`lyse`: a data analysis system for process-as-you-go automated data analysis

Chris Billington

March 2, 2012

Contents

1 Introduction

lyse is a data analysis system which gets *your code* running on experimental data as it is acquired. It is fundamenally based around the ideas of experimental *shots* and analysis *routines*. A shot is one trial of an experiment, and a routine is a Python script, written by you, that does something with the measurement data from one or more shots.

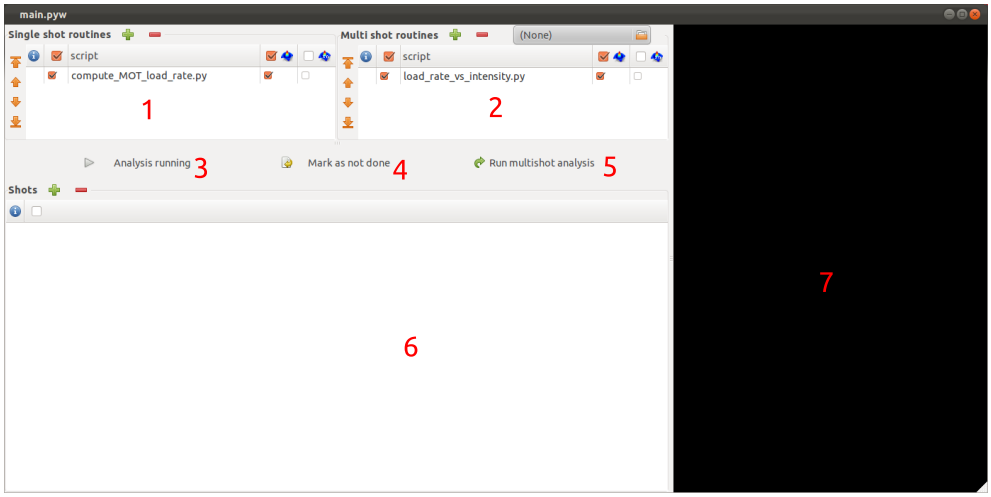
Analysis routines can be either *single-shot* or *multi-shot*. This determines what data and functions are available to your code when it runs. A single-shot routine has access to the data from only one shot, and functions available for saving results only to the hdf5 file for that shot. A a multi-shot routine has access to the entire dataset from all the runs that are currently loaded into lyse, and has functions available for saving results to an hdf5 file which does not belong to any of the shots—it's a file that exists only to save the 'meta results'.

Actually things are far less magical than that. The only enforced difference between a single shot routine and a multi-shot routine is a single variable provided to your code when lyse runs it. Your code runs in a perfectly clean Python environment with this one exception: a variable in the global namespace called path, which is a path to an hdf5 file. If you have told lyse that your routine is a singleshot one, then this path will point to the hdf5 file for the current shot being analysed. On the other hand, if you've told lyse that your routine is a multishot one, then it will be the path to an h5 file that has been selected in lyse for saving results to.

The other differences listed above are conventions only[1], and pertain to how you use the API that lyse provides, which will be different depending on what sort of analysis you're doing.

Here's a screenshot of lyse:

[1] Though lyse's design is based around the assumption that you'll follow these conventions most of the time



1. Here's where single shot routines can be added and removed, with the plus and minus buttons. They will be executed in order on each shot (more on how that works shortly). They can be reordered, or enabled/disabled with the checkboxes on the left. The checkboxes to the right, underneath the plot icons don't currently do anything, but they are intended to provide control over how plots generated by the analysis routines are displayed and updated.

2. Here is where multi-shot routines can be added or removed. The file selection button at the top allows you to select what hdf5 file multi-shot routines will get given (to which they will save their results).

3. Allows pausing of analysis. `lyse` by default will run all single-shot routines on a shot when it arrives (either via the HTTP server or having been manually added). After all the shots have been processed, only then will the multi-shot routines be executed. So if you load ten shots in quickly, the multi-shot routines won't run until they've all been processed by the single-shot routines. However most of the time there will be sufficient delay in between shots arriving that multi-shot routines will be executed pretty much every time a new shot arrives.

4. If you want to re-run single-shot analyses on some shots, select them and click this button. They'll then be processed in order.

5. This will rerun all the multi-shot analyses.

6. Here is where shots appear, either having arrived over HTTP of having been added manually via the file browser (by clicking the plus button). Many columns will populate this part of the screen, one for each global and each of the results (as saved by single-shot routines) present in the shots. A high-priority planned feature is to be able to choose exactly which globals and results are displayed. Otherwise this display is overwhelming to the point of uselessness. The data displayed here represents the entirety of what is available to multi-shot routines via the API provided by `lyse`.

7. This is where the output of routines is displayed, errors in red. If you're putting `print` statements in your analysis code, here is where to look to see them. Likewise if there's an exception and analysis stops, look here to see why.

2 The `lyse` API

SO GREAT, you've got a single filepath. What data analysis could you possibly do with that? It might seem like you have to still do the same amount of work that you would without an analysis system! Whilst that's not quite true, it's intentionally been designed that way so that you can run your code outside `lyse` with very little modification. Another motivating factor is to minimise the amount of magic black box behaviour, such that an analysis routine is actually just an ordinary `Python` script which makes use of an API designed for our purposes. `lyse` is both a program which executes your code, and an API that your code can call on.

To get started, you'll want to begin your analysis routine with:

```
1 from lyse import *
```

The `lyse` module[2] provides the following one function and two classes:

`data(filepath=None, host='localhost')` . The `data` function when called with no arguments obtains the current dataset from a running instance of `lyse` on the same computer. It returns a `pandas DataFrame` with the same rows and columns as you see in the main program of `lyse`. This is a simple way to get at your data, that doesn't require at all that your code is being run from within `lyse`. You can simply open a python interactive session, type `from lyse import *; df = data()`, and begin pulling out columns and plotting them against each other. Callin `data()` this way is intended for pulling data for multi-shot analysis, and should be avoided in single-shot mode.

When called with the `host` argument, the `data` function instead connects to a running instance of `lyse` on that computer, downloading its `DataFrame` over the network. I'm planning on including automatic `SSH` tunnelling through `bec.physics` to allow for us to obtain our data from outside the lab subnet without the need for a VPN.

When called with the `filepath` argument, the `data` function instead returns a `pandas Series` object with the globals and results from just the h5 file specified. This is intended for use in single-shot mode, with the filepath being that single global variable that `lyse` implants into the namespace, as I mentioned in sec 1.

[2] importing lyse imports the functions in `pythonlib/lyse/__init__.py`, whereas the main program is `pythonlib/lyse/main.pyw`

`Run(h5_path)` [3] Sometimes you need more than just the globals and results in single shot mode. In fact, you cannot produce any results without having access to measurement data—that is traces and images. Run objects provide methods for obtaining this data from an h5 file. They also provide methods for saving your results back to the same h5 file.

`t, V = Run.get_trace(name)` Returns an array of times and an array of voltages for an analogue input trace named `name`, as specified in a call to `AnalogIn.acquire` in `labscript`.

`im = Run.get_image(orientation,label,image)` Returns an image (as an array) from the camera with specified orientation (eg *side*, *top*), image label (eg *fluorescence*, *absorption*), and specific image name (eg *OD*, *atoms*, *flat*).

`Run.save_result(name,value)` Saves a single-value result to the hdf5 file. The result will be saved as an attribute to the group `/results/ your_script's_filename`, with the attribute `name` . Results saved in this way will be available to subsequent routines in the `DataFrame`s and `Series` returned by `data()` under the hierarchy `dataframe[your_script's_ filename, your_result's_name]`.

`Run.save_result_array(name,data)` This method saves an array which can be any numpy datatype convertible to hdf5 datatypes (which is pretty much any numpy array, including numpy 'record' arrays—those are the ones with named columns). The array will be saved in a dataset under the group `/results/ your_script's_filename`, with the dataset's name being `name`. It will not be accessible alongside globals and single-value results, but can be accessed with the `get_result_array` method.

`arr = Run.get_result_array(group,name)` This returns a numpy array as saved by the `save_result_array` function. The `group` argument specifies the name of the group that the result array was saved to within the results group of the hdf5 file. This will be then filename of the analysis routine which saved the result.

`Run.set_group(groupname)` When running `Python` in interactive mode, the `Run` object can't know what filename to use as the hdf5 group name to which results are saved with `save_result` and `save_result_array`. So if you try to instantiate a `Run` object in interactive mode, you'll be prompted to call this method to set what the group name should be instead.

`t1, V1,...tn, Vn = Run.get_trace(name_1,..., name_n)`
A convenience method for getting many traces at once.

`Run.save_results(name_1,value_1..., name_n, value_n)`
A convenience method for saving many results at once.

`arr1,...arrn = Run.get_result_arrays(group, name_1,..., name_n)`
A convenience method for getting many result arrays at once, provided they are within the same group.

[3] There is another argument to this function—`no_write=False`, but is is intended for use only internally by `Sequence`, which instantiates many runs but disables their functions for writing to file

`Run.save_result_arrays(name_1,data_1..., name_n, data_n)`
A convenience method for saving many result arrays at once.

`Sequence(h5_path,run_paths)` A `Sequence` object represents many runs. It provides methods for getting data from the runs, and for saving the results of multi-shot analyses to the file specified by `h5_path`, which should be the filepath that `lyse` provides to your multi-shot analysis script. `run_paths` should be a list of filepaths that you would like to be included in this `Sequence`. You can pull out these filenames from the `DataFrame` provided by the `data()` function with `df['filepaths']`. You might use this to pass in the filepaths for only a subset of the shots. You can also pass in the entire `DataFrame` as the `run_paths`, and if it contains a column called `'filepaths'`, then those filepaths will be used.

`Sequence.runs` The sequence object contains a `Run` object for each of the files in `run_paths`. `Sequence.runs` is a dictionary of these `Run` objects, keyed by filepath. This dictionary is mainly for internal use by the `Sequence` object, but is included here in case you really do need to delve into the data from individual shots during a multi-shot routine. All methods of these `Run` objects that would write to their hdf5 file have been disabled.

`Sequence.get_result_array` Takes the same arguments as `Run.get_result_array`, and returns a dictionary of result arrays, one for each run, keyed by filepath.

`Sequence.get_trace` Takes the same arguments as `Run.get_result_array`, and returns a dictionary of t, V tuples, one for each run, keyed by filepath.

The `Sequence` object also has the methods `save_result`, `save_result_array`, `save_results` and `save_result_arrays`, which work identically to equivalent methods in the `Run` object[4], the only difference being that you're saving the results to the h5 file associated with the `Sequence` object, rather than a file associated with a single shot.

3 Examples

3.1 Single-shot example

```
1 from lyse import *
2 from pylab import *
3
4 # Let's obtain our data for this shot -- globals, image attributes and
5 # the results of any previously run single-shot routines:
6 ser = data(path)
7
8 # Get a global called x:
9 x = ser['x']
```

[4] `Sequence` is actually a subclass of `Run`

```

# Get a result saved by another single-shot analysis routine which has
# already run. The result is called 'y', and the routine was called
# 'some_routine':
y = ser['some_routine','y']

# Image attributes are also stored in this series:
w_x2 = ser['side','absorption','OD','Gaussian_XW']

# If we want actual measurement data, we'll have to instantiate a Run object:
run = Run(path)

# Obtaining a trace:
t, mot_fluorecence = run.get_trace('mot fluorecence')

# Now we might do some analysis on this data. Say we've written a
# linear fit function (or we're calling some other libaries linear
# fit function):
m, c = linear_fit(t, mot_fluorecence)

# We might wish to plot the fit on the trace to show whether the fit is any good:

plot(t,mot_fluorecence,label='data')
plot(t,m*t + x,label='linear fit')
xlabel('time')
ylabel('MOT flourescence')
legend()

# Don't call show() ! lyse will introspect what figures have been made
# and display them once this script has finished running.  If you call
# show() it won't find anything. lyse keeps track of figures so that new
# figures replace old ones, rather than you getting new window popping
# up every time your script runs.

# We might wish to save this result so that we can compare it across
# shots in a multishot analysis:
run.save_result('mot loadrate', c)
```

3.2 Multi-shot example

```
from lyse import *
from pylab import *

# Let's obtain the dataframe for all of lyse's currently loaded shots:
df = data()

# Now let's see how the MOT load rate varies with, say a global called
# 'detuning', which might be the detuning of the MOT beams:

detunings = df['detuning']

```

```
# mot load rate was saved by a routine called calculate_load_rate:

load_rates = df['calculate_load_rate', 'mot loadrate']

# Let's plot them against each other:

plot(detunings, load_rates,'bo',label='data')

# Maybe we expect a linear relationship over the range we've got:
m, c = linear_fit(detunings, load_rates)
# (note, not a function provided by lyse, though I'm sure we'll have
# lots of stock functions like this available for import!)

plot(detunings, m*detunings + c, 'ro', label='linear fit')
legend()

#To save this result to the output hdf5 file, we have to instantiate a
#Sequence object:
seq = Sequence(path, df)
seq.save_result('detuning_loadrate_slope',c)
```