PIGOR Documentation

Nico Einsidler

Oct 17, 2019

CONTENTS:

Pigor is a lightweight analysis tool for the the polarimeter instrument NEPTUN beam port of the 250kW research reactor hosted at Atominstitut of TU Wien, Austria. For more information visit our homepage.

CHAPTER

ONE

HOW TO INSTALL PIGOR

1.1 Installing PIGOR

Assuming you have a running installation of **Python 3.7** or higher, to install PIGOR please follow these steps:

1. **Download the source from Bitbucket with `git clone https://github.com/nicoeinsidler/pigor.git`. This**

    - `measurement.py`
    - `pigor.py`
    - `README.md`
    - `requirements.py`
    - `requirements-dev.py`
    - `requirements-measurement.py`

2. Install the requirements with:

    ```
    pip install -r requirements.txt
    ```

Note: If you are using two separate python 2.x and 3.x installations, you might need to use `pip3` instead of `pip`.

You can now run PIGOR with the following command in the folder where the `pigor.py` and `measurement.py` files are located:

```
python pigor.py
```

Note: If you only want to use the `measurement.Measurement` class, you can only install it's dependencies with `pip install -r requirements-measurement.txt`. But since most dependencies are shared, it won't make much of a difference.

1.2 Installing only the Measurement Class

Warning: This is not maintained yet. Please use this feature with caution and build it first.

In some cases the user may only want to install the measurement module to use the Measurement class as one of their python installations modules. A zip or tarball file should be provided for this use case. See deployment-measurement-module.

To install (and copy into the user's installed modules) simply unpack the zip or tarball. Head over to its directory and type from there:

```
python setup.py install
```

1.3 Installation for Developers

A known working set of python modules used to create PIGOR can be found in `requirements-dev.txt`. To install all these requirements, simply type:

```
pip install -r requirements-dev.txt
```

It is recommended to use a virtual environment.

MEASUREMENT NAMING CONVENTION

In order for the Measurement class to work properly, the user should follow this naming convention:

`YYYY_MM_DD_HHmm_IDENTIFIER_TYPE.dat` (not case sensitive)

where…

- **`IDENTIFIER` denotes the type of measurement or the reason for the measurement. The following values are common and**
  - `dcX#`: DC coil (for the magnetic field in x-direction) scan for the DC coil with the number #. A sine fit is automatically applied, when `TYPE` is not set.
  - `dcZ#`: DC coil scan in z-direction (compensation field) for DC coil number #. Fitted with a polynomial of order 5 by default.
  - `pos#`: position scan with stage number #. Fitted with a Gaussian by default.
- **`TYPE` specifies and overrides the type of fitting/analysis that should be applied by PIGOR. Possible values:**
  - **fitting types**
    - `poly5`: polynom of order 5
    - `sine_lin`: sine plus additional added linear function
    - `sine`: sine
    - `gauss`: Gauß fit
  - **special analysis types**
    - `pol`: specifies that this measurement is a degree of polarisation measurement. This will automatically calculate the degree of polarisation for any given number of points (number of data points must be multiple of four)

Types can be combined within a types group (gauss and pol for example, like “gauss_pol” or “pol_gauss”).

CHAPTER

THREE

QUICKSTART GUIDE

Note: Please refer to the *Installing PIGOR* page for more information on the installation.

Suppose you have some measurement data files with in a directory. The hierachy looks like this:

- **measurements**
  - **2019-01**
    - 2019-01-01-dc1x.dat
    - 2019-01-04-dc1z.dat
    - 2019-01-05-dc1z.dat
  - **2019-02**
    - 2019-02-10-dc2x.dat
    - 2019-02-11-dc2x.dat
    - 2019-02-14-dc2z.dat
    - 2019-01-19-dc2z.dat
    - 2019-01-20-dc2z.dat
    - 2019-01-28-dc1x.dat
  - **2019-03**
    - 2019-02-02-dc3x.dat
    - 2019-02-03-dc3z.dat

Note: Here we will assume that `python` will start Python 3. On some installations it must be explicitly started with `python3`.

