CUED DataLogger Documentation

Release 0

Theo Brown, En Yi Tee

Sep 15, 2017

Contents:

CHAPTER 1

Introduction

This documentation, in its current form, is simply an API Reference for the CUED DataLogger developers.

Hopefully at some point it will develop to include a full manual and other such exciting things.

See the *Quick start guide* for how to get the DataLogger up and running, and *Using the DataLogger* for more detailed information on how to use the DataLogger.

CHAPTER 2

Quick start guide

Install the DataLogger by following the relevant instructions below.

Then run the DataLogger from a command line using:

```
cued_datalogger
```

To specify a Workspace when running the DataLogger, use:

```
cued_datalogger -w /path/to/workspace.wsp
```

For further options type:

```
cued_datalogger --help
```

For a debugging version type:

```
cued_datalogger_dbg
```

See the documentation for more information.

Installation

Installing on Windows

1. Download and install Anaconda / Miniconda.
2. Check that your Anaconda is using the latest version of `pip`. In an Anaconda Prompt, type:

   ```
   conda install pip
   ```

3. Install `cued_datalogger` using `pip`:

```
pip install cued_datalogger
```

Installing on OS X

1. Install `portaudio` with `brew` *:

```
brew install portaudio
```

2. Install `cued_datalogger` using `pip` (from Terminal or from Anaconda Prompt or wherever):

```
pip install cued_datalogger
```

* If you do not have brew installed, install Homebrew then permit it to run with `xcode-select --install`

Installing on Linux

1. Install the `portaudio` development headers using your package manager.

   Debian / Ubuntu:

```
sudo apt install libportaudio2 portaudio19-dev
```

   CentOS:

```
yum install portaudio portaudio-devel
```

2. Install `cued_datalogger` using `pip`:

```
pip install cued_datalogger
```

CHAPTER 3

Using the DataLogger

The DataLogger is designed for the following directory structure:

```
..
name_of_lab/

    addons/
        lab_addon1.py
        lab_addon2.py

    lab_workspace_file.wsp
```

In normal use, you would navigate to the `name_of_lab/` folder and then run:

```
cued_datalogger -w lab_workspace_file.wsp
```

This launches the DataLogger and loads the `lab_workspace_file`. The DataLogger will then automatically find and include all the addons found in the `addons/` folder.

It may be useful to read the documentation on *Workspaces* and *Data Storage* to familiarise yourself with how the DataLogger works.

CHAPTER 4

Information For Developers

This section contains information for people who are involved in continuing the development of the DataLogger.

Dependencies

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. However, in general plots should be created using the DataLogger's *InteractivePlotWidget*, which provides some additional functionality.

See the PyQtGraph Documentation.

Matplotlib

Some parts of the DataLogger use Matplotlib for displaying or exporting additional plots. It should be used as a last resort only when finer control is needed over how the data is displayed (for example in contour maps), as Matplotlib is much slower than PyQtGraph, less well integrated into PyQt, and does not fit with the styling of the DataLogger.

See the Matplotlib Documentation.

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.

Code style and formatting

Please adhere to the Google Python Style Guide as closely as possible, with the following exception:

Docstrings must follow the Numpy Style Guide, as the docstrings are parsed using `numpydoc` to produce the documentation.

Package management

The `cued_datalogger` package is installable from PyPI (the Python Package Index) via `pip` (see quickstart for more information).

This section of documentation attempts to describe how the package was set up.

Compiling the package

Python provides a package for creating packages, `setuptools`. The `setup.py` script uses `setuptools` to compile the code into a Python package.

To compile the package and upload the new version to PyPI, run:

```
python setup.py sdist upload
```

This runs the setup script to create a source code distribution (tarball) and uploads the distribution to PyPI.

> **Warning:** Do not attempt to create a Python wheel for the package. There are some issues with using the `install_requires` parameter from `setuptools`. `install_requires` installs dependencies using the PyPI source distribution. For some packages (PyQt5) there is no source distribution available. To get round this, the current `setup.py` script installs Python wheels (binaries) manually for all the dependencies. As there are no packages in `install_requires`, compiling a binary wheel from the setup script will not result in an distribution with the necessary dependencies.

Installing a local developer's version

If you have downloaded the Git repository and made changes to files, you need to locally install your changed version so that all of the module imports work correctly.

Navigate to the Git repository and run `pip install -e .` to get a developer's version of the package installed locally.

Package structure

```
cued_datalogger/ (repository)

    cued_datalogger/ (package)

         acquisition/ (subpackage)
             Contains all modules unique to data acquisition and the AcquisitionWindow

         analysis/ (subpackage)
             Contains all modules unique to data analysis and the AnalysisWindow

         api/ (subpackage)
             Contains all modules that provide the general functionality of the␣
↪DataLogger

         __main__.py (module)
             Functions for running the DataLogger

    docs/
        Contains source code for documentation

    lib/
        Contains additional libraries installed during setup)

    tests/

    setup.py (installation script)
```

Documentation

The documentation for the DataLogger is stored in the `docs` directory in the git repository. It is built using Sphinx. For tutorials on writing documentation using Sphinx, see here and here.

Writing documentation

For the majority of the documentation, use Sphinx's autodoc functionality.

Compiling documentation

Local version

To create a local version of the documentation (eg. for checking that the documentation compiles) navigate to the top-level `docs/` directory and run:

```
make html
```

The built version should appear in `docs/build/html`.

ReadTheDocs version

ReadTheDocs is a documentation hosting website. The DataLogger documentation can be found at cued-datalogger.readthedocs.io.

The ReadTheDocs project is currently set to track the `docs` branch of the Git repository.

To build a new version of the documentation:

1. Navigate to the ReadTheDocs project homepage.
2. Under **Build a version**, click *Build*. You can check the progress of the build in the *Builds* tab.
3. Click *View Docs* to view the documentation. If lots of the documentation is missing, the `autodoc` directives have probably failed, suggesting that the build did not successfully install the `cued_datalogger` module. Check the *Builds* tab in the project homepage.

CHAPTER 5

API Reference

Infrastructure

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 `"""` `"""`)

Running the DataLogger with a Workspace

To specify a Workspace when running the DataLogger, use:

```
cued_datalogger -w /path/to/workspace.wsp
```

The `Workspace` class

class `cued_datalogger.api.workspace.`**`Workspace`**

Bases: `object`

The `Workspace` class stores the workspace attributes and has methods for saving, loading, configuring, and displaying the workspace settings.

Workspaces are designed so that specific configurations of the DataLogger can be created, eg. for undergraduate labs, with different features enabled or disabled, and stored in a `.wsp` file that can be read using the `Workspace` class. In the DataLogger, a `CurrentWorkspace` instance is normally initiated that will store the current settings and all the workspace functionality will be accessed through the `CurrentWorkspace`.

Attributes

name	(str) A human-readable name for this workspace, eg. `"Lab 4C6"`
path	(str) The path to this workspace's directory. Addons will be loaded from the directory `path/addons/`, and files will be saved to `path` (not implemented yet). Default value is `"./"`.
add_ons_enabled	(bool) Flag that sets whether to addons are enabled (not implemented yet - currently has no effect). Default value is `True`
pyqtgraph_inverted	(bool) Flag that sets whether pyqtgraph uses a white background and black lines (`False`), or black background and white lines (`True`). Default value is `False`.
default_pen	(str) The default colour of the pen, set by `pyqtgraph_inverted`. Cannot be set manually.
pyqtgraph_antialias	(bool) Flag that sets whether pyqtgraph uses antialiasing for smoother lines. Default value is `True`.

Methods

`configure``()`
Set the global configuration of the DataLogger to the settings in this workspace.

`load``(`*workspace*`)`
Load the settings found in the .wsp file given by *workspace*‘ (of the form `"/path/to/workspace.wsp"`).

`save``(`*destination*`)`
Save this workspace to *destination* (of the form `"/path/to/workspace.wsp"`).

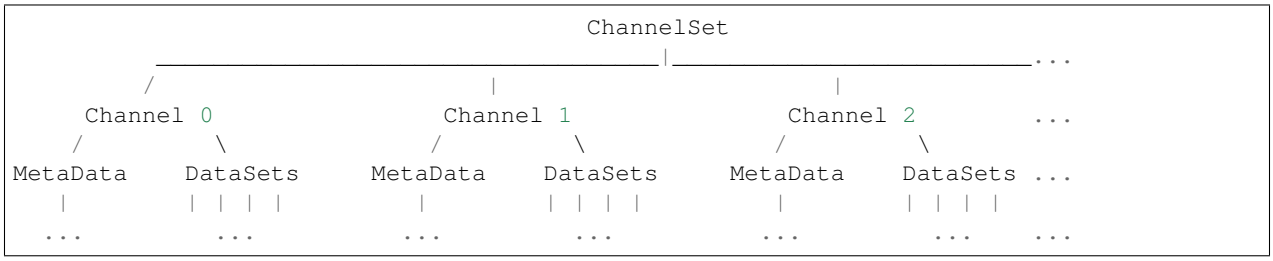
`settings``()`
A convenience method to access this workspace's configuration

Widgets

Not implemented yet.

Data Storage

The Datalogger uses a three-tier structure for storing data, comprising of ChannelSets, Channels and 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.

ChannelSet

class `cued_datalogger.api.channel.`**`ChannelSet`**(*initial_num_channels=0*)

Bases: `object`

A group of channels, with methods for setting and getting data.

In theory, a user will only need to interact with the ChannelSet to interact with the channels and data. Each ChannelSet will normally be derived from one set of results, or one run of an experiment, and then the ChannelSet will contain all the information and analysis from that run. ChannelSets can be initialised as empty, and channels added later, or initialised with a number of empty channels, to which DataSets can be added later. Channels are stored in a matlab-style list structure (see *`MatlabList`*) which uses tuple indexing, eg. `channelset.channels[1, 2, range(5,10)]`, so that multiple channels can be selected easily.

Attributes

channels	(MatlabList) A list of the channels in this set.
colormap	(ColorMap) A `ColorMap` used for colouring the channels in this set.

Methods

`__init__`(*initial_num_channels=0*)

Create the ChannelSet with a number of blank channels as given by *initial_num_channels*.

`__len__`()

Return the number of Channels in this ChannelSet.

`add_channel_dataset`(*channel_index*, *id_*, *data=None*, *units=None*)

Add a DataSet with *id_* to the Channel specified by *channel_index*. DataSet can be initialised as empty (default) or with *data* and/or *units*.

`add_channels`(*num_channels=1*)

Add a number (*num_channels*) of new empty Channels to the end of the channel list.

channel_colour(*channel_index*)
Get the RGBA tuple specifying the colour of the Channel at *channel_index*.

channel_data(*channel_index*, *id_*)
Return the data from the DataSet given by *id_* in the Channel specified by *channel_index*.

channel_ids(*channel_index*)
Return the ids of all the datasets in the Channel specified by *channel_index*.

channel_metadata(*channel_index*, *metadata_id=None*)
Return the metadata (either a specific item given by *metadata_id* or the full dict of metadata) of the Channel specified by *channel_index*.

channel_units(*channel_index*, *id_*)
Return the units from the DataSet given by *id_* in the Channel specified by *channel_index*.

info()
Print the information about all of the Channels in this ChannelSet.

set_channel_colour(*channel_index*)
Set the RGBA tuple specifying the colour of the Channel at *channel_index* to a value determined by its index and the ChannelSet `colormap`.

set_channel_data(*channel_index*, *id_*, *data*)
Set the data of DataSet with *id_* to *data* in the Channel specified by *channel_index*.

set_channel_metadata(*channel_index*, *metadata_dict*)
Set metadata of the Channel specified by *channel_index* using the keys and values given in *metadata_dict*.

set_channel_units(*channel_index*, *id_*, *units*)
Set the units of DataSet with *id_* to *units* in the Channel specified by *channel_index*.

update_channel_colours()
Update the `colormap` so that the channels are mapped to the full range of colours, and update all the channel colours.

Channel

class cued_datalogger.api.channel.**Channel**(*name=''*, *datasets=[]*, *comments=''*, *tags=[]*, *sample_rate=1000*, *calibration_factor=1*, *transfer_function_type='displacement'*, *colour=None*)

Bases: `object`

Contains a group of DataSets and associated metadata.

Channels are the basic structure used throughout the CUED DataLogger. Channels may contain many DataSets, but each must have a unique id_ (ie. cannot have two 'time' DataSets). Typically a Channel will be initialised with just 'time_series' data, and other DataSets will be added as analysis is performed - eg. a Fourier Transform produces a 'spectrum' DataSet. Channels also contain metadata about the data.

Attributes

name	(str) A human-readable string identifying this channel (eg. `'Input 0'`, or `'Left Accelerometer'`).
datasets	(list) A list of this Channel's DataSets.
comments	(str) A string for any additional comments.
tags	(list) A list of tags (eg. ['accelerometer', 'input']) for quick selection and sorting.
sample_rate	(float) The rate (in Hz) that the data was sampled at.
calibration_factor	(float) #TODO#
transfer_function_type	(str) Either 'None', 'displacement', 'velocity', or 'acceleration'- indicates what type of transfer function is stored.
colour	(tuple) An RGBA tuple for this channel's colour - usually set by its parent ChannelSet

Methods

__init__(*name=''*, *datasets=[]*, *comments=''*, *tags=[]*, *sample_rate=1000*, *calibration_factor=1*, *transfer_function_type='displacement'*, *colour=None*)
Create a new Channel. Can be initialised as empty, or with given metadata and/or with given DataSets.

add_dataset(*id_*, *units=None*, *data=[]*)
Create a new dataset in this channel with *id_*, *units*, *data*. If a dataset given by *id_* exists set its units and data.

data(*id_*)
Return the data from the DataSet given by *id_*.

dataset(*id_*)
Return the DataSet in this channel with *id_*.

ids()
Return a list of the DataSet ids that this channel has.

info()
Print this Channel's attributes, including DataSet ids and metadata.

is_dataset(*id_*)
Return a boolean of whether the dataset given by *id_* exists with data already.

metadata(*metadata_id=None*)
Return the value of this channel's metadata associated with *metadata_id*. If none given, returns all of this channel's metadata in a dictionary.

set_data(*id_*, *data*)
Set the data in dataset *id_* to *data*.

set_metadata(*metadata_dict*)
Set the channel metadata to the metadata given in *metadata_dict*.

set_units(*id_*, *units*)
Set the units of dataset *id_* to *units*.

units(*id_*)
Return the units from the DataSet given by *id_*.

update_autogenerated_datasets()
Regenerate the values in the automatically generated DataSets.

DataSet

class cued_datalogger.api.channel.**DataSet**(*id_*, *units=None*, *data=array([], dtype=float64)*)
Bases: object

A DataSet is the basic unit for data storage - a named 1d vector with units.

Notes

Permitted values for the DataSet id_ are:

- "time_series" - The raw input time series data
- "time"* - Calculated from the sample rate and number of samples (units 's')
- "frequency"* - Calculated from the sample rate and number of samples (units 'Hz')
- "omega"* - Angular frequency (units '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. Calculated from the sonogram parameters.
- "sonogram_omega"* - The frequency bins (rad) used in plotting the sonogram. Calculated from the sonogram parameters.
- "coherence"
- "transfer_function"

(* indicates that this DataSet is auto-generated by the Channel)

Attributes

id_	(str) A lower-case string containing the name of the data stored in the vector. See Notes for permitted values.
units	(str) The SI unit in which the data is measured.
data	(ndarray) A numpy array of data points associated with id_.

