Developer Notes

These notes are intended to give a comprehensive breakdown of the Datalogger structure.

TODO: Maybe move this to Sphinx / ReadTheDocs as an `API Reference` page? eg. PyQtGraph

Contents

1. Infrastructure

1.1 GUI

PyQt5:

PyQt5 is used as the main engine for the GUI. Each item to display should be created as its own widget.

See the PyQt5 Reference Guide and the Qt5 Reference Pages for more.

PyQtGraph:

PyQtGraph is used for all graph plotting. Use the built-in `PlotWidget` for creating plots.

See the PyQtGraph Documentation.

1.2 Computation & calculation

Numpy:

Numpy is used as the core backend for all of the computation.

See the NumPy Reference.

SciPy:

Functions to perform common tasks (eg. signal processing, curve fitting) are often found in the SciPy library, and are much easier to use than creating your own.

See the SciPy Reference.

1.3 Code style and formatting

Please adhere to the Google Python Style Guide as closely as possible.

2. Datalogger: Universal components

2.1 Workspaces

Workspaces provide a way for the user to set up, save, and load customised configurations of the DataLogger. In this way, specific workspaces can be created (eg. for undergraduate teaching) to limit the functionality available.

The `.wsp` format

Workspaces are saved in a unique format, `.wsp` . WSP files are effectively a list of the settings for the DataLogger, allowing the user to enable add ons, set display options and suchlike. An example of a `.wsp` file can be found in `tests/test_workspace.wsp` .

Rules for a `.wsp` file:

- Only settings defined in the `Workspace` class are permitted (see below)
- Settings that are strings (eg. workspace names, paths) must use single quotes `''`
- Either boolean (`False / True`) or integer (`0 / 1`) values may be used for flags. It is recommended to use integers, for clarity
- The only form of line that will be interpreted as a setting is `variable_name=variable_value` where `variable_value` can either be a string (`variable_name='example'`), integer `variable_name=1` , or boolean (`variable_name=False`)
- Hence comments may be inserted into the `.wsp` file. It is recommended to use Python comment syntax (`#` and `""" """`)

The `Workspace` class

The `Workspace` class stores the workspace attributes and has methods for saving, loading, configuring, and displaying the workspace settings. Normally, a `CurrentWorkspace` instance is initiated that will store the current settings and all the workspace functionality will be accessed through the `CurrentWorkspace` .

2.2 Data storage and interaction

ChannelSets, Channels and DataSets:

The Datalogger uses a three-tier structure for storing data, comprising of ChannelSets, Channels and DataSets. See `channel.py` for further information.

```
                                           ChannelSet
          _____________________________________|_________________________________...
         /                         |                          |
     Channel 0                  Channel 1                  Channel 2             ...
    /         \                /         \                /         \
MetaData    DataSets       MetaData    DataSets       MetaData    DataSets       ...
   |        | | | |           |        | | | |           |        | | | |
  ...         ...            ...         ...            ...         ...          ...
```

DataSets: These are the lowest structure, effectively a vector of values with a name (id_) and units.

Channels: Normally created from one stream of input data, Channels include the original DataSet, any derived DataSets (eg. frequency spectra, sonogram) and metadata about the channel. They also have methods for getting and setting the attributes of the DataSets.

ChannelSets: The main object to interface with, with methods for getting and setting channel and dataset attributes. Each ChannelSet will typically be derived from one set of results or one run of the experiment.

Variable names:

For consistency, only the following names are permitted for DataSets and metadata. (* indicates that the variable is auto-generated)

DataSets:

- `time_series` - The raw input time series data
- `time` * - Calculated from the sample rate and number of samples
- `frequency` * - Calculated from the sample rate and number of samples (Hz)
- `omega` * - Angular frequency (rad), calculated from the sample rate and number of samples
- `spectrum` - The complex spectrum given by the Fourier Transform
- `sonogram` - The complex sonogram array, with shape (number of FFTs, frequencies)
- `sonogram_frequency` * - The frequency bins (Hz) used in plotting the sonogram. Not implemented yet, but will be calculated from the sonogram parameters.
- `sonogram_omega` * - The frequency bins (rad) used in plotting the sonogram. Not implemented yet, but will be calculated from the sonogram parameters.

Metadata:

- `name` - A human-readable string identifying this channel
- `comments` - A string for any additional comments
- `tags` - A list of tags for quick selection and sorting
- `sample_rate` - The rate (in Hz) that the data was sampled at
- `calibration_factor` - Not implemented yet
- `transfer_function_type` - either `None` , `'displacement'` , `'velocity'` , or `'acceleration'` , indicating what

type of transfer function is stored

2.3 Graph interaction

Pyqtgraph has built-in mouse interactions. These interactions are described in pyqtgraph documentations. Several methods are implemented in `PlotItem` to modify the interaction behaviours, such as setting the panning limits, setting mouse mode, enabling auto-ranging, and more.

The interactions can be modified if so desired, especially the right-click context menu. This requires subclassing the `ViewBox` [source]. To modify the context menu, reimplement the 'menu' variable and 'raiseContextMenu()' method. After that, initialise a `PlotItem` with the custom ViewBox as its 'viewBox' keyword argument.

In addition to mouse interactions, pyqtgraph also included interactive data selection. The documentation includes a basic introduction.

For our purposes, the `LinearRegionItem` and `InfiniteLine` from pyqtgraph are used. In general, the item is initialised and added to a PlotItem, Then, its Signal (usually emitted when its position has changed) is connected to a method which does something to the data in the selected region.

Examples by pyqtgraph on graph interaction can be accessed by inputting the line on command line after installing pyqtgraph:

```
python -m pyqtgraph.examples
```

2.4 Import / export

Not implemented yet.

There will be options for importing and exporting from:

- `.mat` files from the old Datalogger
- CSV, Excel, spreadsheet, text files etc
- Any custom file types (eg. George Stoppani)
- `.wav` , `.mp3` and other audio filetypes.

2.5 History Tree

Not implemented yet.

There will be some exciting and ideally simple way of undoing operations, perhaps using a PhotoShop/GIMP-style history tree that stores what operations have been performed and offers the ability to revert back to a specific point.

3. Datalogger: Acquisition module

3.1 Recorder Class

Two modules are created to handle data acquisition from different types of hardware: `myRecorder` to handle data from soundcards, and `NIRecorder` to handle data from National Instruments. Both modules contain a class named **Recorder** derived from the abstract class **RecorderParent** from `RecorderParent` module which provides methods for storing data acquired.

The import convention of the module is **mR** for `myRecorder` and **NIR** for `NIRecorder` .

RecorderParent contains methods to initialise a circular buffer, initialise a recording array, and initialise a trigger.

Recorder store data acquired from its stream using methods of **RecorderParent** as part of its callback routine, such as writing to buffer, writing to recording array, or checking for trigger.

The **Recorder** is written in such a way that it could be used in a python console. Thus, it is not restricted to be used in the GUI only, but can be used as part of a script.

The **Recorder** class will only output ***raw data***. Any kind of data processing is done outside of the class.

Any new **Recorder** or equivalent class can be implemented by deriving from the **RecorderParent**, and re-implement the required functions.

3.2 Window Layout

The acquisition window consists entirely of one main **QSplitter**, in which more **QSplitters**(namely left, middle, and right) are nested within in. The main splitter is oriented horizontally, while the nested splitters are oriented vertically. The *Widgets* written separately as a class are then added to the nested **QSplitters**.

This produces a resizable 'panel' for each *Widget*. The good thing about this implementation is some *Widget* can be hidden away from the user through the code.

3.3 Window Function

The window acts as a central hub to connect the widgets together. This is where most slots and signals are connected, meaning most callback functions are implemented here. The window also holds the variables for plotting the processed data from the Recorder classes (i.e. time series data, DFT data, and channel levels). A window without any extra *Widget* added to it would consist only the data plots within the QSplitters, and would technically function.

The window contains important variables such as:

- **Recorder**
    - *playing* - indicate whether the stream is playing
    - *rec* - Holds the recorder object
- **Plot Data**
    - *timedata* - Time vector for plotting
    - *freqdata* - Frequency vector for plotting
- **Plot Colourmaps**
    - *plot_colourmap* - Colourmap for plotting different channel
    - *level_colourmap* - Colourmap for indicating channel levels
- **Channel Set**
    - *live_chanset* - ChannelSet for holding metadata
- **Time + FFT Plots**
    - *plotlines* - Holds the lines of time + FFT plots
    - *plot_xoffset* - Indicate the x offsets of each plot line
    - *plot_yoffset* - Indicate the y offsets of each plot line
    - *plot_colours* - Indicate the colours of each plot line
    - *def_colours* - Holds the default colours of each plot line
    - *sig_hold* - Indicate whether to 'freeze' a signal plot
- **Channel Level Plots**

  - *peakplots* - Holds all the peak plots of each channel
  - *peak_trace* - Indicate the value of each trace of the peak of each channel
  - *peak_decays* - Indicate the level of decay of each trace
  - *trace_counter* - Indicate counter of each trace before it decays
  - *trace_countlimit*- Maximum amount of count before the trace decays

3.4 Widgets

All *Widgets* except `PlotWidgets` from `pyqtgraph` are reimplemented as a class of its own. They are derived from the abstract class **BaseWidget** which re-implement function for styling and provide a template for self-written *Widgets*. The construction of the UI components is implemented under `initUI()` method. No callback functions are done here, unless the callback only affect components within the *Widget*.

- **ChanToggleUI** - Toggling the plot of the channels
- **ChanConfigUI** - Configure the plot of the channels + metadata
- **DevConfigUI** - Configure the recorder
- **StatusUI** - Display status and some buttons
- **RecUI** - Record data
- **AdvUI** - Advance channel toggling

4. Datalogger: Analysis module

The `analysis` module contains all of the tools used for processing data and modal analysis.

4.1 Module structure

Each tool in the `analysis` module is designed so that it could be run independently of the master analysis window, as is its own PyQt widget. This makes it easy to add new tools and slot them into the master window as new tabs.

4.2 Window layout

Not implemented yet.

The master analysis window has:

- A menubar (see 4.3)
- A widget containing the tools used for analysis on the left ('Toolbox')
- A TabWidget to display the results of the different tools in the middle for different type of analysis
- A set of always-accessible widgets for performing tasks common to all (or most) tools - eg. channel selection, on the right ('Global Toolbox')

Both the left and right widgets are custom tabwidget, written to be collapsible tabs. The idea of this is to hide away the tools to allow maximum view of the results in the middle, while having the tabs visible to the user, allowing them to open up the tools.

Read on 4.4 for the implementation

4.3 Menus

A simple menu is implemented by subclassing **QMenu** and adding **QActions** or another **QMenu** to the **QMenu**. Then, the subclassed **QMenu** can be added to the Main Window's *menubar*. In the menubar, there are the following menus:

- Project - *ProjectMenu* class

- Data - *DataMenu* Class

- View - *ViewMenu* Class

Project menu

Not implemented yet.

This menu contains project-wide options, such as setting project preferences and saving the whole project.

Data menu

Not implemented yet.

This menu contains options for importing and exporting data.

4.4 Collapsible Tabs

The collapsible left and right widgets mentioned in 4.2 is a custom class **CollapsingSideTabWidget** that subclasses **QSplitter**.

The widget is then mainly made up of two widgets: **QStackedWidget** and **QTabBar**. **QStackedWidget** is to contain all the analysis tools but showing one tool at a time, while **QTabBar** shows all the tools available and switch the respective tool when clicked on.

Animation is implemented to take advantage of the **QSplitter** resizing capabilities, which involves a third empty widget.

4.5 Tools

Currently, the idea is each type of analysis has its own set of tools, *e.g. time series analysis and frequency series analysis has distinct set of tools*. Thus, when switching tabs for different analysis in the middle widget, the left tool widget will switch to the relevant tools.

An *abstract* class **BaseTools** is written to streamline the process of addi ng and display tools in the collapsible tabs. This merely contains the tools and the label to the respective analysis.

To properly utilise this class, it must be derived and redefine the `initTools` method. Under here, a tool can be written as a **QWidget**, and then added using the method `add_tool`, supplying a name for that tool.

A variable named *parent* is provided to allow any signal and slots to be made to the main window.

Four **Tools** are created: ***TimeTools****: Tools for time series analysis* **FreqTools**: Tools for frequency series analysis ***ModalTools****: Tools for modal analysis* **GlobalTools**: Tools that is common for all analysis, this is added to the right widget mentioned before

A new **Tools** class can be derived if the tool does not belong to any of the above categories, but that new class must be added to the `prepare_tools` method in the Main Window, and check that it opens for the correct analysis.

5. Addons

Addons (extra extension scripts) may be written to extend the functionality of the DataLogger.

5.1 Addon structure

File structure

See `datalogger/addons/example_addon.py` and `datalogger/addons/addon_template.py` for examples of addons.

Addons must all be structured according to the `addon_template.py`. That is:

```
#datalogger_addon

 #--------------------------------------------------------------------------------
 # Put metadata about this addon here
 #--------------------------------------------------------------------------------
 addon_metadata = {
   "name": "<name>",
   "author": "<author>",
   "description": "<description>",
   "category": "<category>"}

 #--------------------------------------------------------------------------------
 # Master run function - put your code in this function
 #--------------------------------------------------------------------------------
 def run(parent_window):
  #--------------------------------------------------------------------------------
  # Your addon functions
  #--------------------------------------------------------------------------------
  <any user defined functions>
  #----------------------------------------------------------------------------
  # Your addon code:
  #----------------------------------------------------------------------------
  <code goes here>
```

Header (`#datalogger_addon`): This informs the datalogger that this is an addon file.

Metadata (`addon_metadata`): Contains information about the addon. Displayed in the Addon Manager. Addons are sorted according to their `"category"` .

Main code (`run`): The actual addon code is all kept under the `run` function. This is the function that is called when the addon is run. Only variables, functions, classes etc defined within the `run` function will be accessible by the addon, so don't put any code outside of `run` .

What addons can do

In an addon, it is possible to:

- Import modules
- Define functions, classes and variables
- Access widgets, attributes, and methods of the `parent_window` (eg. to plot data in the Analysis Window, or to do calculations with the current data)
- Display popups and Qt dialog boxes

And probably a lot of other things as well.

5.2 Addon Manager

The Addon Manager provides a widget for loading, selecting, and running addons. It also has a console for displaying the addon output.

The Addon Manager is included in the Global Toolbox of the Analysis Window.

Creating addons

Not implemented yet. It would be nice to have some way of creating addons from within the manager.

Loading addons

Addons located in `CurrentWorkspace.path/addons/` will be automatically discovered and added to the AddonManager. Other addons may be loaded from within the addon manager, using the `Load Addon` button, which opens up a file explorer. Multiple addons can be added simultaneously in this way.

Running addons

Addons are run using the `Run Selected` button in the addon manager, or by double clicking them in the tree. When the addon is run, a new `QThread` is created that routes `stdout` to the `QTextEdit` console widget in the manager. This means that anything the addon prints to `stdout` will be displayed in the console, so `print` calls can be used in the addon as normal to display results in the console widget.

Note that the thread will route everything that goes to `stdout` to the console widget, including things printed from the main window.