In order to analyse these files with PIGOR, head to the directory where PIGOR is located. Start it by executing the command `python pigor.py`. If it is the first startup it will go through some questions regarding your analysis setup. The most important question is the first one: Where should PIGOR look for files to analyse? This is the folder where all your measurements are located in and will be refered as PIGOR's root folder. In our example type in the path to `measurements/`. You can either use absolute paths or relative ones. PIGOR will automatically check if the path actually exists, if not you will be prompted to enter a new one again. It also does not matter if you are using a POSIX type (IEEE Std 1003.1-1988) or as a DOS type path. (POSIX uses / and DOS uses as separators)

If PIGOR does not ask these question on startup, they have already been configured. Either delete the `pigor.config` file in the directory where `pigor.py` is located, or just use the command [i] within PIGOR to reconfigure it.

Now you should see something like that, depending on PIGOR's version:

```
$ python pigor.py

======================
Welcome to PIGOR v1.1.
======================

i ... init
h ... print_help
a ... analyse_files
j ... create_index
r ... remove_generated_files
x ... print_root
q ... quit PIGOR v1.1

PIGOR v1.1 will look for measurement files in D:\measurements.

If you need more information about a command, just type h + [command] + <ENTER>
to get more help. For example: h + a + <ENTER>.

Please type a command you want to perform and press <ENTER>.
```

On startup PIGOR is printing the root, which in this case is `D:\measurements`. It will also include a list of available commands. Each command can be triggered by pressing the corresponding letter followed by an <ENTER>. Some commands allow more options, each separated by a space.

We will describe these commands by [cmd] where cmd stands for the specific command.

To get more help on a specific command, just type [h] + [cmd]. For example typing `h a` will give us the help on `analyse_files()`. Let's try it:

```
Please type a command you want to perform and press <ENTER>.
h a
a:

Analyses all given files in list. This function can be used by the command [a].

    :param filepaths:   list of files to analyse with
                        their relative dir path added

    .. todo:: Change to no override mode. measurement.Measurement.plot(override=False)
    .. todo:: a + today => only analyse files for today
    .. todo:: a + override => override=True
```

This is the same help text as found in this documentation.

We can now analyse our files with [a]. You can see on which file PIGOR is currently working on. If an error occurs, you will see it as well and PIGOR will skip the file.

After PIGOR is done analysing files, you may want to access these files. Use the [j] command to create an index to quickly go through all the files that have been analysed. Now you should see a new file has been created:

- **measurements**
    - index_pigor.html

  - 2019-01
  - 2019-02
  - 2019-03

Open `index_pigor.html` to see the list. From there you can review the original data and the files that PIGOR created from this.

CHAPTER

FOUR

PIGOR

4.1 What PIGOR does

PIGOR ('Python IGOR' = PIGOR) aims to help physicists on the NEPTUN beamline to quickly extract the needed information when measuring and configuring or preparing an experiment. It will go through all files in its root folder and will continue to **look for files in all subdirectories recursively** as well. It will then **auto detect**[1] **the type of measurement** and guess what the user wants to know. After analysis of all files, **additional files will show up alongside the original measurement files**:

- .png file: plot of the data
- .md file: usefull information gathered about the measurement in plain text as markdown
- .html file: same content as the markdown file, but nicely viewable in a modern browser

4.2 PIGORs inner workings

[1] This will only work if the correct naming convention is used.

CHAPTER

FIVE

THE MEASUREMENT CLASS

5.1 Flow at Startup

1. read the data
2. detect measurement type
3. read position file if a position file exists
4. clean up and description gathering
5. select columns for plotting

5.2 Flow when plotting

5.3 Class Usecases

There are many ways to interact with or use the Measurement class. Here are the three main ways:

5.4 Methods

Todo: Method attributes are shown, but value is always None.

CHAPTER

SIX

MEASUREMENT CLASS 2.0

Due to some design flaws in the existsing `measurement.Measurement` class, it will be rewritten from ground up using already existsing libraries. This undertaking will lead to a cleaner version and will make it possible to adapt `measurement.Measurement` more easily for the interferometer experiments.

6.1 Structure

The new immproved structure will make use of:

- LMFIT package
- pandas dataframe

So that the fitting will be done with LMFIT models, whereas the data is handled in pandas dataframes. This creates huge advantages for the developer as well as for the user.

6.2 Previous Development

There have been efforts previously to build a unique own modular `measurement.Measurement` class by Nico. These efforts can be examined in the branch improvements/core.

These following elements were attemted to build:

- data column: has data and a head, knows its name etc.; functions can easily be applied to it
- fit object: used for fitting and finding bounds; each instance can have its own bounds
- **data set: these objects can be plotted by Measurement, so Measurement will try to create one of those objects; they consist**
    - data columns objects
    - fit objects

Column Class

Methods:

- `reverse()`: reverse order of data
- `__init__(self, desc, data)`
- `__repr__()`: plots `'<column object 'desc' of lenght len(data)>'` or something like that

Variables:

- `columns.data`: holds the data as numpy array in float64

- `columns.desc`: holds the name of the columns heading as string

Fit Function Class

Method:

- `fit_function()`
- `find_bounds()`: tries to find the bounds
- `bounds`: holds the bounds to be used when fitting as array of tuples

Variables:

- `parameters`: dictionary holding the names of the parameters and the parameters themselves

Fit Class

Variables:

- `type` with which function the fit should be carried out, string
- `popt`
- `pcov`

Methods:

- `fit()`

Data Set Class

This object stores data points (lists or Column objects) to form a data set. It must contain at least two data point lists. These lists must have the same number of elements, if not the lists that don't have enough elements will get padded with 0.

Variables:

- `desc`: a description of what the data set describes (optional)
- `data`: data is stored in list; len(list) > 1;

It is obvious to every reader that indeed almost all of this functionality is already included in the python package pandas and lmfit. This led to the conclusion that improvements/core won't be maintained any longer.

6.3 The brand new Measurement Class

The brand new measurement class will include only the following featues:

- ready-to-use lmfit fit models
- fit() method to actually produce a fit using lmfits minimize()
- plot() method for implementing plotting functionality
- contrast() to calculate the contrast
- degree_of_polarisation() to calculate the degree of polarisation
- read() to read data in a very generic way
- export_meta() to export meta data

Its child classes will then implement the experiment specific data handling details like cleaning the data and extending the meta data that will be exported.

CHAPTER

SEVEN

FIT FUNCTIONS

The following fit functions are implemented and can be used within PIGOR or the Measurement class. They can be found in `fit_functions.py`.

`fit_functions.`**`gauss`**(*x*, *a*, *x0*, *sigma*, *export=False*)

Gaussian function, used for fitting data.

Parameters

- **x** – parameter
- **a** – amplitude
- **x0** – maximum
- **sigma** – width
- **export** – enable text output of function

`fit_functions.`**`poly`**(*x*, **args*, *export=False*)

Polynom nth degree for fitting.

Parameters

- **x** (*int, float*) – parameter
- ***args** – list of coefficients [a_N,a_N-1, …, a_1, a_0]
- **export** (*bool or string, optional*) – enable text output of function, defaults to False

Returns returns the polynomial

Return type str, int, float

```
>>> poly(3.4543, 5,4,3,2,1, export='Mathematica')
'5*3.4543^5 + 4*3.4543^4 + 3*3.4543^3 + 2*3.4543^2 + 1*3.4543^1'
```

```
>>> poly(3.4543, 5,4,3,2,1)
920.4602110784704
```

`fit_functions.`**`poly5`**(*x*, *a5*, *a4*, *a3*, *a2*, *a1*, *a0*, *export=False*)

Polynom 5th degree for fitting.

Parameters

- **x** – parameter
- **a5** – coeff
- **a4** – coeff

- **a3** – coeff
- **a2** – coeff
- **a1** – coeff
- **a0** – coeff
- **export** – enable text output of function

Returns function – polynomial 5th degree

`fit_functions.`**`register_fit_function`** (*func*, *bounds=(-inf*, *inf)*)
This decorator registers a new fit function and writes an entry to fit_function_list.

`fit_functions.`**`sine`** (*x*, *a*, *omega*, *phase*, *c*, *export=False*)
Sine function for fitting data.

Parameters

- **x** – parameter
- **a** – amplitude
- **omega** – frequency
- **phase** – phase
- **c** – offset
- **export** – enable text output of function

`fit_functions.`**`sine_lin`** (*x*, *a*, *omega*, *phase*, *c*, *b*, *export=False*)
Sine function with linear term added for fitting data.

Parameters

- **x** – parameter
- **a** – amplitude
- **omega** – frequency
- **phase** – phase
- **c** – offset
- **b** – slope
- **export** – enable text output of function

CHAPTER

EIGHT

SPRINT PLANNING

This site gives a quick overview what will come next. Each sprint should take about 1 week to finish.

8.1 PIGOR

1. **Sprint**
    - ✓ feature: remove all generated files (html, md, png)
    - ✓ feature: introducing a config file (PIGOR start directory, . . .)
    - ✓ improvement: auto create config file if not present
    - ✓ improvement: auto register all functions for help menu (decorators)
2. **Sprint**
    - feature: remove last generated files (html, md, png)
    - feature: remove all html/md or png files
    - ✓ improvement: use JSON for config file
3. **Sprint:**
    - feature: auto run command in specified intervals; syntax maybe: time + [cmd] + <ENTER>

8.2 Measurement Class

1. **Sprint**
    - ✓ improvement: switching from self.y –> self.y[] and self.y_error –> self.y_error[]
    - ✓ [not tested yet] improvement: plot multiple self.y's
    - feature: auto detect interferometer measurements
2. **Sprint**
    - feature: remove all associated files from file system, except the measurement file itself
    - ✓ improvement: auto register all available fit functions via decorators
    - improvement: adding __repr__
3. **Sprint: finish branch `feature/interferometer`**
    - fixing / understanding inheritance of instance variables (see python test file in branch)

- **creating subclasses from Measurement:**
  - Interferometer: adding custom maps to COLUMN_MAPS and overriding clean_data() and detect_measurement()
  - Polarimeter: adding custom maps to COLUMN_MAPS and overriding clean_data() and detect_measurement()

4. Sprint: not planned yet

8.3 Ideas

Building Measurement from ground up with custom objects like:

- data column: has data and a head, knows its name etc.; functions can easily be applied to it
- fit object: used for fitting and finding bounds; each instance can have its own bounds
- **data set: these objects can be plotted by Measurement, so Measurement will try to create one of those objects; they consist**
  - data columns objects
  - fit objects

8.4 Column Class

Methods:

- `reverse()`: reverse order of data
- `__init__(self, desc, data)`
- `__repr__()`: plots `'<column object 'desc' of lenght len(data)>'` or something like that

Variables:

- `columns.data`: holds the data as numpy array in float64
- `columns.desc`: holds the name of the columns heading as string

8.5 Fit Function Class

Method:

- `fit_function()`
- `find_bounds()`: tries to find the bounds
- `bounds`: holds the bounds to be used when fitting as array of tuples

Variables:

- `parameters`: dictionary holding the names of the parameters and the parameters themselves

8.6 Fit Class

Variables:

- `type` with which function the fit should be carried out, string
- `popt`
- `pcov`

Methods:

- `fit()`

8.7 Data Set Class

This object stores data points (lists or Column objects) to form a data set. It must contain at least two data point lists. These lists must have the same number of elements, if not the lists that don't have enough elements will get padded with 0.

Variables:

- `desc`: a description of what the data set describes (optional)
- `data`: data is stored in list; len(list) > 1;

CHAPTER

NINE

TODO LIST

Todo: check if dependencies are correct with dependencies.txt

(The original entry is located in /Users/nicoeinsidler/Code/pigor/doc/index.rst, line 55.)

Todo: Method attributes are shown, but value is always None.

(The original entry is located in /Users/nicoeinsidler/Code/pigor/doc/measurement.rst, line 62.)

CHAPTER TEN

OLD TODOS

Note: This list of ToDos should be integrated into the docstrings or into the sprint planning where those ToDos belong.

Here are all ToDos listed. Feel free to contribute and check this project out on Bitbucket.

- self.x_error: not yet implemented
- a lot of commenting
- error of fit
- verbose mode on/off
- getting PIGOR ready for shipment by creating a setup.py
- better display of self.pcov
- separation between pure functions and functions with context (methods)
- auto comment decorator for functions
- use decorators to auto register fit functions with their input argument list
- setuptools in setup.py
- https://click.palletsprojects.com/en/7.x/quickstart/

CHAPTER ELEVEN

PROJECT DEPENDENCIES

Todo: check if dependencies are correct with dependencies.txt

- numpy
- re
- matplotlib
- os.path
- glob
- difflib
- datetime
- pathlib
- scipy
- markdown

CHAPTER

TWELVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX