a pandas dataframe. It performs a type conversion of a dictionary (e.g. a dictionary that represents some kind of valid time intervals), returning a pandas.DataFrame.

Code examples:

When correct parameters are provided, it **returns** a `pandas.DataFrame` object:

Example 1: correct use of function `pvlab.dataframes.dfdicts`.

```
>>> from pvlab.dataframes.dfdicts import dict_to_df

>>> dates = {'START_1': (2021,5,5,8,1,0), 'END_1': (2021,5,6,22,52,0)}

>>> columns = ['%Y', '%m', '%d', '%H', '%M', '%S']

>>> dict_to_df(dates, columns)
           %Y  %m  %d  %H  %M  %S
START_1  2021   5   5   8   1   0
END_1    2021   5   6  22  52   0
```

Otherwise, a `ValueError` is raised when the length of `columns` does not match the length of the values of the given dictionary:

Example 2: list of columns shorter than expected.

```
>>> from pvlab.dataframes.dfdicts import dict_to_df

>>> columns = ['%Y', '%m', '%d', '%H', '%M']
```

(continues on next page)

(continued from previous page)

```
>>> dict_to_df(dates, columns)
Traceback (most recent call last):
    ...
ValueError: Length of columns list is equal to 5, but has to be equal to 6.
```

5.1.2 Function print_dict

Example 3: list of columns longer than expected.

```
>>> from pvlab.dataframes.dfdicts import dict_to_df

>>> columns = ['%Y', '%m', '%d', '%H', '%M', '%S', '%mS']

>>> dict_to_df(dates, columns)
Traceback (most recent call last):
    ...
ValueError: Length of columns list is equal to 7, but has to be equal to 6.
```

pvlab.dataframes.dfdicts.**print_dict**(*dictionary: dict*, *columns: list*, *title: str = ''*) → None

Prettyprint a dictionary of dates, adding a title. It appears to be similar to dict_to_df, but print_dict just print, (it does not return a pandas.DataFrame object, it returns None):

Example 4: correct use of function `pvlab.dataframes.print_dict`.

```
>>> from pvlab.dataframes.dfdicts import print_dict

>>> dates = {'START_1': (2021,5,5,8,1,0), 'END_1': (2021,5,6,22,52,0)}

>>> columns = ['%Y', '%m', '%d', '%H', '%M', '%S']

>>> title = 'Valid time intervals'

>>> print_dict(dates, columns, title)
Valid time intervals
--------------------
           %Y  %m  %d  %H  %M  %S
START_1  2021   5   5   8   1   0
END_1    2021   5   6  22  52   0
```

CHAPTER

SIX

MATH OPERATIONS

Provide tools for mathematical or statistical operations.

6.1 Composition

Composition of quantities, intended for statistical purposes.

6.1.1 Function module

pvlab.math.module.**module**(**components: Sequence[float]*) → float:

Calculate the module of a vector, or the result of a quadratic composition, given its components. It supports n-dimensional components. It is useful when working with versions of python older than 3.8 (e.g. for determining the total *type B* uncertainty from homogeneous contributions).

Note: In python v3.8, it was added support for n-dimensional points in built-in function `math.hypot`. Then, in python 3.10, accuracy was improved. Here for further information.

Example 1: correct use of function `pvlab.math.module`.

```
from pvlab.math.module import module

components = [5, 8, 3, 6]

round(module(*components), 3)
11.576
```

Example 2: `components` must be `float` or `int` types.

```
from pvlab.math.module import module

components = [5, 3, 8, '6']
module(*components)

Traceback (most recent call last):
    ...
TypeError: components items must be int or float types.
```

CHAPTER

SEVEN

I/O MANAGEMENT

Provide tools for data input/output.

7.1 Create dicts from Source Files

Generate python dictionaries from multiple files and data types.

New functions are intended to read files of parameters and convert each data file into a python dictionary. Different type of parameters should be located in different files, for better performance.

It contains the following functions:

7.1.1 Function get_dict

pvlab.io.dictmaker.**get_dict**(*file: TypeVar('File', str, io.StringIO)*, *dtype: str*, *sep: str = ':'*, *isStringIO: bool = False*) → dict:

Generate a python dictionary from a file of parameters.

Function `get_dict` reads a file that contains parameters in the form *[name]: [value]* (or in other general form *[name][sep] [value]*, if specified). It requires specifying the type of data contained in the file, admitting types `int`, `float`, `tuple` or `str` (unknown data types are admitted, but they will be parsed as *raw* strings). Originally designed for data-acquisition purposes. It also admits `io.StringIO` objects instead, if argument `isStringIO` is specified as `True` (e.g. useful for exemplification purposes).

Example 1: `int` type arguments:

```
from io import StringIO
from pvlab.io.dictmaker import get_dict

data = "readings:21\nminG:600\nrefG:1000"
settings = get_dict(io.StringIO(data), dtype='int', isStringIO=True)
# ... argument "io.StringIO(data)" can be replaced by a file name.

settings
{'readings': 21, 'minG': 600, 'refG': 1000}
```

Example 2: `str` type arguments:

```
from io import StringIO
from pvlab.io.dictmaker import get_dict

```

(continues on next page)

(continued from previous page)

```
data = "man.:'manufacturer'\nmod.:'model'\nsn.:'seriesnr'"
mydict = get_dict(io.StringIO(data), dtype='str', isStringIO=True)

mydict
{'man.': 'manufacturer', 'mod.': 'model', 'sn.': 'seriesnr'}
```

7.1.2 Function get_dicts_list

pvlab.io.dictmaker.**get_dicts_list**(*filelist: Iterable[str]*, *dtypelist: Iterable[str]*, *isStringIO: Iterable[bool]* = *False*, *sep: str* = *':'*) → dict:

Generate a list of dicts from a list of parameter files or `io.StringIO` objects.

It calls the previous function `get_dict` recursively, from correlative values of `filelist` and `dtypelist` arguments.

Example 3: source objects containing both `float` and `str` arguments.

```
from io import StringIO
from pvlab.io.dictmaker import get_dicts_list

floatdata = "maxdev:0.02\noffsetthreeshold:2.0"
filters = io.StringIO(floatdata)  # StringIO_1 (or filename_1)

strdata = "mode_refpyr:'voltage'\nmode_dut:'currentloop'"
calmode = io.StringIO(strdata)  # StringIO_2 (or filename_2)

isstringio = ['True', 'True']  # io.StringIO objects? (defaults False)
caliblist = get_dicts_list([filters, calmode], ['float', 'str'], isStringIO=isstringio)
↪# it returns a list of python dicts.

caliblist[0]  # ...data from StringIO_1 (or filename_1)
{'maxdev': 0.02, 'offsetthreeshold': 2.0}

caliblist[1]  # ... data from StringIO_2 (or filename_2)
{'mode_refpyr': 'voltage', 'mode_dut': 'currentloop'}
```

7.2 Create a list of channels

Provide tools to facilitate the selection of relevant data.

It contains the following functions:

7.2.1 Function set_channels

pvlab.io.channels.**set_channels**(*numbers: Iterable[int]*, *names: Iterable[int]*, *nameafter: bool = True*) → Iterable[str]:

Generate a list of channel names from a set of numbers and a set of names.

It is designed to automate the selection of specific active channels,*e.g.* within data frames containing a big number of data columns.

Given a list of *n* numbers (`numbers`) and *m* names (`names`), it generates a list of channel names in the form:

[[**number_1**][**name_1**], [**number_1**][**name_2**], ..., [**number_1**][**name_m**], , ..., , ..., , ..., [**number_n**][**name_1**], **number_n**][**name_2**], ..., [**number_n**][**name_m**],]

If argument `nameafter` is True (by default), names are added after numbers. Otherwise, names are added before numbers.

Item types (both numbers and names) must be convertible into strings.

If the `numbers` list is empty, it directly retuns the `names` list.

In the same way, if the `names` list is empty, it returns the `numbers` list. Anyway, it performs a previous conversion into `str` types.

At least one list must not be empty.

Example 1: function `set_channels`.

```
from pvlab.io.channels import set_channels

numbers = [101, 115, 207]
names = ['(Time stamp)', '(VDC)']

set_channels(numbers, names).__class__ == list
True
len(set_channels(numbers, names)) == 6
True

channels = set_channels(numbers, names)

channels[:2]
['101(Time stamp)', '101(VDC)']
channels[2:4]
['115(Time stamp)', '115(VDC)']
channels[4:]
['207(Time stamp)', '207(VDC)']
```

7.2.2 Function set_channels_grouped

pvlab.io.channels.**set_channels_grouped**(*numbergroups: Iterable[list]*, *namegroups: Iterable[list]*, *nameafter: bool = True*, *unify: bool = True*, *init_channels: list = []*) → Iterable[str]:

Generate a list of channels from multiple lists of numbers and names.

It applies recursively the fuction `set_channels` to multiple sets of numbers and names. Therefore, it allows the generation of multiple channel names that contains different names.

Argument `nameafter` possesses the same significance than in `set_channels`, and defaults to `True`.

If argument `unify` (defaults `True`) is True, function returns a unique list of channels. If it is `False`, function returns separate lists.

If arguments 'numbergroups' and 'namegroups' are not of the same length, the shorter one marks the end of parsing, and further terms in the larger argument are neglected, so `numbers3` argument in an entry like: `set_channels_grouped([numbers1, numbers2, numbers3], [names1, names2])` is neglected, and so it is `names3` argument in entry `set_channels_grouped([numbers1, numbers2], [names1, names2, names3])`.

Example 2: function `set_channels_grouped`.

```
from pvlab.io.channels import set_channels_grouped

numbers1 = [101, 102, 104]
numbers2 = [201, 202, 204]

names1 = ['(Time stamp)', '(voltage)']
names2 = ['(Time stamp)', '(temperature)']

channels = set_channels_grouped([numbers1, numbers2], [names1, names2])

# let's do some checking:
channels.__class__ == list  # it should return a list
True
channels[:2]  # the first two elements ...
['101(Time stamp)', '101(voltage)']
channels[-2:]  # ... and the last two.
['204(Time stamp)', '204(temperature)']

# On the other hand, being ...
len_1 = len(numbers1) * len(names1)
# and ...
len_1 = len(numbers1) * len(names1)
# the total amount of items generated should be ...
len(channels) == len_1 + len_2
True

# Finally, all items must be strings...
[type(item) for item in channels] == [str] * len(channels)
True
```

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX