enDAQ Documentation

Release 1.1.1.post1

Nov 04, 2021

CONTENTS:

CHAPTER

ONE

ENDAQ API REFERENCE

1.1 *endaq.ide*

endaq.ide.**extract_time**(*doc*, *out*, *start=0*, *end=None*, *channels=None*, ***kwargs*)

Efficiently extract data within a certain interval from an IDE file. Note that due to the way data is stored in an IDE, the exported interval will be slightly wider than the specified start and end times; this ensures the data is copied verbatim and without loss.

The *start* and *end* times, if used, may be specified in several ways:

- *int*/*float* (Microseconds from the recording start)
- *str* (formatted as a time from the recording start, e.g., *MM:SS*, *HH:MM:SS*, *DDd HH:MM:SS*). More examples:
  - `":01"` or `":1"` or `"1s"` (1 second)
  - `"22:11"` (22 minutes, 11 seconds)
  - `"3:22:11"` (3 hours, 22 minutes, 11 seconds)
  - `"1d 3:22:11"` (1 day, 3 hours, 22 minutes, 11 seconds)
- *datetime.timedelta* or *pandas.Timedelta* (time from the recording start)
- *datetime.datetime* (an explicit UTC time)

Parameters

- **doc** – A *Dataset* or the name of a local IDE file. *Dataset* objects do not have to be fully imported.
- **out** – A filename or stream to which to save the extracted data.
- **start** – The starting time. Defaults to the start of the recording.
- **end** – The ending time. Defaults to the end of the recording.
- **channels** – A list of channel IDs to specifically export. If *None*, all channels will be exported. Note excluded channels will still appear in the new IDE's *channels* dictionary, but the file will contain no data for them.

Returns The total number of bytes written, and total number of ChannelDataBlock elements copied.

endaq.ide.**filter_channels**(*channels*, *measurement_type=<MeasurementType: Any/all>*)

Filter a list of *Channel* and/or *SubChannel* instances by their measurement type(s).

Parameters

- **channels** – A list or dictionary of channels/subchannels to filter.
- **measurement_type** – A *MeasurementType*, a measurement type 'key' string, or a string of multiple keys generated by adding and/or subtracting *MeasurementType* objects. Any 'subtracted' types will be excluded.

endaq.ide.**get_channel_table**(*dataset*, *measurement_type=<MeasurementType: Any/all>*, *start=0*, *end=None*, *formatting=None*, *index=True*, *precision=4*, *timestamps=False*, ***kwargs*)

Get summary data for all *SubChannel* objects in a *Dataset* that contain one or more type of sensor data. By using the optional *start* and *end* parameters, information can be retrieved for a specific interval of time.

The *start* and *end* times, if used, may be specified in several ways:

- *int*/*float* (Microseconds from the recording start)
- *str* (formatted as a time from the recording start, e.g., *MM:SS*, *HH:MM:SS*, *DDd HH:MM:SS*). More examples:
  - `":01"` or `":1"` or `"1s"` (1 second)
  - `"22:11"` (22 minutes, 11 seconds)
  - `"3:22:11"` (3 hours, 22 minutes, 11 seconds)
  - `"1d 3:22:11"` (1 day, 3 hours, 22 minutes, 11 seconds)
- *datetime.timedelta* or *pandas.Timedelta* (time from the recording start)
- *datetime.datetime* (an explicit UTC time)

Parameters

- **dataset** – A *idelib.dataset.Dataset* or a list of channels/subchannels from which to build the table.
- **measurement_type** – A *MeasurementType*, a measurement type 'key' string, or a string of multiple keys generated by adding and/or subtracting *MeasurementType* objects to filter the results. Any 'subtracted' types will be excluded.
- **start** – The starting time. Defaults to the start of the recording.
- **end** – The ending time. Defaults to the end of the recording.
- **formatting** – A dictionary of additional style/formatting items (see *pandas.DataFrame.style.format()*). If *False*, no additional formatting is applied.
- **index** – If *True*, show the index column on the left.
- **precision** – The default decimal precision to display. Can be changed later.
- **timestamps** – If *True*, show the start and end as raw microsecond timestamps.

Returns A table (*pandas.io.formats.style.Styler*) of summary data.

Return type pandas.DataFrame

endaq.ide.**get_channels**(*dataset*, *measurement_type=<MeasurementType: Any/all>*, *subchannels=True*)

Get a list of *Channel* or *SubChannel* instances from a *Dataset* by their measurement type(s).

Parameters

- **dataset** – The *Dataset* from which to retrieve the list.
- **measurement_type** – A *MeasurementType*, a measurement type 'key' string, or a string of multiple keys generated by adding and/or subtracting *MeasurementType* objects. Any 'subtracted' types will be excluded.

- **subchannels** – If *False*, get only *Channel* objects. If *True*, get only *SubChannel* objects.

Returns A list of matching *SubChannel* instances from the *Dataset*.

endaq.ide.**get_doc**(*name=None*, *filename=None*, *url=None*, *parsed=True*, *start=0*, *end=None*, *localfile=None*, *params=None*, *cookies=None*, ***kwargs*)

Retrieve an IDE file from either a file or URL.

Note: *name*, *filename*, and *url* are mutually exclusive arguments. One and only one must be specified. Attempting to supply more than one will generate an error.

Example usage:

```
get_doc("my_recording.ide")
get_doc("https://example.com/remote_recording.ide")
get_doc(filename="my_recording.ide")
get_doc(url="https://example.com/remote_recording.ide")
get_doc(filename="my_recording.ide", start="1:23")
```

The *start* and *end* times, if used, may be specified in several ways:

- *int*/*float* (Microseconds from the recording start)
- *str* (formatted as a time from the recording start, e.g., *MM:SS*, *HH:MM:SS*, *DDd HH:MM:SS*). More examples:
  - `":01"` or `":1"` or `"1s"` (1 second)
  - `"22:11"` (22 minutes, 11 seconds)
  - `"3:22:11"` (3 hours, 22 minutes, 11 seconds)
  - `"1d 3:22:11"` (1 day, 3 hours, 22 minutes, 11 seconds)
- *datetime.timedelta* or *pandas.Timedelta* (time from the recording start)
- *datetime.datetime* (an explicit UTC time)

Parameters

- **name** – The name or URL of the IDE. The method of fetching it will be automatically chosen based on how it is formatted.
- **filename** – The name of an IDE file. Supplying a name this way will force it to be read from a file, avoiding the possibility of accidentally trying to retrieve it via URL.
- **url** – The URL of an IDE file. Supplying a name this way will force it to be read from a URL, avoiding the possibility of accidentally trying to retrieve it from a local file.
- **parsed** – If *True* (default), the IDE will be fully parsed after it is fetched. If *False*, only the file metadata will be initially loaded, and a call to *idelib.importer.readData()*. This can save time.
- **start** – The starting time. Defaults to the start of the recording. Only applicable if *parsed* is *True*.
- **end** – The ending time. Defaults to the end of the recording. Only applicable if *parsed* is *True*.
- **localfile** – The name of the file to which to write data recieved from a URL. If none is supplied, a temporary file will be used. Only applicable when opening a URL.
- **params** – Additional URL request parameters. Only applicable when opening a URL.

- **cookies** – Additional browser cookies for use in the URL request. Only applicable when opening a URL.

Returns The fetched IDE data.

Additionally, *get_doc()* will accept the keyword arguments for *idelib.importer.importFile()* or *idelib.importer.openFile()*

endaq.ide.**get_measurement_type**(*channel*)

Get the appropriate *MeasurementType* object for a given *SubChannel*. Calling with a *Channel* returns a list of *MeasurementType* objects, with one for each child *SubChannel*.

Parameters **channel** – A *Channel* or *SubChannel* instance (e.g., from a *Dataset*).

Returns A *MeasurementType* object (for a *SubChannel*), or a list of *MeasurentType* objects (one for each child) if a *Channel* was supplied.

endaq.ide.**to_pandas**(*channel*, *time_mode='datetime'*)

Read IDE data into a pandas DataFrame.

Parameters

- **channel** (*Union[idelib.dataset.Channel, idelib.dataset.SubChannel]*) – a *Channel* object, as produced from *Dataset.channels* or *endaq.ide.get_channels*
- **time_mode** – how to temporally index samples; each mode uses either relative times (with respect to the start of the recording) or absolute times (i.e., date-times): - "seconds" - a *pandas.Float64Index* of relative timestamps, in seconds - "timedelta" - a *pandas.TimeDeltaIndex* of relative timestamps - "datetime" - a *pandas.DateTimeIndex* of absolute timestamps

Returns a *pandas.DataFrame* containing the channel's data

Return type pandas.core.frame.DataFrame

1.2 *endaq.calc*

1.2.1 *endaq.calc.filters*

endaq.calc.filters.**butterworth**(*df*, *low_cutoff=1.0*, *high_cutoff=None*, *half_order=3*)

Apply a lowpass and/or a highpass Butterworth filter to an array.

This function uses Butterworth filter designs, and implements the filter(s) as bi-directional digital biquad filters, split into second-order sections.

Parameters

- **df** (*pandas.core.frame.DataFrame*) – the input data; cutoff frequencies are relative to the timestamps in *df.index*
- **low_cutoff** (*Optional[float]*) – the low-frequency cutoff, if any; frequencies below this value are rejected, and frequencies above this value are preserved
- **high_cutoff** (*Optional[float]*) – the high-frequency cutoff, if any; frequencies above this value are rejected, and frequencies below this value are preserved
- **half_order** (*int*) – half of the order of the filter; higher orders provide more aggressive stopband reduction

Returns the filtered data

Return type pandas.core.frame.DataFrame

See also:

SciPy Butterworth filter design Documentation for the butterworth filter design function.

1.2.2 *endaq.calc.integrate*

endaq.calc.integrate.**integrals**(*df*, *n=1*, *highpass_cutoff=None*, *tukey_percent=0*)
Calculate *n* integrations of the given data.

Parameters

- **df** (*pandas.core.frame.DataFrame*) – the data to integrate, indexed with timestamps
- **n** (*int*) – the number of integrals to calculate
- **highpass_cutoff** (*Optional[float]*) – the cutoff frequency for the initial highpass filter; this is used to remove artifacts caused by DC trends

Returns a length *n+1* list of the kth-order integrals from 0 to n (inclusive)

Return type List[pandas.core.frame.DataFrame]

See also:

SciPy trapezoid integration Documentation for the integration function used internally.

SciPy Butterworth filter design Documentation for the butterworth filter design function used in preprocessing.

SciPy Tukey window Documentation for the Tukey window function used in preprocessing.

endaq.calc.integrate.**iter_integrals**(*df*, *highpass_cutoff=None*, *filter_half_order=3*, *tukey_percent=0*)
Iterate over conditioned integrals of the given original data.

Parameters

- **df** (*pandas.core.frame.DataFrame*) – the input data
- **highpass_cutoff** (*Optional[float]*) – the cutoff frequency of a preconditioning highpass filter; if None, no filter is applied
- **filter_half_order** (*int*) – the half-order of the preconditioning highpass filter, if used
- **tukey_percent** (*float*) – the alpha parameter of a preconditioning tukey filter; if 0, no filter is applied

Returns an iterable over the data's successive integrals; the first item is the preconditioned input data

Return type Iterable[pandas.core.frame.DataFrame]

See also:

SciPy trapezoid integration Documentation for the integration function used internally.

SciPy Butterworth filter design Documentation for the butterworth filter design function used in preprocessing.

SciPy Tukey window Documentation for the Tukey window function used in preprocessing.

1.2.3 *endaq.calc.psd*

endaq.calc.psd.**differentiate**(*df*, *n=1*)
Perform time-domain differentiation on periodogram data.

Parameters

- **df** (*pandas.core.frame.DataFrame*) – a periodogram
- **n** (*float*) – the time derivative order; negative orders represent integration

Returns a periodogram of the time-derivated data

Return type pandas.core.frame.DataFrame

endaq.calc.psd.**to_jagged**(*df*, *freq_splits*, *agg='mean'*)
Calculate a periodogram over non-uniformly spaced frequency bins.

Parameters

- **df** (*pandas.core.frame.DataFrame*) – the returned values from *endaq.calc.psd.welch*
- **freq_splits** (*numpy.array*) – the boundaries of the frequency bins; must be strictly increasing
- **agg** (*Union[Literal['mean', 'sum'], Callable[[numpy.ndarray, int], float]]*) – the method for aggregating values into bins; 'mean' preserves the PSD's area-under-the-curve, 'sum' preserves the PSD's "energy"

Returns a periodogram with the given frequency spacing

Return type pandas.core.frame.DataFrame

endaq.calc.psd.**to_octave**(*df*, *fstart=1*, *octave_bins=12*, ***kwargs*)
Calculate a periodogram over log-spaced frequency bins.

Parameters

- **df** (*pandas.core.frame.DataFrame*) – the returned values from *endaq.calc.psd.welch*
- **fstart** (*float*) – the first frequency bin, in Hz; defaults to 1 Hz
- **octave_bins** (*float*) – the number of frequency bins in each octave; defaults to 12
- **kwargs** – other parameters to pass directly to *to_jagged*

Returns a periodogram with the given logarithmic frequency spacing

Return type pandas.core.frame.DataFrame

endaq.calc.psd.**vc_curves**(*accel_psd*, *fstart=1*, *octave_bins=12*)
Calculate Vibration Criterion (VC) curves from an acceration periodogram.

Parameters

- **accel_psd** (*pandas.core.frame.DataFrame*) – a periodogram of the input acceleration
- **fstart** (*float*) – the first frequency bin
- **octave_bins** (*float*) – the number of frequency bins in each octave; defaults to 12

Returns the Vibration Criterion (VC) curve of the input acceleration

Return type pandas.core.frame.DataFrame

endaq.calc.psd.**welch**(*df*, *bin_width=1*, ***kwargs*)
Perform *scipy.signal.welch* with a specified frequency spacing.

Parameters

- **df** (*pandas.core.frame.DataFrame*) – the input data
- **bin_width** (*float*) – the desired width of the resulting frequency bins, in Hz; defaults to 1 Hz
- **kwargs** – other parameters to pass directly to *scipy.signal.welch*

Returns a periodogram

Return type pandas.core.frame.DataFrame

See also:

SciPy Welch's method Documentation for the periodogram function wrapped internally.

1.2.4 *endaq.calc.shock*

endaq.calc.shock.**abs_accel**(*accel*, *omega*, *damp=0.0*)

Calculate the absolute acceleration for a SDOF system.

The "absolute acceleration" follows the transfer function: H(s) = L{x"(t)}(s) / L{y"(t)}(s) = X(s)/Y(s)

for the PDE: x" + (2)x' + (^{2})x = (2)y' + (^{2})y

Parameters

- **accel** (*pandas.core.frame.DataFrame*) – the absolute acceleration y"
- **omega** (*float*) – the natural frequency of the SDOF system
- **damp** (*float*) – the damping coefficient of the SDOF system

Returns the absolute acceleration x" of the SDOF system

Return type pandas.core.frame.DataFrame

See also:

An Introduction To The Shock Response Spectrum, Tom Irvine, 9 July 2012

SciPy transfer functions Documentation for the transfer function class used to characterize the relative displacement calculation.

SciPy biquad filter Documentation for the biquad function used to implement the transfer function.

endaq.calc.shock.**enveloping_half_sine**(*pvss*, *damp=0.0*)

Characterize a half-sine pulse whose PVSS envelopes the input.

Parameters

- **pvss** (*pandas.core.frame.DataFrame*) – the PVSS to envelope
- **damp** (*float*) – the damping factor used to generate the input PVSS

Returns a tuple of amplitudes and periods, each pair of which describes a half-sine pulse

Return type Tuple[pandas.core.series.Series, pandas.core.series.Series]

See also:

Pseudo Velocity Shock Spectrum Rules For Analysis Of Mechanical Shock, Howard A. Gaberson

endaq.calc.shock.**rel_displ**(*accel*, *omega*, *damp=0.0*)

Calculate the relative displacement for a SDOF system.

The "relative" displacement follows the transfer function: $H(s) = L\{z(t)\}(s) / L\{y''(t)\}(s) = (1/s^2)(Z(s)/Y(s))$

for the PDE: $z'' + (2)z' + (^2)z = -y''$

Parameters

- **accel** (*pandas.core.frame.DataFrame*) – the absolute acceleration y"
- **omega** (*float*) – the natural frequency of the SDOF system
- **damp** (*float*) – the damping coefficient of the SDOF system

Returns the relative displacement z of the SDOF system

Return type pandas.core.frame.DataFrame

See also:

Pseudo Velocity Shock Spectrum Rules For Analysis Of Mechanical Shock, Howard A. Gaberson

SciPy transfer functions Documentation for the transfer function class used to characterize the relative displacement calculation.

SciPy biquad filter Documentation for the biquad function used to implement the transfer function.

endaq.calc.shock.**shock_spectrum**(*accel*, *freqs*, *damp=0.0*, *mode='srs'*, *two_sided=False*, *aggregate_axes=False*)

Calculate the shock spectrum of an acceleration signal.

Parameters

- **accel** (*pandas.core.frame.DataFrame*) – the absolute acceleration y"
- **freqs** (*numpy.ndarray*) – the natural frequencies across which to calculate the spectrum
- **damp** (*float*) – the damping coefficient , related to the Q-factor by = 1/(2Q); defaults to 0
- **mode** (*Literal['srs', 'pvss']*) – the type of spectrum to calculate:
  - `'srs'` (default) specifies the Shock Response Spectrum (SRS)
  - `'pvss'` specifies the Pseudo-Velocity Shock Spectrum (PVSS)
- **two_sided** (*bool*) – whether to return for each frequency: both the maximum negative and positive shocks (*True*), or simply the maximum absolute shock (*False*; default)
- **aggregate_axes** (*bool*) – whether to calculate the column-wise resultant (*True*) or calculate spectra along each column independently (*False*; default)

Returns the shock spectrum

Return type pandas.core.frame.DataFrame

See also:

Pseudo Velocity Shock Spectrum Rules For Analysis Of Mechanical Shock, Howard A. Gaberson

An Introduction To The Shock Response Spectrum, Tom Irvine, 9 July 2012

SciPy transfer functions Documentation for the transfer function class used to characterize the relative displacement calculation.

SciPy biquad filter Documentation for the biquad function used to implement the transfer function.

1.2.5 *endaq.calc.stats*

endaq.calc.stats.**L2_norm**(*array*, *axis=None*, *keepdims=False*)
Compute the L2 norm (a.k.a. the Euclidean Norm).

Parameters

- **array** (*np.ndarray*) – the input array
- **axis** (*Union[None, typing.SupportsIndex, Sequence[typing.SupportsIndex]]*) – the axis/axes along which to aggregate; if *None*, the L2 norm is computed along the flattened array
- **keepdims** (*bool*) – if *True*, the axes which are reduced are left in the result as dimensions with size one; if *False* (default), the reduced axes are removed

Returns an array containing the computed values

Return type np.ndarray

endaq.calc.stats.**max_abs**(*array*, *axis=None*, *keepdims=False*)
Compute the maximum of the absolute value of an array.

This function should be equivalent to, but generally use less memory than *np.amax(np.abs(array))*.

Specifically, it generates the absolute-value maximum from *np.amax(array)* and *-np.amin(array)*. Thus instead of allocating space for the intermediate array *np.abs(array)*, it allocates for the axis-collapsed smaller arrays *np.amax(array)* & *np.amin(array)*.

Note - this method does **not** work on complex-valued arrays.

Parameters

- **array** (*np.ndarray*) – the input data
- **axis** (*Union[None, typing.SupportsIndex, Sequence[typing.SupportsIndex]]*) – the axis/axes along which to aggregate; if *None*, the absolute maximum is computed along the flattened array
- **keepdims** (*bool*) – if *True*, the axes which are reduced are left in the result as dimensions with size one; if *False* (default), the reduced axes are removed

Returns an array containing the computed values

Return type np.ndarray

endaq.calc.stats.**rms**(*array*, *axis=None*, *keepdims=False*)
Calculate the root-mean-square (RMS) along a given axis.

Parameters

- **array** (*np.ndarray*) – the input array
- **axis** (*Union[None, typing.SupportsIndex, Sequence[typing.SupportsIndex]]*) – the axis/axes along which to aggregate; if *None*, the RMS is computed along the flattened array
- **keepdims** (*bool*) – if *True*, the axes which are reduced are left in the result as dimensions with size one; if *False* (default), the reduced axes are removed

Returns an array containing the computed values

Return type np.ndarray

endaq.calc.stats.**rolling_rms**(*df*, *window_len*, **args*, ***kwargs*)
Calculate a rolling root-mean-square (RMS) over a pandas *DataFrame*.

This function is equivalent to, but computationally faster than the following:

```
df.rolling(window_len).apply(endaq.calc.stats.rms)
```

Parameters

- **df** (*Union[pandas.core.frame.DataFrame, pandas.core.series.Series]*) – the input data
- **window_len** (*int*) – the length of the rolling window
- **args** – the positional arguments to pass into *df.rolling().mean*
- **kwargs** – the keyword arguments to pass into *df.rolling().mean*

Returns the rolling-windowed RMS

Return type Union[pandas.core.frame.DataFrame, pandas.core.series.Series]

See also:

Pandas Rolling Mean method Pandas Rolling Standard Deviation method - similar to this function, but first removes the windowed mean before squaring

1.2.6 *endaq.calc.utils*

endaq.calc.utils.**logfreqs**(*df*, *init_freq=None*, *bins_per_octave=12*)
Calculate a sequence of log-spaced frequencies for a given dataframe.

Parameters

- **df** (*pandas.core.frame.DataFrame*) – the input data
- **init_freq** (*Optional[float]*) – the initial frequency in the sequence; if None (default), use the frequency corresponding to the data's duration
- **bins_per_octave** (*float*) – the number of frequencies per octave

Returns an array of log-spaced frequencies

Return type numpy.ndarray

endaq.calc.utils.**resample**(*df*, *sample_rate=None*)
Resample a dataframe to a desired sample rate (in Hz)

Parameters

- **df** (*pandas.core.frame.DataFrame*) – The DataFrame to resample, indexed by time
- **sample_rate** (*Optional[float]*) – The desired sample rate to resample the given data to. If one is not supplied, then it will use the same as it currently does, but make the time stamps uniformally spaced

Returns The resampled data in a DataFrame

Return type pandas.core.frame.DataFrame

endaq.calc.utils.**sample_spacing**(*df*, *convert='to_seconds'*)
Calculate the average spacing between individual samples.

For time indices, this calculates the sampling period *dt*.

Parameters

- **df** (*pandas.core.frame.DataFrame*) – the input data
- **convert** (*Literal[None, 'to_seconds']*) – if *"to_seconds"* (default), convert any time objects into floating-point seconds

endaq.calc.utils.**to_dB**(*data*, *reference*, *squared=False*)

Scale data into units of decibels.

Decibels are a log-scaled ratio of some value against a reference; typically this is expressed as follows:

$$dB = 10 \log 10 \left(\frac{x}{x_{\text{ref}}} \right)$$

By convention, "decibel" units tend to operate on units of *power*. For units that are proportional to power *when squared* (e.g., volts, amps, pressure, etc.), their "decibel" representation is typically doubled (i.e., $dB = 10 \log 20(...)$). Users can specify which scaling to use with the *squared* parameter.

Note: Decibels can **NOT** be calculated from negative values.

For example, to calculate dB on arbitrary time-series data, typically data is first aggregated via a total or a rolling RMS or PSD, and the non-negative result is then scaled into decibels.

Parameters

- **data** (*numpy.ndarray*) – the input data
- **reference** (*float*) – the reference value corresponding to 0dB
- **squared** (*bool*) – whether the input data & reference value are pre-squared; defaults to *False*

1.3 *endaq.cloud*

1.3.1 *endaq.cloud.core*

Core enDAQ Cloud communication API

class endaq.cloud.core.**EndaqCloud**(*api_key=None*, *env=None*, *test=True*)

A representation of a connection to an enDAQ Cloud account, providing a high-level interface for accessing its contents.

Constructor for an *EndaqCloud* object, which provides access to an enDAQ Cloud account.

Parameters

- **api_key** (*Optional[str]*) – The Endaq Cloud API associated with your cloud.endaq.com account. If you do not have one created yet, they can be created on the following web page: https://cloud.endaq.com/account/api-keys
- **env** (*Optional[str]*) – The cloud environment to connect to, which can be production, staging, or development. These can be easily accessed with the variables ENV_PRODUCTION, ENV_STAGING, and ENV_DEVELOP
- **test** (*bool*) – If *True* (default), the connection to enDAQ Cloud will be tested before being returned. A failed test will generate a meaningful error message describing the problem.

property account_email: Optional[str]
The email address associated with the enDAQ Cloud account.

property account_id: Optional[str]
The enDAQ Cloud account's unique ID.

get_account_info()
Get information about the connected account. Sets or updates the values of *account_id* and *account_email*.

Returns If successful, a dictionary containing (at minimum) the keys *email* and *id*.

Return type dict

get_devices(*limit=100*)
Get dataframe of devices and associated attributes (part_number, description, etc.) attached to the account.

Parameters **limit** (*int*) – The maximum number of files to return.

Returns A *DataFrame* of recorder information.

Return type pandas.core.frame.DataFrame

get_file(*file_id*, *local_name=None*)
Download the specified file to local_name if provided, use the file name from the cloud if no local name is provided. TODO: This should be made to match *endaq.ide.get_doc()*

Parameters

- **file_id** (*Union[int, str]*) – The file's cloud ID.
- **local_name** (*Optional[str]*) –

Returns The imported file, as an *idelib.Dataset*.

Return type idelib.dataset.Dataset

get_file_table(*attributes='all'*, *limit=100*)
Get a table of the data that would be similar to that you'd get doing the CSV export on the my recordings page, up to the first *limit* files with attributes matching *attributes*.

Parameters

- **limit** (*int*) – The maximum number of files to return.
- **attributes** (*Union[list, str]*) – A list of attribute strings (or a single comma-delimited string of attributes) to match.

Returns A *DataFrame* of file IDs and relevant information.

Return type pandas.core.frame.DataFrame

set_attributes(*file_id*, *attributes*)
Set the 'attributes' (name/value metadata) of a file.

Parameters

- **file_id** (*Union[int, str]*) – The file's cloud ID.
- **attributes** (*list*) – A list of dictionaries of the following structure: [{

  "name": "attr_31", "type" : "float", "value" : 3.3,

  }]

Returns The list of the file's new attributes.

Return type list

endaq.cloud.core.**count_tags**(*df*)

Given the dataframe returned by *EndaqCloud.get_file_table()*, provide some info on the tags of the files in that account.

Parameters **df** (*pandas.core.frame.DataFrame*) – A *DataFrame* of file information, as returned by *EndaqCloud.get_file_table()*.

Returns A *DataFrame* summarizing the tags in *df*.

Return type pandas.core.frame.DataFrame

endaq.cloud.core.**json_table_to_df**(*data*)

Convert JSON parsed from a custom report to a more user-friendly *pandas.DataFrame*.

Parameters **data** (*list*) – A *list* of data from a custom report's JSON.

Returns A formatted *DataFrame*

Return type pandas.core.frame.DataFrame

1.4 *endaq.plot*

1.4.1 *endaq.plot*

endaq.plot.**around_peak**(*df*, *num=1000*, *leading_ratio=0.5*)

A function to plot the data surrounding the largest peak (or valley) in the given data. The 'peak' is defined by the point in the absolute value of the given data with the largest value.

Parameters

- **df** (*pandas.core.frame.DataFrame*) – A dataframe indexed by time stamps
- **num** (*int*) – The number of points to plot
- **leading_ratio** (*float*) – The ratio of the data to be viewed that will come before the peak

Returns A Plotly figure containing the plot

endaq.plot.**gen_map**(*df_map*, *mapbox_access_token*, *filter_points_by_positive_groud_speed=True*, *color_by_column='GNSS Speed: Ground Speed'*)

Plots GPS data on a map from a single recording, shading the points based some characteristic (defaults to ground speed).

Parameters

- **df_map** (*pandas.core.frame.DataFrame*) – The pandas dataframe containing the recording data.
- **mapbox_access_token** (*str*) – The access token (or API key) needed to be able to plot against a map.
- **filter_points_by_positive_groud_speed** (*bool*) – A boolean variable, which will filter which points are plotted by if they have corresponding positive ground speeds. This helps remove points which didn't actually have a GPS location found (was created by a bug in the hardware I believe).
- **color_by_column** (*str*) – The dataframe column title to color the plotted points by.

endaq.plot.**general_get_correlation_figure**(*merged_df*, *recording_colors=None*, *hover_names=None*, *characteristics_to_show_on_hover=[]*, *starting_cols=None*)

A function to create a plot with two drop-down menus, each populated with a set of options corresponding to the scalar quantities contained in the given dataframe. The data points will then be plotted with the X and Y axis corresponding to the selected attributes from the drop-down menu.

Parameters

- **merged_df** (*pandas.core.frame.DataFrame*) – A Pandas DataFrame of data to use for producing the plot
- **recording_colors** (*Optional[collections.abc.Container]*) – The colors of each of the points to be plotted
- **hover_names** (*Optional[collections.abc.Container]*) – The names of the points to display when they are hovered on
- **characteristics_to_show_on_hover** (*list*) – The set of characteristics of the data to display when hovered over
- **starting_cols** (*Optional[collections.abc.Container]*) – The two starting columns for the dropdown menus (will be the first two available if None is given)

Returns The interactive Plotly figure

Return type plotly.graph_objs._figure.Figure

endaq.plot.**get_pure_numpy_2d_pca**(*df*, *recording_colors=None*)

Get a Plotly figure of the 2d PCA for the given DataFrame. This will have dropdown menus to select which components are being used for the X and Y axis.

Parameters

- **df** (*pandas.core.frame.DataFrame*) – The dataframe of points to compute the PCA with
- **recording_colors** (*Optional[collections.abc.Container]*) – See the same parameter in the general_get_correlation_figure function

Returns A plotly figure as described in the main function description

Return type plotly.graph_objs._figure.Figure

endaq.plot.**multi_file_plot_attributes**(*multi_file_db*, *rows_to_plot=array(['accelerationPeakFull', 'accelerationRMSFull', 'velocityRMSFull', 'psuedoVelocityPeakFull', 'displacementRMSFull', 'gpsSpeedFull', 'gyroscopeRMSFull', 'microphonoeRMSFull', 'temperatureMeanFull', 'pressureMeanFull'], dtype='<U22')*, *recording_colors=None*, *width_per_subplot=400*)

Creates a Plotly figure plotting all the desired attributes from the given DataFrame.

Parameters

- **multi_file_db** (*pandas.core.frame.DataFrame*) – The Pandas DataFrame of data to plot attributes from
- **rows_to_plot** (*numpy.ndarray*) – A numpy ndarray of strings with the names of the attributes to plot
- **recording_colors** (*Optional[collections.abc.Container]*) – The colors to make each of the points (All will be the same color if None is given)
- **width_per_subplot** (*int*) – The width to make every subplot

Returns A Plotly figure of all the subplots desired to be plotted

Return type plotly.graph_objs._figure.Figure

endaq.plot.**octave_psd_bar_plot**(*df*, *bins_per_octave=3*, *f_start=20.0*, *yaxis_title=''*, *log_scale_y_axis=True*)

Produces a bar plot of an octave psd.

