

Module PINSoftware

Here is a short overall description. When the program is run, it creates an instance of [MachineState](#) (it will be referred to as ms) which represents all the hardware and data analysis. It also gets a dash app (a dash.Dash instance) from [get_app\(\)](#). It then runs the app in a server.

Then a user can connect and look through the app, however to start getting data he need to grab control. When he does this a field of ms is set to his unique id and he can now run an experiment. When the user clicks the start button in the user interface the [MachineState.start_experiment\(\)](#) method of ms is called. When this happens ms creates a new instance of [DataAnalyser](#) (data) and a new [PINSoftware.DataUpdater](#). The [DataAnalyser](#) is like a buffer, but whenever a new datapoint is added to it (using [DataAnalyser.append\(\)](#)), it first does some processing on it and then appends new values to multiple datasets it has as its fields. To be exact, the [DataAnalyser](#) stores the raw data, the “peak voltages” and the “averaged peak voltages” mentioned in the manual and the Help tab of the software. The [PINSoftware.DataUpdater](#) is either a [NiDAQmxDataUpdater](#) (when the hardware is being used) or [LoadedDataUpdater](#) (when in dummy mode). What it does is, in a separate thread from the server it keeps loading data from the instrument as fast as possible and calls the [DataAnalyser.append\(\)](#) method of data. This way data keeps aggregating all the data and the server (or anything else) can easily retrieve it from it. If data saving was turned on, the ms will also get a [PINSoftware.DataSaver](#) (either [CsvDataSaver](#) or [Hdf5DataSaver](#)) and starts it in another thread. Those check in regular intervals for new data in the [DataAnalyser](#) and save them. Lastly the [Profiler](#) also works in the same way, if it was enabled, the ms creates a new instance and starts it. Whenever the [PINSoftware.DataUpdater](#) adds new data it triggers the [Profiler](#) and it then counts those triggers every second and prints them out.

A note on naming, in the documentation I offer refer to “peak voltage”, in code I call that “processed_y”, and “raw data” is just called “ys” in code.

Sub-modules

- [PINSoftware.DashApp](#)
- [PINSoftware.DashComponents](#)
- [PINSoftware.DataAnalyser](#)
- [PINSoftware.DataSaver](#)
- [PINSoftware.DataUpdater](#)
- [PINSoftware.Debugger](#)
- [PINSoftware.MachineState](#)
- [PINSoftware.Profiler](#)
- [PINSoftware.main](#)

Module PINSoftware.DashApp

Functions

Function linear_correct_func

```
def linear_correct_func(
    cor_a,
    cor_b
)
```

Get a linear function with a and b as its coefficients

Function get_app

```
def get_app(
    ms: PINSoftware.MachineState.MachineState
) -> dash.Dash
```

Creates the Dash app and creates all the callbacks. ms is the MachineState instance to which callbacks should be connected. Returns a dash.Dash instance to be run.

Module `PINSoftware.DashComponents`

This file has all the custom dash components used for the app. Some of the most important ones are the graphs so I will give a bit of explanation to them.

There are two types of graphs defined here - [FullRedrawGraph\(\)](#) and [ExtendableGraph\(\)](#). The difference being that only the `ExtendableGraph` can be extended without resending all the data. Both graphs are created in a container with a title, controls and an interval. Both have at least two switches, the “Show graph” and the “Update graph” buttons. When “Show graph” is off the graph is not rendered and its “Update graph” button is disabled. The “Update graph” toggles the interval, when it is enabled the interval runs, otherwise not. Each type uses the interval differently but both use it to update. More on in their respective documentations.

Functions

Function `SingleSwitch`

```
def SingleSwitch(
    id_,
    label,
    default=False
)
```

This returns a dash component containing a single switch (a nice looking checkbox) (not a part of a Checklist) using bootstrap.

`id` is the id of the Checkbox component inside.

`label` is the label shown beside it.

`default` is the default value of the switch.

Function `get_base_graph_callbacks`

```
def get_base_graph_callbacks(
    app: dash.dash.Dash,
    name: str
)
```

This adds the most basic callbacks to a graph. It takes care of two things. Disabling the interval when the “Update graph” switch is off and disabling the “Update graph” when the “Show graph” button is off.

Function `BaseGraph`

```
def BaseGraph(
    app: dash.dash.Dash,
    name: str,
    title: str,
    interval: int = 2000,
    additional_controls=[],
    base_fig={}
)
```

This creates a dash component with the most basic stuff for an updating graph. It sets up the container, the controls and the `dash_core_components.Interval`. It also sets up the basic callbacks using [get_base_graph_callbacks\(\)](#).

`app` is the app to add the callbacks to.

`name` is the `dash_core_components.Graph` component id.

`title` is the title of the graph, shown at the top of the container.

`interval` is the `dash_core_components.Interval` interval in milliseconds.

`additional_controls` is a list of dash components with should be added in the container, this is useful when you want to add additional controls to the graph.

`base_fig` is what to set the figure property of the `dash_core_components.Graph` (it can be changed later though).

Function `get_full_redraw_graph_callbacks`

```
def get_full_redraw_graph_callbacks(
    app: dash.dash.Dash,
    ms: PINSoftware.MachineState.MachineState,
    name: str,
    fig_func,
    **kwargs
)
```

Adds callbacks to a `FullRedrawGraph()`. Specifically, whenever the `dash_core_components.Interval` triggers, if there is `ms.data` then the `fig_func` is called and its result is set as the new figure.

app the dash app to add the callbacks to.

If `fig_func_output` is a keyword argument, then its value is set as the output of the interval trigger callback (this is the one where `fig_func` is used).

If `fig_func_state` is a keyword argument, then its value is set as the state of the interval trigger callback (this is the one where `fig_func` is used).

The only thing that is fixed is the callback inputs.

Function `FullRedrawGraph`

```
def FullRedrawGraph(
    app: dash.dash.Dash,
    ms: PINSoftware.MachineState.MachineState,
    name: str,
    title: str,
    fig_func,
    interval: int = 2000,
    additional_controls=[],
    **kwargs
)
```

Get a graph which whenever it is updated, the whole figure is changed and passed over the network.

app the dash app to add the callbacks to.

ms the `MachineState` to use for checking for data.

name is the `dash_core_components.Graph` component id.

title is the title of the graph, shown at the top of the container.

`fig_func` is the function to call on update. It is only called when `ms.data` is not `None`. Its output is the graph figure by default but can be changed using the `fig_func_output` keyword argument. It can also get more arguments, the callback state can be changed by the `fig_func_state` keyword argument.

`interval` is the `dash_core_components.Interval` interval in milliseconds.

`additional_controls` is a list of dash components with should be added in the container, this is useful when you want to add additional controls to the graph.

Function `get_extendable_graph_callbacks`

```
def get_extendable_graph_callbacks(
    app: dash.dash.Dash,
    ms: PINSoftware.MachineState.MachineState,
```

```

        name: str,
        extend_func,
        base_fig,
        **kwargs
    )

```

Adds callbacks to a [ExtendableGraph\(\)](#). It adds two callbacks, Firstly, whenever the “Stop” button is enabled, the graphs figure is set to `base_fig`. The second one is that whenever the interval triggers, the graph is extended using `extend_func`.

app the dash app to add the callbacks to.

If `extend_func_output` is a keyword argument, then its value is set as the output of the callback using `extend_func` (the interval one). The default is the graph `'extendData'`.

If `extend_func_state` is a keyword argument, then its value is set as the state of the callback using `extend_func` (the interval one). The default is an empty list.

The only thing that is fixed is the callback inputs.

Function `ExtendableGraph`

```

def ExtendableGraph(
    app: dash.dash.Dash,
    ms: PINSoftware.MachineState.MachineState,
    name: str,
    title: str,
    base_fig,
    extend_func,
    interval: int = 2000,
    additional_controls=[],
    **kwargs
)

```

Get a graph to which new data is added instead of redrawing it completely. Its figure is first set to `base_fig` and is then updated using `extend_func`

app the dash app to add the callbacks to.

ms the [MachineState](#) to use for checking for data.

name is the `dash_core_components.Graph` component id.

title is the title of the graph, shown at the top of the container.

`base_fig` is the basic figure. This is what the figure property of the `dash_core_components.Graph` is set to when the page loads and whenever the “Stop” button becomes enabled (this is to reset when the next acquisition starts) (however this means that it will not reset on new data acquisition for users who are not controlling the setup at that point! Just keep that in mind).

`extend_func` is the function to call on ‘update’, this is whenever the interval triggers. Its output is what is sent to the `dash_core_components.Graphs.extendData` property by default but can be changed using the `extend_func_output` keyword argument. It can also get more arguments, the callback state can be changed by the `extend_func_state` keyword argument.

`interval` is the `dash_core_components.Interval` interval in milliseconds.

`additional_controls` is a list of dash components with should be added in the container, this is useful when you want to add additional controls to the graph.

Module PINSoftware.DataAnalyser

Functions

Function remove_outliers

```
def remove_outliers(  
    data  
)
```

Classes

Class DataAnalyser

```
class DataAnalyser(  
    data_frequency: int,  
    plot_buffer_len: int = 200,  
    debugger: PINSoftware.Debugger.Debugger = <PINSoftware.Debugger.Debugger object>,  
    edge_detection_threshold: float = 0.005,  
    average_count: int = 50,  
    correction_func=<function DataAnalyser.<lambda>>  
)
```

This class takes care of data analysis and storage.

Once this function is created, you should call [DataAnalyser.append\(\)](#) to add a new data point, use `DataAnalyser.y` to get the raw data, `DataAnalyser.processed` to get the peak voltages, `DataAnalyser.processed_timestamps` are timestamps corresponding to the peak voltages, `DataAnalyser.averaged_processed_y`s are the averaged peak voltages and `DataAnalyser.averaged_processed_timestamps` are timestamps corresponding to the averages. Lastly `DataAnalyser.markers` and `DataAnalyser.marker_timestamps` are debug markers and their timestamps, those can be anything and are only adjustable from code, they should not be used normally.

All the timestamps used here are based on the length of `DataAnalyser.y`s at the time. This is very useful for two reasons, its easy to calculate so also fast. Bu mostly because later when you plot the data, you can plot the `DataAnalyser.y`s with “x0=0” and “dx=1” and then plot the peak averaged directly and the data will be correctly scaled on the x axis. The problem is however that this assumes that the data comes at a precise frequency but the NI-6002 can offer that so it should be alright.

`data_frequency` is the frequency of the incoming data, this is used for calculating real timestamps and is saved if hdf5 saving is enabled.

`plot_buffer_len` determines how many datapoints should be plotted in the live plot graph (if the server has been run with the graphing option).

`debugger` is the debugger to use.

`edge_detection_threshold`, `average_count` and `correction_func` are processing parameters. They are described in the Help tab of the program.

`edge_detection_threshold` sets the voltage difference required to find a section transition.

`average_count` is how many peak voltages should be averaged to get the averaged peak voltages.

`correction_func` is the function to run the peak voltages through before using them. This is to correct some systematic errors or do some calculations.

Methods

Method actual_append_first

```
def actual_append_first(  
    self,  
    new_processed_y  
)
```

This appends the new processed value, works on the averaged processed values and possibly appends that too. This is when the first processed value comes in. It sets some initial values, after it runs once [DataAnalyser.actual_append_main\(\)](#) is called instead.

Method `actual_append_main`

```
def actual_append_main(
    self,
    new_processed_y
)
```

This appends the new processed value, works on the averaged processed values and possibly appends that too. For the first processed value, [DataAnalyser.actual_append_first\(\)](#) is run instead, but afterwards this is.

Method `handle_processing`

```
def handle_processing(
    self,
    new_y
)
```

This is the main processing function. It gets the new y (which at this point is not in `DataAnalyser.y` yet) and does some processing on it. It may add new values to `DataAnalyser.processed` and `DataAnalyser.averaged_processed_ys` if new values were found through `DataAnalyser.actual_append`. If the data does not add up a warning is printed. I won't describe the logic here as it is described in the manual and also it may still be best to look through the code.

Method `append`

```
def append(
    self,
    new_y
)
```

The main append function through which new data is added. It just passes the value to the processing function and appends it to `DataAnalyser.y` in the end.

Method `plot`

```
def plot(
    self,
    plt
)
```

This is what plots the data on the raw data graph if graphing is enabled

Module `PINSoftware.DataSaver`

This file is somewhat similar to [PINSoftware.DashComponents](#) in that there are a few support definitions and then two implementations of the same thing along with a base class they both inherit from and which sets a common interface. A [PINSoftware.DataSaver](#) here is an object whose instance runs in a separate thread and periodically checks the [DataAnalyser](#) for new data and then saves it.

Classes

Class `Filetype`

```
class Filetype(
    value,
    names=None,
```

```

        *,
        module=None,
        qualname=None,
        type=None,
        start=1
    )

```

An enum to get the possible saving options reliably

Ancestors (in MRO)

- [enum.Enum](#)

Class variables

Variable Csv

Variable Hdf5

Class SavingException

```

class SavingException(
    ...
)

```

A general exception to be raised when an error occurred during saving

Ancestors (in MRO)

- [builtins.Exception](#)
- [builtins.BaseException](#)

Class BaseDataSaver

```

class BaseDataSaver(
    data: PINSoftware.DataAnalyser.DataAnalyser,
    full_filename: str,
    save_interval: float = 1
)

```

This class creates a common interface for all [PINSoftware.DataSavers](#). A new saver should inherit from this class and override the [BaseDataSaver.do_single_save\(\)](#) method to something which does the saving action itself. It can also possibly override the [BaseDataSaver.close\(\)](#) method which is called on ending the saving (usually you may want to close the file objects there).

data is the [DataAnalyser](#) from which the data should be saved.

full_filename is the full path to the file where the data should be saved (with the extension).

save_interval is the interval in which the [PINSoftware.DataSaver](#) should check for new data.

Ancestors (in MRO)

- [threading.Thread](#)

Descendants

- [PINSoftware.DataSaver.CsvDataSaver](#)
- [PINSoftware.DataSaver.Hdf5DataSaver](#)

Methods

Method `do_single_save`

```
def do_single_save(  
    self  
)
```

This method should be overridden. This method should check for new data and save it.

Method `close`

```
def close(  
    self  
)
```

This method may be overridden, it is called at the end of saving

Method `run`

```
def run(  
    self  
)
```

This method is called when `BaseDataSaver.start` is called, it is the main loop

Method `stop`

```
def stop(  
    self  
)
```

This is a simple setter, it is here to be analogous with the way the thread was started (`BaseDataSaver.start`)

Class `CsvDataSaver`

```
class CsvDataSaver(  
    data: PINSoftware.DataAnalyser.DataAnalyser,  
    save_folder: str,  
    save_base_filename: str,  
    **kwargs  
)
```

This is a very simple [PINSoftware.DataSaver](#) with very few options. It saves the peak voltages (`PINSoftware.DataAnalyser.DataAnalyser.processed`) along with their timestamps in a csv file. The csv file format doesn't allow for storing multiple unrelated data easily so this is all.

Everything except `save_folder` and `save_base_filename` is passed to the [BaseDataSaver](#). `save_folder` and `save_base_filename` are combined along with a timestamp into the `full_filename` which is passed to [BaseDataSaver](#). An attempt is also made to open the file, if it fails [SavingException](#) is raised.

Ancestors (in MRO)

- [PINSoftware.DataSaver.BaseDataSaver](#)
- [threading.Thread](#)

Methods

Method `do_single_save`

```
def do_single_save(  
    self  
)
```

.

Method close

```
def close(  
    self  
)
```

Class Hdf5DataSaver

```
class Hdf5DataSaver(  
    data: PINSoftware.DataAnalyser.DataAnalyser,  
    save_folder: str,  
    save_base_filename: str,  
    items: List[str],  
    **kwargs  
)
```

This is the main saver, it can save all the data in an hdf5 file. The processing parameters are saved as attributes. It is possible to choose what data is saved using the items argument.

data and kwargs are passed to [BaseDataSaver](#).

save_folder and save_base_filename are combined along with a timestamp and extension to form the full_filename. The file is then opened, if that failed a [SavingException](#) is raised.

items determine what data gets saved. It is a list of strings and if certain strings are in there, some data gets saved. If it contains “ys” raw data gets saved, “processed_ys” means peak voltages along with their timestamps, “averaged_processed_ys” means averaged peak voltages and their timestamps. Finally “markers” means markers and their timestamps.

Ancestors (in MRO)

- [PINSoftware.DataSaver.BaseDataSaver](#)
- [threading.Thread](#)

Methods

Method do_single_save

```
def do_single_save(  
    self  
)
```

Method close

```
def close(  
    self  
)
```

Module PINSoftware.DataUpdater

Classes

Class BaseDataUpdater

```
class BaseDataUpdater(  
    data: PINSoftware.DataAnalyser.DataAnalyser,  
    debugger: PINSoftware.Debugger.Debugger = <PINSoftware.Debugger.Debugger object>,
```

```

        profiler: PINSoftware.Profiler.Profiler = None
    )

```

Base class for PINSoftware.DataUpdater.DataUpdaters, it takes care of the profiler, stopping lays out the main loop ([BaseDataUpdater.run\(\)](#)) and so on. It provides a common interface.

data is the [DataAnalyser](#) to add the new data to.

debugger is the [Debugger](#) to use for printouts.

profiler is the [Profiler](#) to use (or None if a profiler should not be run).

Ancestors (in MRO)

- [threading.Thread](#)

Descendants

- [PINSoftware.DataUpdater.LoadedDataUpdater](#)
- [PINSoftware.DataUpdater.NiDAQmxDataUpdater](#)

Methods

Method on_start

```

    def on_start(
        self
    )

```

This is called when the [BaseDataUpdater.run\(\)](#) method is called. This method is meant to be overridden and is a way to run some code at the beginning of the run, it.

Method loop

```

    def loop(
        self
    ) -> int

```

This should be overridden by the actual data loading. This will be called continuously as the [PINSoftware.DataUpdater](#) runs. It should return the number of datapoints added (for the Profiler to use).

Method on_stop

```

    def on_stop(
        self
    )

```

This is called when the [BaseDataUpdater.run\(\)](#) method is finishing. This method is meant to be overridden and is a way to run some code at the end of the run, it.

Method run

```

    def run(
        self
    )

```

This method provides the main loop. This method is called when [BaseDataUpdater.start](#) is called.

Method stop

```

    def stop(
        self
    )

```

Just a simple setter, it is here to be consistent with starting by calling [BaseDataUpdater.start](#)

Class LoadedDataUpdater

```
class LoadedDataUpdater(  
    filename: str,  
    *args,  
    freq: int = 50000,  
    **kwargs  
)
```

This [PINSoftware.DataUpdater](#) is used for the “dummy” mode. It reads data from a text file and adds it to the [DataAnalyser](#) at the specified frequency.

filename is the path to the file to read the data from. The file should be a text file with a number on each line, those numbers are the ones added to the [DataAnalyser](#).

freq is the frequency is the simulated source, it will add this many datapoints per second.

args and kwargs are passed to the [BaseDataUpdater](#).

Ancestors (in MRO)

- [PINSoftware.DataUpdater.BaseDataUpdater](#)
- [threading.Thread](#)

Class NiDAQmxDataUpdater

```
class NiDAQmxDataUpdater(  
    *args,  
    **kwargs  
)
```

This is the most important [PINSoftware.DataUpdater](#), this is the one actually reading from the NI-6002. It first checks if there is exactly one device and if the device is the NI-6002, if not, the app crashes. If it is, then it sets up a nidaqmx.Task and adds the correct channel to it. It then sets the task to continuous acquisition and reads the data.

args and kwargs are passed to the [BaseDataUpdater](#).

Ancestors (in MRO)

- [PINSoftware.DataUpdater.BaseDataUpdater](#)
- [threading.Thread](#)

Module PINSoftware.Debugger

Classes

Class Debugger

```
class Debugger(  
    exit_on_error=True  
)
```

A very simple IO handler, it is here to make printouts more consistent and easy to find. Calling [Debugger.error\(\)](#) is also the correct way to exit on error.

If exit_on_error is true, then whenever Debugger.exit is called, the program halts.

Methods

Method info

```
def info(  
    self,  
    msg  
)
```

Method warning

```
def warning(  
    self,  
    msg  
)
```

Method error

```
def error(  
    self,  
    msg,  
    n=1  
)
```

Module PINSoftware.MachineState

Classes

Class MachineState

```
class MachineState(  
    plt,  
    dummy: bool,  
    dummy_data_file: str,  
    profiler: bool = False,  
    plot_update_interval: int = 100,  
    log_directory: str = 'logs'  
)
```

This is the main class covering all hardware control and data analysis (everything except the UI). There should always be only one instance at a time and the program keeps it for the whole duration of the run.

plt should be the matplotlib.pyplot module or something equivalent, this is for plotting the live data graph on the host machine when the graphing option is enabled.

dummy determines whether the data should be grabbed from the NI-6002 or a dummy file. If dummy is True, dummy_data_file should be specified otherwise the program will crash.

dummy_data_file is a path to the data to use for dummy mode.

profiler is whether the DataUpdater should be profiled.

plot_update_interval is the update interval of the live data graph.

log_directory is the directory where to put saved data.

Methods

Method init_graph

```
def init_graph(  
    self  
)
```

Setup for the live graphing

Method `animate`

```
def animate(  
    self,  
    i  
)
```

The `animate` function for the `animation.FuncAnimation` class

Method `onKeyPress`

```
def onKeyPress(  
    self,  
    event  
)
```

The function to call when a key is pressed in the live graph window

Method `run_graphing`

```
def run_graphing(  
    self  
)
```

This runs the actual graphing, this hangs until the window is closed

Method `grab_control`

```
def grab_control(  
    self,  
    controller_sid  
)
```

Wrapper around the `MachineState.controller` field where extra effects can be added, called when someone grabs control.

Method `release_control`

```
def release_control(  
    self  
)
```

Wrapper around the `MachineState.controller` field where extra effects can be added, called when the control is released.

Method `start_experiment`

```
def start_experiment(  
    self,  
    save_base_filename: str = None,  
    save_filetype: PINSoftware.DataSaver.Filetype = Filetype.Csv,  
    items: List[str] = ['ys', 'processed_ys'],  
    **kwargs  
)
```

This starts a data acquisition run. It creates a new [DataAnalyser](#) and an appropriate `DataUpdater`. Then it may also create and start a `DataSaver` and/or a `Profiler` based on the situation.

`save_base_filename` is the base part of the filename for the new log file. The `save_base_filename` is appended with a timestamp and an appropriate file extension and that makes up the filename.

`save_filetype` determines the `DataSaver` type, more information in [PINSoftware.DataSaver](#).

`items` is used when `save_filetype` is [Filetype.Hdf5](#) and is passed to the [Hdf5DataSaver](#).

kwargs are passed to the new [DataAnalyser](#) instance.

More information on how it all works look in the module documentation: [PINSoftware](#).

Method `stop_experiment`

```
def stop_experiment(  
    self  
)
```

This stops the current experiment and all the threads working on it

Method `stop_everything`

```
def stop_everything(  
    self  
)
```

This currently just calls `stop_experiment` but there might be stuff added here, this is meant to be a sort of stop all button

Module `PINSoftware.Profiler`

Classes

Class `Profiler`

```
class Profiler(  
    name: str = 'PROFILER',  
    start_delay: float = 1  
)
```

A very simple profiler. It keeps an internal counter and every second it prints out its value and resets it. [Profiler.add_count\(\)](#) is used to increment the counter.

name is the name of the profiler, this is used when printing out so that the output is clear.

start_delay is the time to wait after `Profiler.start` was called before actually starting.

Ancestors (in MRO)

- [threading.Thread](#)

Methods

Method `add_count`

```
def add_count(  
    self,  
    counts=1  
)
```

Call this to increase the counter by counts

Method `run`

```
def run(  
    self  
)
```

Method `stop`

```
def stop(  
    self  
)
```

Simple setter to stop the profiler

Module `PINSoftware.main`

Functions

Function `main`

```
def main()
```

The program entrypoint, here the arguments are parser and the program is started. If graphing is enabled, then the web server goes in a non-main thread, otherwise otherwise.

Generated by *pdoc* 0.8.4 (<https://pdoc3.github.io>).