kpfm Documentation

Release v0.3-1-g4bbb3ef-dirty

Ryan Dwyer

Jul 29, 2017

Contents

Kelvin Probe Force Microscopy (KPFM) data analysis.

CHAPTER 1

Motiviation

Collect useful code for custom Kelvin probe force microscopy experiments into a single repository. The motivation for using a single repository is prior experience and this article.

CHAPTER 2

Structure

A tentative structure for the repository is

```
kpfm
- __init__.py
- _version.py
- experiment
|   - __init__.py
- tests
|   - __init__.py
|   - test_kpfm.py
- scratch
|   - __init__.py
|   - temporary_code.py
- util
    - __init__.py
    - plot.py
```

Each experiment is contained in its own module. The `util` module collects useful functions and decorators that are useful across a range of modules. Tests are placed in the `tests` module. Scratch contains a place for code that will not be maintained. If a particular experiment or procedure is useful over a longer period, it can be moved to its own experiment module.

CHAPTER 3

Development life cycle of new code

1. Workup data and write provisional code in an IPython notebook
2. **If many data files are going to be processed in this way, move the routine to a new module in `scratch`.**
    - Workups in `scratch` come with no guarantees that they will not break.
    - In practice, it would be useful to create a new copy of the temporary code if there is likely to be significant breakage.
3. If the experiment is useful enough, copy / edit the scratch code and move it to its own experiment folder. In doing so, we commit to leave the experiment in a usable, backward compatible state, which means consistent LabView code for collecting the data, as well as consistent, backward-compatible routines for working up the data.

The idea would be that with a powerful set of utility, signal processing, and data analysis tools in the `kpfm` package, many analyses could be performed quickly at (1) or (2). Even having a file in scratch (perhaps copied occasionally as the data analysis or LabView code changes significantly) is a significant improvement over code copied between IPython notebooks. The other consideration is that it is a large investment of time to maintain code, so that code related to a "one-off" experiment should be saved, but not maintained.

CHAPTER 4

Features

- TODO

Contents:

lockin

This module contains classes and functions for performing digital lock-in amplifier data analysis.

class kpfm.lockin.**LockIn**(*t*, *x*, *fs=None*)

A basic digital lock-in amplifier.

Run an input signal x through a digital lock-in amplifier. A finite impulse response (FIR) lock-in filter can be provided by *lock* or *lock2*, or a custom FIR filter can be used by directly calling *run*. After generating the complex lock-in output, the lock-in can be phased by running *phase*, or *autophase*. After phasing, the lock-in output channels are X, the in-phase channel and Y, the out-of-phase channel.

Parameters

- **t** (*array_like*) – Time array
- **x** (*array_like*) – Input signal array
- **fs** (*float*) – Sampling rate

Example

```
>>> fs = 1000.0
>>> t = np.arange(1000)/fs
>>> A = 1 - 0.1 * t
>>> f = 80 + 0.1 * t
>>> x = A * np.sin(np.cumsum(f)*2*np.pi/fs)
>>> li = LockIn(t, x, fs)
```

We process the data with a 20 Hz bandwidth lock-in amplifier filter.

```
>>> li.lock(bw=20.0)
Response:
f   mag       dB
 0.000 1.000    0.000
10.000 0.996   -0.035
20.000 0.500   -6.022
40.025 0.000  -91.020
80.051 0.000 -113.516
500.000 0.000 -204.987
```

The lock-in amplifier automatically infers the reference frequency. The printed response shows the lock-in amplifier gain at different frequencies. For the output to be valid the gain at the reference frequency must be very small (-60 dB or smaller).

We phase the lock-in amplifier output, and then have the lock-in variables available for use.

```
>>> li.phase()
>>> li('t') # Shortcut for accessing masked version of the signal.
```

__init__(*t*, *x*, *fs=None*)

classmethod from_x(*Cls*, *x*, *fs*, *t0=0*)
Generate the time array internally.

__repr__()

run(*f0=None*, *fir=None*)
Run the lock-in amplifier at reference frequency `f0`, using the finite impulse response filter `fir`.

lock(*bw=None*, *f0=None*, *bw_ratio=0.5*, *coeff_ratio=9.0*, *coeffs=None*, *window='blackman'*)
Standard, windowed finite impulse response filter.

lock_butter(*N*, *f3dB*, *t_exclude=0*, *f0=None*, *print_response=True*)
Butterworth filter the lock-in amplifier output

manual_phase(*phi0*, *f0corr=None*)
Manually phase the lock-in output with phase phi0 (in radians).

absolute_phase(*mask*, *guess=0.0*)
Perform a curve fit

class kpfm.lockin.**FIRStateLock**(*fir*, *dec*, *f0*, *phi0*, *t0=0*, *fs=1.0*)
Lock-in amplifier object which uses an FIR filter, decimates data, and processes data in batches.

Pass data in with the `filt` function. Lock-in amplifier output stored in `z_out`. Time array accessible with the `get_t` function.

Parameters

- **fir** (*array_like*) – finite-impulse-response (FIR) filter coefficients
- **dec** (*int*) – Decimation factor (output sampling rate = input sampling rate /dec)
- **f0** (*scalar*) – Lock-in amplifier reference frequency.
- **phi0** (*scalar*) – Initial lock-in amplifier phase.
- **t0** (*scalar, optional*) – Inital time associated with the first incoming data point. Defaults to 0.
- **fs** (*scalar, optional*) – Input sampling rate. Defaults to 1.

__init__ (*fir*, *dec*, *f0*, *phi0*, *t0=0*, *fs=1.0*)

class kpfm.lockin.**FIRStateLockVarF** (*fir*, *dec*, *f0*, *phi0*, *t0=0*, *fs=1.0*)

Variable frequency lock-in amplifier object which uses an FIR filter, decimates data, and processes data in batches.

Pass data in with the `filt` function. Lock-in amplifier output stored in `z_out`. Time array corresponding to the data in `z_out` accessible with the `get_t` function.

Parameters

- **fir** (*array_like*) – finite-impulse-response (FIR) filter coefficients
- **dec** (*int*) – Decimation factor (output sampling rate = input sampling rate /dec)
- **f0** (*function*) – Lock-in amplifier reference frequency as a function of time
- **phi0** (*scalar*) – Initial lock-in amplifier phase.
- **t0** (*scalar, optional*) – Inital time associated with the first incoming data point. Defaults to 0.
- **fs** (*scalar, optional*) – Input sampling rate. Defaults to 1.

__init__ (*fir*, *dec*, *f0*, *phi0*, *t0=0*, *fs=1.0*)

kpfm.lockin.**freq_from_fft** (*sig*, *fs*)

Estimate frequency from peak of FFT

kpfm.lockin.**parabolic** (*f*, *x*)

Quadratic interpolation for estimating the true position of an inter-sample maximum when nearby samples are known.

f is a vector and x is an index for that vector.

Returns (vx, vy), the coordinates of the vertex of a parabola that goes through point x and its two neighbors.

Example: Defining a vector f with a local maximum at index 3 (= 6), find local maximum if points 2, 3, and 4 actually defined a parabola.

In [3]: f = [2, 3, 1, 6, 4, 2, 3, 1]

In [4]: parabolic(f, argmax(f)) Out[4]: (3.2142857142857144, 6.1607142857142856)

kpfm.lockin.**fir_weighted_lsq** (*weight_func*, *N*)

Return intercept, slope filter coefficients for a linear least squares fit with weight function `weight_func`, using `N` most recent points.

kpfm.lockin.**lock2** (*f0*, *fp*, *fc*, *fs*, *coeff_ratio=8.0*, *coeffs=None*, *window='blackman'*, *print_response=True*)

Create a gentle fir filter. Pass frequencies below fp, cutoff frequencies above fc, and gradually taper to 0 in between.

These filters have a smoother time domain response than filters created with lock.

utility

This module contains useful utility functions, decorators, and plotting helpers that are useful throughout the `kpfm` package.

Should my function go here? If you find yourself wanting it in many IPython notebooks, yes! Just add a docstring and make sure the function is useful on its own (not dependent on any external data, doesn't make assumptions that limit broader usefulness of the code).

kpfm.util.**silent_remove**(*filename*)
If `filename` exists, delete it. Otherwise, return nothing. See http://stackoverflow.com/q/10840533/2823213.

kpfm.util.**align_labels**(*axes_list*, *lim*, *axis='y'*)
Align matplotlib axis labels to the same horizontal (y-axis) or vertical (x-axis) position.

kpfm.util.**color2gray**(*x*)
Convert an RGB or RGBA (Red Green Blue Alpha) color tuple to a grayscale value.

kpfm.util.**txt_filename**(*func*)
Decorator to allow seamless use of filenames rather than file handles for functions that operate on a text file.

To use this decorator, write the function to take a file object as the function's first argument.

kpfm.util.**h5filename**(*func*)
Decorator to allow seamless use of filenames rather than file handles for functions that operate on an HDF5 file.

To use this decorator, write the function to take an HDF5 file handle as the function's first argument.

Example: We create a simple function and HDF5 file.

```
>>> @h5filename
>>> def h5print(fh):
>>>     print(fh.values())
>>>
>>> fh = h5py.File('test.h5')
>>> fh['x'] = 2
```

We can call the function on the file handle

```
>>> h5print(fh)
[<HDF5 dataset "x": shape (), type "<i8">]
```

or call the function on the filename

```
>>> fh.close()
>>> h5print('test.h5')
[<HDF5 dataset "x": shape (), type "<i8">]
```

kpfm.util.**h5ls_str**(*g*, *offset=''*, *print_types=True*)
Prints the input file/group/dataset (g) name and begin iterations on its content.

See goo.gl/2JiUQK.

kpfm.util.**h5ls**(**args*)
List the contents of an HDF5 file object or group. Accepts a file / group handle, or a string interpreted as the hdf5 file path.

kpfm.util.**prnDict**(*aDict*, *br='\n'*, *html=0*, *keyAlign='l'*, *sortKey=0*, *keyPrefix=''*, *keySuffix=''*, *valuePrefix=''*, *valueSuffix=''*, *leftMargin=4*, *indent=1*, *braces=True*)
Return a string representive of aDict in the following format:

```
{
 key1: value1,
 key2: value2,
 ...
 }
```

Spaces will be added to the keys to make them have same width.

Parameters

- **sortKey** (*0 or 1*) – set to 1 if want keys sorted;
- **keyAlign** – either 'l' or 'r', for left, right align, respectively.
- **keySuffix, valuePrefix, valueSuffix** (*keyPrefix,*) – The prefix and suffix to wrap the keys or values. Good for formatting them for html document (for example, keyPrefix='', keySuffix=''). Note: The keys will be padded with spaces to have them equally-wide. The pre- and suffix will be added OUTSIDE the entire width.
- **html** (*0 or 1*) – If set to 1, all spaces will be replaced with ' ', and the entire output will be wrapped with '<code>' and '</code>'.
- **br** (*string*) – Determine the carriage return. If html, it is suggested to set br to '
'. If you want the html source code easy to read, set br to ‘‘‘
- '``. –

References

version: 04b52 author : Runsun Pan require: odict() # an ordered dict, if you want the keys sorted.

Dave Benjamin http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/161403

Installation

At the command line via pip:

```
$ pip install kpfm
```

Or install the latest GitHub version using:

```
$ git clone https://github.com/marohngroup/kpfm.git
$ cd kpfm
$ python setup.py install
```

History

0.1.0 (2017-03-28)

- First release on GitHub.

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

Types of Contributions

Report Bugs

Report bugs at https://github.com/marohngroup/kpfm/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

Write Documentation

kpfm could always use more documentation, whether as part of the official kpfm docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/marohngroup/kpfm/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

Get Started!

Ready to contribute? Here's how to set up *kpfm* for local development.

1. Fork the *kpfm* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/kpfm.git
```

3. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

Now you can make your changes locally.

4. When you're done making changes, check that your changes pass tests by running:

```
$ python setup.py test
```

5. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

6. Submit a pull request through the GitHub website.

Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7 and Python 3.5. Check https://travis-ci.org/marohngroup/kpfm/pull_requests to verify that all configurations pass.

Credits

Development Lead

- Ryan Dwyer <ryanpdwyer@gmail.com>

Contributors

None yet. Why not be the first?

Feedback

If you have any suggestions or questions about **kpfm** feel free to email me at ryanpdwyer@gmail.com.

If you encounter any errors or problems with **kpfm**, please let me know! Open an Issue at the GitHub `http://github.com/marohngroup/kpfm` main repository.

Python Module Index

k

Index