Methods

__init__(*id_*, *units=None*, *data=array([], dtype=float64)*)
Create a new DataSet with unique *id_*. Can either be initialised as empty, or with units and/or data.

set_data(*data*)
Set the DataSet's data array to *data*.

set_id(*id_*)
Set the DataSet's id_ to *id_*.

set_units(*units*)
Set the DataSet's units to *units*.

Widgets

See *ChannelSelectWidget* and *ChannelMetadataWidget* for widgets to interact with ChannelSets.

Plot interaction

A description.

class cued_datalogger.api.pyqtgraph_extensions.**InteractivePlotWidget**(*parent=None*, *show_region=True*, *show_crosshair=True*, *show_label=True*, **args*, ***kwargs*)

Bases: PyQt5.QtWidgets.QWidget

A QWidget containing a CustomPlotWidget with mouse tracking crosshairs, a LinearRegionItem, and spinboxes to display and control the values of the bounds of the linear region. Any additional arguments to **:method:‘__init__‘** are passed to the CustomPlotWidget.

Attributes

PlotWidget	(pg.PlotWidget) The PlotWidget contained in the InteractivePlotWidget.
ViewBox	(pg.ViewBox) The ViewBox contained in the InteractivePlotWidget.
region	(pg.LinearRegionItem) The LinearRegionItem contained in the InteractivePlotWidget.
vline	(pg.InfiniteLine) Vertical mouse-tracking line.
hline	(pg.InfiniteLine) Horizontal mouse-tracking line.
label	(pg.LabelItem) LabelItem displaying current mouse position.
lower_box	(QSpinBox) QSpinBox displaying lower bound of region.
upper_box	(QSpinBox) QSpinBox displaying upper bound of region.
zoom_btn	(QPushButton) Press to zoom to the region with a set amount of padding.
show_region	(bool) Controls whether the region is displayed.
show_crosshair	(bool) Controls whether the crosshair is displayed.
sig_region_changed	(pyqtSignal([int, int])) The signal emitted when the region is changed.

Methods

clear()
Clear the PlotWidget and add the default items back in.

getRegionBounds()
Return the lower and upper bounds of the region.

plot(*x=None*, *y=None*, **args*, ***kwargs*)
update_limits() from the x and y values, then plot the data on the plotWidget.

update_limits(*x*, *y*)
Set the increment of the spinboxes, the limits of zooming and scrolling the PlotItem, and move the region to x=0

zoomToRegion(*padding=0.1*)
Zoom to the region, with given padding.

class cued_datalogger.api.pyqtgraph_extensions.**CustomPlotWidget**(**args*, *show_region=True*, *show_crosshair=True*, *show_label=True*, ***kwargs*)

Bases: pyqtgraph.widgets.PlotWidget.PlotWidget

Attributes

lastFileDir	

Methods

autoRange_override(*padding=None*, *items=None*)
Autorange the view to fit the plotted data, ignoring the location of the crosshair. Accessed as autoRange(), not as autoRange_override().

clear_override(**args*, ***kwargs*)
Clear the PlotItem and add the default items back in. Accessed as clear(), not as clear_override().

mouseMoved(*mouse_moved_event*)
Update the crosshair and label to match the mouse position.

plot_override(**args*, ***kwargs*)
Plot data on the widget and autoRange. Accessed as plot(), not as plot_override().

set_show_crosshair(*show_crosshair*)
Set whether the crosshair is visible.

set_show_label(*show_label*)
Set whether the label is visible.

set_show_region(*show_region*)
Set whether the region is visible.

Import & Export

Importing

cued_datalogger.api.file_import.**import_from_mat**(*file*, *channel_set=None*)
A function for importing data and metadata to a ChannelSet from an old-style DataLogger .mat file.

Parameters **file** : path_to_file

The path to the .mat file to import data from.

channel_set : ChannelSet

The ChannelSet to save the imported data and metadata to. If None, a new ChannelSet is created and returned.

Exporting

Not implemented yet.

Widgets

See *DataImportWidget*.

Acquisition

Description of the window for acquiring data for analysis.

Recorder Parent

This module contains the abstract class to implement a proper Recorder Class. To do so, subclass RecorderParent when creating a new Recorder class.

Example: from RecorderParent import RecorderParent

class newRecorder(RecorderParent):

If you have PyQt, it will import RecEmitter for emitting Signals.

Attributes

QT_EMITTER : Indicates whether you can use qt Signals

class cued_datalogger.acquisition.RecorderParent.**RecorderParent**(*channels=1*, *rate=44100*, *chunk_size=1024*, *num_chunk=4*)

Bases: object

Recorder abstract class. Sets up the buffer and skeleton for audio streaming

Attributes

channels: int	Number of Channels
rate: int	Sampling rate
chunk_size: int	Number of samples to get from each channel in one chunk
num_chunk: int	Number of chunks to store in circular buffer
recording: bool	Indicate whether to record

Methods

allocate_buffer()
Set up the circular buffer

audiodata_to_array(*data*)
Convert audio data obtained into a proper array

Parameters data: Numpy Array

Audio data

available_devices()
Displays all available device for streaming

chunk_size

int Number of samples to get from each channel in one chunk The setter method will calculate the maximum possible size based on an arbitrary number of sample limit (2^25 in here)

close()
Close the audio object, to be called if streaming is no longer needed

current_device_info()
Displays information about available the current device set

flush_record_data()
Add in any partial posttrigger data Slice the recorded data into the requested amount of samples Add in any pretrigger data

Returns flushed_data: numpy array

2D numpy array (similar to get_buffer)

get_buffer()
Convert the buffer data as a 2D array by stitching the chunks together

Returns Buffer data: Numpy Array

with dimension of(chunk_size * num_chunk) x channels The newest data on the most right

num_chunk

int Number of chunks to store in circular buffer The setter method will calculate the maximum possible number of chunks based on an arbitrary number of sample limit (2^25 in here)

open_recorder()
Initialise the variables for recording.

record_cancel()
Cancel a recording and clear any recorder data

record_data(*data*)
Append recorded chunk to recorder_data and stop doing so if neccessary amount of chunks is recorded

record_init(*samples=None*, *duration=3*)
Remove any pretrigger and postrigger data Calculate the number of chunk to record It will record more samples than necessary, then slice down to the amount of samples required + putting in pretrigger data

Parameters **samples: int**

Number of samples to record

duration: int

The recording duration

record_start()
Start recording if it is possible

Returns bool

True if possible, False otherwise

set_device_by_name(*name*)
Set the device to be used for audio streaming

show_stream_settings()
Show the settings of the recorder

stream_close()
Callback function for closing the audio streaming.

stream_init(*playback=False*)
Callback function for initialising audio streaming.

Parameters **playback: bool**

Whether to output the stream to a device

Returns bool

True if successful, False otherwise

stream_start()
Callback function for starting the audio streaming.

stream_stop()
Callback function for stopping the audio streaming.

trigger_init()
Initialise the variable for the trigger recording

trigger_start(*duration=3*, *threshold=0.09*, *channel=0*, *pretrig=200*, *posttrig=5000*)
Start the trigger if possible

Returns bool

True if successful, False otherwise

write_buffer(*data*)
Write the data obtained into buffer and move to the next chunk

Parameters **data: Numpy Array**

Audio data

Pyaudio Recorder

This module contains the class to record data from a soundcard. It uses PyAudio to do so. Please check the PyAudio Documentation for more information.

Typical example of using the module:

>>>import myRecorder as mR
>>>recorder = mR.Recorder()
Channels: 1
Rate: 44100
Chunk size: 1024
Number of chunks: 4
You are using pyAudio for recording
Device not found, reverting to default
Selected device: Line (3- U24XL with SPDIF I/O)
>>>recorder.stream_init()
stream already started
Input latency: 2.322e-02
Output latency: 0.000e+00
Read Available: -9977
Write Available: -9976

True
>>>recorder.record_init()
Recording function is ready! Use record_start() to start
True
>>>recorder.record_start()
stream already started
Recording Start!
True
>>>Recording Done! Please flush the data with flush_record_data().
data = recorder.flush_record_data()
>>>recorder.close()

class cued_datalogger.acquisition.myRecorder.**Recorder**(*channels=1*, *rate=44100*, *chunk_size=1024*, *num_chunk=4*, *device_name=None*)

Bases: *cued_datalogger.acquisition.RecorderParent.RecorderParent*

Sets up the recording stream through a SoundCard

Attributes

device_index: int	Index of the device to be used for recording
device_name: str	Name of the device to be used for recording
max_value: float	Maximum value of recorded data

Methods

audiodata_to_array(*data*)
Re-implemented from RecorderParent

available_devices()
Searches for any available input devices

Returns names: List

Name of the devices

index: List

Index of the devices

close()
Re-implemented from RecorderParent, but terminate the PyAudio Object too.

current_device_info()
Display the current selected device info

open_recorder()
Re-implemented from RecorderParent. Prepare the PyAudio Object too.

set_device_by_name(*name*)
Set the recording audio device by name. Revert to default if no such device found

Parameters **name: str**

Name of the device

stream_audio_callback(*in_data*, *frame_count*, *time_info*, *status*)
Callback function for audio streaming. First, it writes data to the circular buffer, then record data if it is recording, finally check for any trigger.

Inputs and Outputs are part of the callback format. More info can be found in PyAudio documentation

stream_close()
Re-implemented from RecorderParent.

stream_init(*playback=False*)
Re-implemented from RecorderParent.

stream_start()
Re-implemented from RecorderParent.

stream_stop()
Re-implemented from RecorderParent.

National Instrument Recorder

This module contains the class to record data from a National Instrument. It uses PyDAQmx to do so, but requires NIDAQmx drivers to function. Please check the PyDAQMx and NIDAQmx C API reference for more information.

Typical example of using the module:

>>>import myRecorder as NIR
>>>recorder = NIR.Recorder()
Channels: 1
Rate: 30000
Chunk size: 1000
Number of chunks: 4
You are using National Instrument for recording
Input device name not found, using the first device
Selected devices: Dev3
>>>recorder.stream_init()
Channels Name: Dev3/ai0
True
>>>recorder.record_init()
Recording function is ready! Use record_start() to start
True
>>>recorder.record_start()
stream already started
Recording Start!
True
>>>Recording Done! Please flush the data with flush_record_data().
data = recorder.flush_record_data()
Data flushed
>>>recorder.close()

class cued_datalogger.acquisition.NIRecorder.**Recorder**(*channels=1*, *rate=30000.0*, *chunk_size=1000*, *num_chunk=4*, *device_name=None*)
Bases: *cued_datalogger.acquisition.RecorderParent.RecorderParent*

Sets up the recording stream through a National Instrument

Attributes

device_name: str	Name of the device to be used for recording
max_value: float	Maximum value of recorded data

Methods

audiodata_to_array(*data*)
Re-implemented from RecorderParent

available_devices()
Get all the available input National Instrument devices.

Returns devices_name: List of str

Name of the device, e.g. Dev0

device_type: List of str

Type of device, e.g. USB-6003

current_device_info()
Prints information about the current device set

set_channels()
Create the string to initiate the channels when assigning a Task

Returns channelname: str

The channel names to be used when assigning Task e.g. Dev0/ai0:Dev0/ai1

set_device_by_name(*name*)
Set the recording audio device by name. Uses the first device found if no such device found.

stream_audio_callback()
Callback function for audio streaming. First, it writes data to the circular buffer, then record data if it is recording, finally check for any trigger.

Returns 0 as part of the callback format. More info can be found in PyDAQmx documentation on Task class

stream_close()
Re-implemented from RecorderParent.

stream_init(*playback=False*)
Re-implemented from RecorderParent.

stream_start()
Re-implemented from RecorderParent.

stream_stop()
Re-implemented from RecorderParent.

Acquisition Window

The program was inspired by a program known as livefft, written in pyqt4, by Dr Rick Lupton (CUED). The livefft program is under the MIT license.

Window Layout

To be consistent with the analysis window layout, the acquisition window adopts a similar style of layout to the analysis window. On the left contains the tools to toggle plots, to configure plots, and to configure recording device. In the middle, there are the plots of the stream in time domain and frequency domain, with a status at the bottom. On the right contains the recording settings and the plot of the channel levels.

Acquisition Widgets

Created on Tue Aug 22 11:19:29 2017 @author: eyt21

This module contains the widget classes to the acquisition window. However, they are not limited to that window, and can be reused for other window, like the analysis window.

Attributes

NI_DRIVERS: bool Indicates whether NIDAQmx drivers and pyDAQmx module are installed when attempting to import NIRecorder module The module is needed to check on the available National Instrument devices

MAX_SAMPLE: int Arbritrary maximum number of samples that can be recorded.

class `cued_datalogger.acquisition.RecordingUIs.`**`BaseWidget`**`(`*arg*, ***kwarg*`)`
Bases: `PyQt5.QtWidgets.QWidget`

A base widget reimplemented to allow custom styling. Pretty much identical to a normal QWidget

Methods

`initUI``()`
Construct the UI, to be reimplemented.

`paintEvent``(`*evt*`)`
Reimplemented from QWidget.paintEvent()

class cued_datalogger.acquisition.RecordingUIs.**ChanToggleUI**(**arg*, ***kwarg*)
Bases: *cued_datalogger.acquisition.RecordingUIs.BaseWidget*

A Channel Toggling widget. Contains:

•Checkboxes to toggle channel,

•Buttons to select all, deselect all, and invert selection.

•LineEdits to toggle by expression or tags

Attributes

toggleChanged: pyqtsignal	Emits when a channel toggle changes, Sends out the channel num(int) and the state(bool)
channels_box: QWidget	The widget containing the checkboxes
checkbox_layout: QGridLayout	Defines the layout of the checkboxes
chan_btn_group: QButtonGroup	Widget to handle the checkboxes presses
sel_all_btn:QPushButton	'Select All' button
desel_all_btn: QPushButton	'Deselect All' button
inv_sel_btn: QPushButton	'Invert Selection' button
chan_text: ChanLineText	For toggling by expression
chan_text2: QLineEdit	For toggling by tags
chan_text3: QLineEdit	For displaying the channels toggled (may be changed to QLabel instead)
search_status: QStatusBar	For displaying whether the toggling is successful

Methods

adjust_channel_checkboxes(*new_n_btns*)
Add or delete checkboxes based on new number of buttons needed

Parameters **new_n_btns: int**

New number of buttons required

chan_line_toggle(*chan_list*)
Callback to intepret the input expressions and toggle the channels accordingly

Parameters **chan_list: List of str**

Input expressions

initUI()
Reimplemented from BaseWidget.

invert_checkboxes()
Callback to invert selection

toggle_all_checkboxes(*state*)
Callback to select all or deselect all

Parameters **state: int**

State of the checkboxes to be in (either Qt.Unchecked or Qt.Checked)

toggle_channel_plot(*btn*)
Callback when a checkbox is clicked. Emits sigToggleChanged.

Parameters btn: QCheckBox

button that is clicked on

class cued_datalogger.acquisition.RecordingUIs.**ChanConfigUI**(**arg*, ***kwarg*)
Bases: *cued_datalogger.acquisition.RecordingUIs.BaseWidget*

A Channel Plots Configuration widget. Contains:

•ComboBox to switch channel plot info,

•Spinboxes to set the offsets

•Buttons to change the colour of a plot

•Checkbox to hold a signal

•Button to open a window to edit metadata

Attributes

timeOffsetChanged: pyqtsignal	Emits when a time domain offset is changed, Sends out the channel num(int) and the x and y offsets(float,float)
freqOffsetChanged: pyqtsignal	Emits when a frequency domain offset is changed, Sends out the channel num(int) and the x and y offsets(float,float)
sigHoldChanged: pyqtsignal	Emits when a state of holding the plot is changed, Sends out the channel num(int) and the state(bool)
colourReset: pyqtsignal	Emits when a plot colour is reset, Sends out the channel num(int)
colourChanged: pyqtsignal	Emits when a plot colour is changed, Sends out the channel num(int) and the color(QColor)
chans_num_box: QComboBox	The widget to select the channel plot
hold_tickbox: QCheckBox	Toggles whether to hold the signal or not
colbox:QPushButton	Set the colour of the plot
defcol_btn: QPushButton	Reset the colour of the plot to the default colour
meta_btn: QPushButton	Opens the metadata editing window
time_offset_config: List of SpinBox	Sets the X and Y offsets of time domain plot
fft_offset_config: List of SpinBox	Sets the X and Y offsets of frequency domain plot

Methods

initUI()
Reimplemented from BaseWidget.

set_colour_btn(*col*)
Set the colour of the colour button.

Parameters col: QColor

Colour to set

set_offset_step(*cbox*, *step_val*)
Sets the single step of a spinbox

Parameters **cbox: SpinBox**

SpinBox to set

step_val: float

The new value of the single step

set_plot_colour(*reset=False*)
Set the colour of the colour button. Emits either sigColourReset or sigColourChanged

Parameters **reset: bool**

Whether to reset the colour or not

set_plot_offset(*dtype*)
Callback to set the offset. Emits sigTimeOffsetChanged or sigFreqOffsetChanged depending on dtype

Parameters **dtype: str**

Either 'Time' of 'DFT' to indicate the time domain or frequency domain plot respectively

signal_hold(*state*)
Callback to hold the plot. Emits sigHoldChanged

Parameters **dtype: str**

Either 'Time' of 'DFT' to indicate the time domain or frequency domain plot respectively

class cued_datalogger.acquisition.RecordingUIs.**DevConfigUI**(**arg*, ***kwarg*)
Bases: *cued_datalogger.acquisition.RecordingUIs.BaseWidget*

A Channel Plots Configuration widget. Contains widgets to setup the recorder

Attributes

configRecorder: pyqtsignal	Emits the configuration of the recorder is set
typebtngroup: QButtonGroup	Contains the buttons to select source of audio stream Either SoundCard or NI
config_button: QPushButton	Confirm the settings and set up the new recorder
rec: Recorder object	Reference of the Recorder object
configboxes: List of widgets	Widgets for the configuration settings, in order: ['Source','Rate','Channels','Chunk Size','Number of Chunks'] with type, respectively: [QComboBox, QLineEdit, QLineEdit, QLineEdit, QLineEdit]

Methods

config_setup()
Configure the inputs of the config_boxes

Parameters **recorder: Recorder object**

The reference of the Recorder object

display_sources()
Display the available sources from the type of recorder Either SoundCard(myRecorder) or NI(NIRecorder)

initUI()
Reimplemented from BaseWidget.

read_device_config()
Display the available sources from the type of recorder Either SoundCard(myRecorder) or NI(NIRecorder)

Returns recType: str

Type of recorder

configs: list

The configurations ['Source','Rate','Channels','Chunk Size','Number of Chunks'] with type, respectively:[str, int, int, int, int]

set_recorder(*recorder*)
Set the recorder for reference

Parameters **recorder: Recorder object**

The reference of the Recorder object

class cued_datalogger.acquisition.RecordingUIs.**StatusUI**(**arg*, ***kwarg*)
Bases: *cued_datalogger.acquisition.RecordingUIs.BaseWidget*

A Status Bar widget. Contains:

•QStatusBar to display the stream status

•Button to reset the splitters

•Button to resume/pause the stream

•Button to grab a snapshot of the stream

Attributes

statusbar: QStatusBar	Displays the status of the stream
resetView: QPushButton	Reset the splitter view
togglebtn: Recorder object	Resume/pause the stream
sshotbtn: List of widgets	Grab a snapshot of the stream

Methods

initUI()
Reimplemented from BaseWidget.

trigger_message()
Display a message when the recording trigger is set off

class cued_datalogger.acquisition.RecordingUIs.**RecUI**(**arg*, ***kwarg*)
Bases: *cued_datalogger.acquisition.RecordingUIs.BaseWidget*

A Recording Configuration widget. Contains:

- ComboBox to change recording mode,
- **Widgets for setting up the recording:**
  - Recording samples/ duration
  - Triggering
- **Additional widgets for specific recording mode:**
  - Normal: None
  - Average transfer function: Buttons to undo or clear past autospectrum and crossspectrum

Attributes

startRecording: pyqtsignal	Emits when record button is pressed
cancelRecording: pyqtsignal	Emits when cancel button is pressed
undoLastTfAvg: pyqtsignal	Emits when undo last transfer function button is pressed
clearTfAvg: pyqtsignal	Emits when clear past transfer functions button is pressed
switch_rec_box: QComboBox	Switch recording options
rec_boxes: List of Widgets	Configurations: ['Samples','Seconds','Pretrigger','Ref. Channel','Trig. Level'] with types : [QLineEdit, QLineEdit, QLineEdit, QComboBox, QLineEdit]
spec_settings_widget:QStackedWidget	Stacked widget additional settings
input_chan_box: QComboBox	Additional settings to put input channel for average transfer function calculation

Methods

autoset_record_config(*setting*)
Recalculate samples or duration

Parameters **setting: str**

Input type. Either 'Time' or 'Samples'

get_input_channel()

Returns int

Current index of input_chan_box

get_record_config(**arg*)

Returns rec_configs: list

List of recording settings

get_recording_mode()

Returns str

Current text of switch_rec_box

initUI()
Reimplemented from BaseWidget.

reset_configs()
Reset the channels for triggering and reset validators

set_recorder(*recorder*)
Set the recorder reference

toggle_trigger(*string*)
Enable or disable the trigger settings

update_TFavg_count(*val*)
Update the value of the number of recordings for average transfer function

Acquisition Live Graphs

Created on Thu Aug 24 17:35:00 2017

@author: eyt21

This module contains the live graph classes to the acquisition window.

Attributes

CHANLVL_FACTOR: float Not used

TRACE_DECAY: float The decay factor of the peak plots

TRACE_DURATION: float Duration before the peak plots decay

class cued_datalogger.acquisition.RecordingGraph.**LiveGraph**(**args*, ***kwargs*)
Bases: pyqtgraph.widgets.PlotWidget.PlotWidget

A base PlotWidget reimplemented to store extra plot information, such as offsets, colours, and visibility.

Attributes

plotColourChanged: pyqtsignal	Emits when colour of a plot change Sends out QColor
plotlines: list of PlotDataItem	Contains the individual PlotDataItem
plot_xoffset: list of float	Contains the X offset of each plot
plot_yoffset: list of float	Contains the Y offset of each plot
plot_colours:list of QColor	Contains the current colour of each plot
plot_visible: list of bool	Contains the visibility of each plot

Methods

check_line(*line*)
Check whether a plot line exists

Returns int

Index of the plot line, if it exists None otherwise

gen_default_colour()
Generate the default colours of the plots

plot(**arg*, ***kwargs*)
Plot the data and set it to be clickable

Returns PlotDataItem

The plot line, effectively

reset_colour()
Clear the colours of the plots

reset_default_colour(*num*)
Set the default colour of the specified plot

Parameters **num: int**

Index of the line to be set

reset_offsets()
Reset the offsets of the plots

reset_plot_visible()
Reset the visibilities of the plots

reset_plotlines()
Clear all of the lines

set_offset(*num*, *x_off=None*, *y_off=None*)
Set the offsets of the specific line

Parameters **num** : int

Index of the line to be set

x_off : float

X offset of the line to be set, if given a value

x_off : float

Y offset of the line to be set, if given a value

set_plot_colour(*num*, *col*)
Set the colour of the specific line

Parameters **num: int**

index of the line to be set

col: QColor

Colour of the line to be set

toggle_plotline(*num*, *visible*)
Set the visibility of the specific line

Parameters **num: int**

index of the line to be set

visible: bool

Visibility of the line to be set

update_line(*num*, *x=None*, *y=None*, **arg*, ***kwargs*)
Update the existing lines with new data, with the offsets

Parameters **num** : int

index of the line to be set

x : float

X data of the line to be set, if given a value

y : float

Y data of the line to be set, if given a value

The rest to pass to PlotDataItem.setData

class cued_datalogger.acquisition.RecordingGraph.**TimeLiveGraph**(**args*, ***kwargs*)
Bases: *cued_datalogger.acquisition.RecordingGraph.LiveGraph*

Reimplemented LiveGraph. Displays the time domain plot

Attributes

sig_hold: list of bool	Contains whether the signal is being held

Methods

set_sig_hold(*num*, *state*)
Set the hold status of the specific line

Parameters **num: int**

Index of the line to be set

state: bool

Hold status of the line to be set

class cued_datalogger.acquisition.RecordingGraph.**FreqLiveGraph**(**args*, ***kwargs*)
Bases: *cued_datalogger.acquisition.RecordingGraph.LiveGraph*

Reimplemented LiveGraph. Displays the frequency domain plot

Attributes

lastFileDir	

Methods

class cued_datalogger.acquisition.RecordingGraph.**LevelsLiveGraph**(*rec*, **args*, ***kwargs*)
Bases: *cued_datalogger.acquisition.RecordingGraph.LiveGraph*

Reimplemented LiveGraph. Displays the channel levels

Attributes

thresholdChanged: pyqtSignal	Emits when the threshold line is moved Sends out the value of the threshold
peak_plots: list of plotDataItem	The lines which indicate the channels' peaks
peak_trace: list of float	The values of the channels' peaks
trace_counter: list of int	Counter for the peak plots before they decay
chanlvl_pts: list of plotDataItem	Rms plots
chanlvl_bars: list of bool	Instantaneous channels' peaks plots
threshold_line:	The line indicating the trigger threshold
level_colourmap:	The colour for the peak levels

Methods

change_threshold(*arg*)
Set the trigger threshold If arg is str, set the threshold_line to match the value otherwise, emit the value of the threshold_line

Parameters **arg: str or InfiniteLine**

gen_default_colour()
Reimplemented from LiveGraph.

reset_channel_levels()
Reset the channel levels plot

reset_channel_peaks(*rec*)
Reset the channel peaks plot

reset_colour()
Reimplemented from LiveGraph.

reset_default_colour(*chan*)
Reimplemented from LiveGraph.

set_channel_levels(*value*, *maximum*)
Set the value of the levels plots Parameters ————- value: float

rms values

maximum: float Instantaneous maximum value of the plot

set_peaks(*num*, *maximum*)
Set the value of the peak plots Parameters ————- num: int

index of the peak to be set

maximum: float Instantaneous maximum value of the peak

set_plot_colour(*num*, *col*)

Parameters **num: int**

index of the point to be set

col: QColor

Colour of the point to be set

Channel MetaData Window

Created on Wed Aug 2 16:24:57 2017

@author: eyt21

This module contains the widget to open the window to edit metadata in the acqusition window

class `cued_datalogger.acquisition.ChanMetaWin.`**`ChanMetaWin`**(*livewin=None*)
Bases: `PyQt5.QtWidgets.QDialog`

This is the Modal Dialog Window to edit metadata from acquisition window. it shows the channel names on the left in a list, and the metadata on the right.

Attributes

livewin: acquisition window	Window to get the metadata from
all_info: list	Contains metadata for each channel
channel_listview: QListWidget	Display list of names of channels
meta_configs: list	Widget for ('Channel', 'Name', 'Calibration Factor', 'Tags', 'Comments') of types (QLabel, QLineEdit, QLineEdit, QLineEdit, CommentBox)

Methods

`display_metadata`()
Display the selected channel metadata

`export_metadata`()
Export the metadata to the livewin ChannelSet

`initUI`()
Initialise the UI

`update_metadata`(*meta_name*, *UI*)
Update the selected channel metadata

class `cued_datalogger.acquisition.ChanMetaWin.`**`CommentBox`**
Bases: `PyQt5.QtWidgets.QTextEdit`

Reimplement QTextEdit to be similar to QLineEdit, i.e. having editingFinished signal and text()

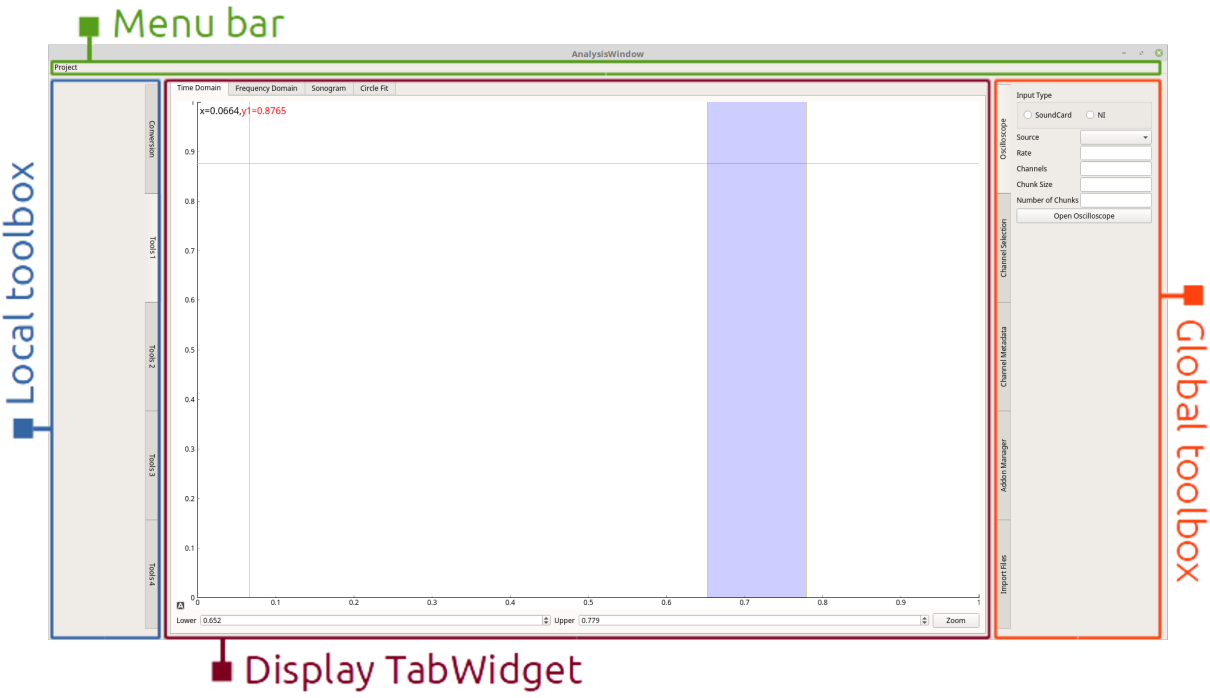
Methods

Analysis

This section contains the documentation for the `AnalysisWindow`.

Window structure

The AnalysisWindow is comprised of three main widgets and a menu bar.



The widgets use PyQt's signal and slot mechanism to interact with each other, rather than interacting directly.

Menu bar

Accessed as `menubar` in `AnalysisWindow`.

Currently the menu bar is only a placeholder, it has no functionality.

The menu bar will be the place for functions that are not tools for interacting with or affecting data and tools that will not be used regularly during normal DataLogger operation. For example, workspaces will be loaded, configured and saved from the menu bar, as this operation will normally only be performed once per session.

Local toolbox

Accessed as `toolbox` in `AnalysisWindow`.

The local toolbox contains all the operations and conversions that are associated with the widget that is currently showing in the display TabWidget. If something changes the channel data, or changes the way that the data is viewed, then it goes in the local toolbox.

Functions in the local toolbox should be grouped into tabs (eg. 'Conversion', 'Peaks') and then into grouped boxes within a tab (eg. 'Transfer function conversion options', 'Sonogram conversion options').

Display TabWidget

Accessed as `display_tabwidget` in `AnalysisWindow`.

This is the central widget for the AnalysisWindow, where graphs, data, and results are displayed. For each section of the analysis window (time domain, sonogram, etc) there is one `QWidget` that is created for display, which is the focal point of that section.

In general it is simply an *InteractivePlotWidget*, but it can contain other widgets (eg. *CircleFitWidget*) if they are absolutely necessary to smooth operation (such as the results tree in the CircleFitWidget).

The user should not have to jump around between the toolboxes and the display TabWidget to view their results. Operations are kept in the toolboxes; the display TabWidget is for data interaction and visualisation.

Note: Currently no decision has been made about how future modal analysis tools will be added to the DataLogger. Will the Circle Fit tab remain solely for circle fitting or will it become a Modal Analysis tab containing options for circle fitting, RFP fitting, etc?

Global toolbox

Accessed as `global_toolbox` in `AnalysisWindow`.

The global toolbox contains operations that have a universal effect, and are not limited to one specific analysis widget. Examples include interacting with channel selection and metadata, or running addons.

The global toolbox is actually contained within a *MasterToolbox*, `global_master_toolbox` to provide an interface symmetric with the local toolbox. However, the user should never need to interact with the MasterToolbox, and all of the global functionality should be located in the the `global_toolbox`.

AnalysisWindow widget

Time domain

class `cued_datalogger.analysis.time_domain.`**`TimeDomainWidget`**(*parent=None*)
Bases: *cued_datalogger.api.pyqtgraph_extensions.InteractivePlotWidget*

The TimeDomainWidget is the main display widget for everything in the time domain.

Methods

`set_selected_channels`(*selected_channels*)
Update which channels are plotted

class `cued_datalogger.analysis.time_domain.`**`TimeToolbox`**(*parent=None*)
Bases: *cued_datalogger.api.toolbox.Toolbox*

Toolbox containing the Time Domain controls.

Attributes

sig_convert_to_sonogram	(pyqtSignal) Signal emitted when the 'Convert to sonogram' button is clicked.
sig_convert_to_fft	(pyqtSignal) Signal emitted when the 'Convert to frequency spectrum' button is clicked.

Methods

Frequency domain

class cued_datalogger.analysis.frequency_domain.**FrequencyDomainWidget**(*parent=None*)
Bases: *cued_datalogger.api.pyqtgraph_extensions.InteractivePlotWidget*

The FrequencyDomainWidget is the main display widget for everything in the frequency domain.

Attributes

channels	(list of Channel) The currently selected channel objects
current_plot_type	(str) Any of 'linear magnitude', 'log magnitude', 'phase', 'real part', 'imaginary part', 'nyquist'. The current type of plot that is displayed.
show_coherence	(bool) If *True*, coherence is also plotted on the axes.

Methods

calculate_spectrum()
Calculate the frequency spectrum of all the selected channels.

calculate_transfer_function(*input_channel=None*)
Calculate the transfer function, using the channel object given by *input_channel* as the input. If no channel specified, treat the first selected channel as input.

set_plot_type(*plot_type*)
Set what type of plot is displayed. *plot_type* can be any of 'linear magnitude', 'log magnitude', 'phase', 'real part', 'imaginary part', 'nyquist'.

set_selected_channels(*selected_channels*)
Update which channels are plotted. Sets *self.channels* to *selected_channels*.

set_show_coherence(*show_coherence*)
Set whether the coherence is displayed.

update_plot(*plot_transfer_function=False*)
If *plot_transfer_function*, plot the transfer function. Otherwise, plot the spectrum.

class cued_datalogger.analysis.frequency_domain.**FrequencyToolbox**(*parent=None*)
Bases: *cued_datalogger.api.toolbox.Toolbox*

Toolbox containing the Frequency Domain controls.

Methods

Sonogram

class cued_datalogger.analysis.sonogram.**SonogramDisplayWidget**(*parent=None*, *window_width=256*, *window_overlap_fraction=8*, *contour_spacing_dB=5*, *num_contours=5*)

Bases: *cued_datalogger.api.pyqtgraph_extensions.ColorMapPlotWidget*

The SonogramDisplayWidget is the main display widget for everything in the sonogram domain.

Methods

calculate_sonogram()
Calculate the sonogram, and store the values in the channel (including autogenerated datasets). Sonogram data is in complex form.

set_selected_channels(*selected_channels*)
Update which channel is being plotted.

update_contour_spacing(*value*)
Slot for updating the plot when the contour spacing is changed.

update_num_contours(*value*)
Slot for updating the plot when the number of contours is changed.

update_plot()
Clear the canvas and replot.

update_window_overlap_fraction(*value*)
Slot for updating the plot when the window overlap fraction is changed.

update_window_width(*value*)
Slot for updating the plot when the window width is changed.

class cued_datalogger.analysis.sonogram.**SonogramToolbox**(*parent=None*)
Bases: *cued_datalogger.api.toolbox.Toolbox*

Toolbox containing Sonogram controls.

Methods

set_selected_channels(*selected_channels*)
Update which channel is being plotted

class cued_datalogger.analysis.sonogram.**MatplotlibSonogramContourWidget**(*sonogram_toolbox=None*, *channel=None*, *contour_spacing_dB=None*, *num_contours=None*)

Bases: cued_datalogger.api.pyqt_extensions.MatplotlibCanvas

A MatplotlibCanvas widget displaying the Sonogram contour plot.

Attributes

fixed_dpi	

Methods

set_selected_channels(*selected_channels*)
Update which channel is being plotted.

update_contour_sequence()
Update the array which says where to plot contours, how many etc.

update_contour_spacing(*value*)
Slot for updating the plot when the contour spacing is changed.

update_num_contours(*value*)
Slot for updating the plot when the number of contours is changed.

update_plot()
Redraw the sonogram on the canvas.

Modal fitting

Circle fitting

class cued_datalogger.analysis.circle_fit.**CircleFitWidget**(*parent: QWidget = None, flags: Union[Qt.WindowFlags, Qt.WindowType] = Qt.WindowFlags()*)

Bases: PyQt5.QtWidgets.QWidget

Methods

error_function(*parameters*)
The error function for least squares fitting.

refresh_nyquist_plot()
Clear the nyquist plot and add the items back in.

refresh_transfer_function_plot()
Clear the transfer function plot and add the items back in.

sdof_get_parameters()
Fit a SDOF peak to the data with a least squares fit, using values from the current peak as a first guess.

sdof_peak_with_offset(*w*, *wn*, *zn*, *an*, *phi*)
An SDOF modal peak fitted to the data using the geometric circle.

set_selected_channels(*selected_channels*)
Update which channels are plotted.

class cued_datalogger.analysis.circle_fit.**CircleFitToolbox**(*parent=None*)

Bases: *cued_datalogger.api.toolbox.Toolbox*

The Toolbox for the CircleFitWidget.

This Toolbox contains the tools for controlling the circle fit. It has two tabs: 'Transfer Function', for tools relating to the construction of a transfer function, and 'Autofit Controls', which contains tools for controlling how the circle is fit to the data.

Attributes

sig_construct_transfer_fn	(pyqtSignal) The signal emitted when a new transfer function is to be constructed.
sig_show_transfer_fn	(pyqtSignal(bool)) The signal emitted when the visibility of the transfer function is changed. Format (visible).

Methods

`cued_datalogger.analysis.circle_fit.`**`fit_circle_to_data`**(*x*, *y*)

Fit a geometric circle to the data given in x, y.

Parameters **x** : ndarray

y : ndarray

Returns **x0** : float

The x-coordinate of the centre of the circle.

y0 : float

The y-coordinate of the centre of the circle.

R0 : float

The radius of the circle.

Notes

This function solves a standard eigenvector formulation of the circle fit problem. See *[R11]* for the derivation.

References

[R11]

Addons

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

Addon structure

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

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

```
#cued_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 (`#cued_datalogger_addon`): This informs the cued_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 `run()` will be accessible by the addon, so don't put any code outside of `run()`.

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.

Addon Manager

Addons are normally run through the *AddonManager*.

Widgets

AddonManager

class cued_datalogger.api.addons.**AddonManager**(*parent: QWidget = None, flags: Union[Qt.WindowFlags, Qt.WindowType] = Qt.WindowFlags()*)

Bases: PyQt5.QtWidgets.QWidget

Methods

discover_addons(*path*)
Find any addons contained in path and load them

Analysis Window

See AnalysisWindow.

ChannelMetadataWidget

class cued_datalogger.api.channel.**ChannelMetadataWidget**(*parent: QWidget = None, flags: Union[Qt.WindowFlags, Qt.WindowType] = Qt.WindowFlags()*)

Bases: PyQt5.QtWidgets.QWidget

Methods

ChannelSelectWidget

class cued_datalogger.api.channel.**ChannelSelectWidget**(*parent=None*)
Bases: PyQt5.QtWidgets.QWidget

A widget used in the Global Toolbox to select channels.

This widget is used as the master controller of what channels are selected. It allows channel selection by checkboxes, an 'Invert Selection' button, a 'Select All' button, a 'Deselect All' button, and Matlab-style list indexing (eg. 1:10:2, 4 selects all the odd channels between 1 and 10 and channel 4). Possible additional features to be implemented include selection by tag and by other channel metadata.

When the channel selection is changed it emits a signal containing the list of currently selected channels. Widgets can be set to receive this signal and set the channels that they are displaying to that list.

Attributes

sig_channel_selection_changed	(pyqtSignal) The signal emitted when the selected channels are changed, containing a list of Channel objects

Methods

selected_channels()
Return a list of all the currently selected `Channel` objects.

selected_channels_index()
Return a list of channel numbers of all currently selected channels.

set_channel_set(*channel_set*)
Set the `ChannelSet` used by this widget.

ColorMapPlotWidget

class cued_datalogger.api.pyqtgraph_extensions.**ColorMapPlotWidget**(*parent=None*, *cmap='jet'*)
Bases: *cued_datalogger.api.pyqtgraph_extensions.InteractivePlotWidget*

An InteractivePlotWidget optimised for plotting color(heat) maps. Uses the Matplotlib colormap given by *cmap* to color the map.

Attributes

lookup_table	(ndarray) The lookup table generated from *cmap* to colour the image with
num_contours	(int) The number of different colour levels to plot
contour_spacing	(int) How closely spaced the colour levels are

Methods

plot_colormap(*x*, *y*, *z*, *num_contours=5*, *contour_spacing_dB=5*)
Plot *x*, *y* and *z* on a colourmap, with colour intervals defined by *num_contours* at *contour_spacing_dB* intervals.

DataImportWidget

class cued_datalogger.api.file_import.**DataImportWidget**(*parent: QWidget = None*, *flags: Union[Qt.WindowFlags, Qt.WindowType] = Qt.WindowFlags()*)
Bases: `PyQt5.QtWidgets.QWidget`

Methods

load_pickle()
This is probably a temporary solution to loading data. Probably have to write a better way of storing data. PLEASE DO NOT OPEN ANY UNTRUSTED PICKLE FILES. UNPICKLING A FILE CAN EXECUTE ARBITRARY CODE, WHICH IS DANGEROUS TO YOUR COMPUTER.

Frequency Domain Widgets

See *Frequency domain*.

MasterToolbox

class cued_datalogger.api.toolbox.**MasterToolbox**(*parent=None*)
Bases: PyQt5.QtWidgets.QStackedWidget

A QStackedWidget of one or more Toolboxes that toggle collapse when the tabBar is double clicked.

In the MasterToolbox, only the top Toolbox is expanded, and all the others are collapsed. When the index is changed with `set_toolbox()`, the top Toolbox is changed and all other Toolboxes are collapsed and hidden. The MasterToolbox is the normal location for all tools and controls in the DataLogger.

Attributes

Inherited attributes :	See PyQt5.QtWidgets.QStackedWidget for inherited attributes.

Methods

add_toolbox(*toolbox*)
Add a Toolbox to the MasterToolbox stack.

set_toolbox(*toolbox_index*)
Set current Toolbox to the Toolbox given by *toolbox_index*, by quick-collapsing and hiding all of the other Toolboxes. The new current Toolbox will be in the same collapse/expand state as the former current Toolbox (ie if the previous Toolbox was collapsed, the new current Toolbox will be collapsed, and vice versa).

toggle_collapse()
Toggle collapse of the MasterToolbox by toggling the collapse of the Toolbox that is on top.

Modal Fitting Widgets

See *Modal fitting*.

Plot Widgets

See *Plot interaction*.

Sonogram Widgets

See *Sonogram*.

Time Domain Widgets

See *Time domain*.

Toolbox

class cued_datalogger.api.toolbox.**Toolbox**(*widget_side='left'*, *parent=None*)
Bases: PyQt5.QtWidgets.QWidget

A side-oriented widget similar to a TabWidget that can be collapsed and expanded.

A Toolbox is designed to be a container for sets of controls, grouped into 'pages' and accessible by a TabBar, in the same way as a TabWidget. A page is normally a QWidget with a layout that contains controls. A widget can be added as a new tab using `addTab()`. The Toolbox has slots for triggering its collapse and expansion, both in an animated mode (soft slide) and a 'quick' mode which skips the animation. Commonly the collapse/expand slots are connected to the tabBar's `tabBarDoubleClicked()` signal. Normally in the DataLogger a Toolbox is created and then added to a *MasterToolbox*, which connects the relevant signals for collapsing and expanding the Toolbox.

Attributes

tabBar	(QTabBar)
tabPages	(QStackedWidget) The stack of widgets that form the pages of the tabs.
collapse_animation	(QPropertyAnimation) The animation that controls how the Toolbox collapses.

Methods

addTab(*widget*, *title*)
Add a new tab, with the page widget *widget* and tab title *title*.

changePage(*index*)
Set the current page to *index*.

clear()
Remove all tabs and pages.

collapse()
Collapse the widget so that only the tab bar is visible.

expand()
Expand the widget so that the pages are visible.

removeTab(*title*)
Remove the tab with title *title*.

toggle_collapse()
If collapsed, expand the widget so the pages are visible. If not collapsed, collapse the widget so that only the tabBar is showing.

Convenience functions and classes

A few simple functions and classes are defined in the DataLogger package to streamline the implementation of some functionality.

cued_datalogger.api.numpy_extensions.**to_dB**(*x*)
A simple function that converts x to dB: `20*np.log10(x)`

cued_datalogger.api.numpy_extensions.**from_dB**(*x*)
A simple function that converts x in dB to a ratio over 1: `10**(x/20)`

class cued_datalogger.api.numpy_extensions.**MatlabList**
Bases: list

A list that allows slicing like Matlab.

eg: `l[1, 2, slice(3, 5), slice(10, 20, 2)]`

Methods

cued_datalogger.api.numpy_extensions.**sdof_modal_peak**(*w*, *wn*, *zn*, *an*, *phi*)
Return a modal peak generated from the given parameters.

Parameters **w** : ndarray

An array of omega (angular frequency) values.

wn [float] The resonant angular frequency.

zn [float] The damping factor.

an [float] The complex modal constant.

Returns ndarray

The modal peak.

rac{a_n}{omega_n^2 - omega^2 + 2izeta_nomega_nomega}

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Bibliography

[R11] Maia, N.M.M., Silva, J.M.M. et al, Theoretical and Experimental Modal Analysis, p221, Research Studies Press, 1997.

Python Module Index

C

Index

Symbols

A

B

C

D

E

F

G

I

L

M

N

O

P

R

S

T

U

W

Z