Parameters

- **df** (*pandas.core.frame.DataFrame*) – The dataframe of sensor data
- **bins_per_octave** (*int*) – The number of frequency bins per octave
- **f_start** (*float*) – The center of the first frequency bin
- **yaxis_title** (*str*) – The text to label the y-axis
- **log_scale_y_axis** (*bool*) – If the y-axis should be log scaled

endaq.plot.**octave_spectrogram**(*df*, *window*, *bins_per_octave=3*, *freq_start=20.0*, *max_freq=inf*, *db_scale=True*, *log_scale_y_axis=True*)

Produces an octave spectrogram of the given data.

Parameters

- **df** (*pandas.core.frame.DataFrame*) – The dataframe of sensor data. This must only have 1 column.
- **window** (*float*) – The time window for each of the columns in the spectrogram
- **bins_per_octave** (*int*) – The number of frequency bins per octave
- **freq_start** (*float*) – The center of the first frequency bin
- **max_freq** (*float*) – The maximum frequency to plot
- **db_scale** (*bool*) – If the spectrogram should be log scaled for visibility (with 10*log10(x))
- **log_scale_y_axis** (*bool*) – If the y-axis of the plot should be log scaled

Returns a tuple containing: - the frequency bins - the time bins - the spectrogram data - the corresponding plotly figure

Return type plotly.graph_objs._figure.Figure

endaq.plot.**rolling_min_max_envelope**(*df*, *desired_num_points=250*, *plot_as_bars=False*, *plot_title=''*, *opacity=1*, *colors_to_use=None*)

A function to create a Plotly Figure to plot the data for each of the available data sub-channels, designed to reduce the number of points/data being plotted without minimizing the insight available from the plots. It will plot either an envelope for rolling windows of the data (plotting the max and the min as line plots), or a bar based plot where the top of the bar (rectangle) is the highest value in the time window that bar spans, and the bottom of the bar is the lowest point in that time window (choosing between them is done with the *plot_as_bars* parameter).

Parameters

- **df** (*pandas.core.frame.DataFrame*) – The dataframe of sub-channel data indexed by time stamps
- **desired_num_points** (*int*) – The desired number of points to be plotted for each sub-channel. The number of points will be reduced from it's original sampling rate by applying metrics (e.g. min, max) over sliding windows and then using that information to represent/visualize the data contained within the original data. If less than the desired number of points are present, then a sliding window will NOT be used, and instead the points will be plotted as they were originally recorded (also the subchannel will NOT be plotted as a bar based plot even if *plot_as_bars* was set to true).

- **plot_as_bars** (*bool*) – A boolean value indicating if the data should be visualized as a set of rectangles, where a shaded rectangle is used to represent the maximum and minimum values of the data during the time window covered by the rectangle. These maximum and minimum values are visualized by the locations of the top and bottom edges of the rectangle respectively, unless the height of the rectangle would be 0, in which case a line segment will be displayed in it's place. If this parameter is *False*, two lines will be plotted for each of the sub-channels in the figure being created, creating an 'envelope' around the data. An 'envelope' around the data consists of a line plotted for the maximum values contained in each of the time windows, and another line plotted for the minimum values. Together these lines create a boundary which contains all the data points recorded in the originally recorded data.
- **plot_title** (*str*) – The title for the plot
- **opacity** (*float*) – The opacity to use for plotting bars/lines
- **colors_to_use** (*Optional[collections.abc.Container]*) – An 'array-like' object of strings containing colors to be cycled through for the sub-channels. If None is given (which is the default), then the *colorway* variable in Plotly's current theme/template will be used to color the data on each of the sub-channels uniquely, repeating from the start of the *colorway* if all colors have been used.

Returns The Plotly Figure with the data plotted

Return type plotly.graph_objs._figure.Figure

1.4.2 *endaq.plot.dashboards*

endaq.plot.dashboards.**rolling_enveloped_dashboard**(*channel_df_dict*, *desired_num_points=250*, *num_rows=None*, *num_cols=3*, *width_for_subplot_row=400*, *height_for_subplot_row=400*, *subplot_colors=None*, *min_points_to_plot=1*, *plot_as_bars=False*, *plot_full_single_channel=False*, *opacity=1*, *y_axis_bar_plot_padding=0.06*)

A function to create a Plotly Figure with sub-plots for each of the available data sub-channels, designed to reduce the number of points/data being plotted without minimizing the insight available from the plots. It will plot either an envelope for rolling windows of the data (plotting the max and the min as line plots), or a bar based plot where the top of the bar (rectangle) is the highest value in the time window that bar spans, and the bottom of the bar is the lowest point in that time window (choosing between them is done with the *plot_as_bars* parameter).

Parameters

- **channel_df_dict** (*dict*) – A dictionary mapping channel names to Pandas DataFrames of that channels data
- **desired_num_points** (*int*) – The desired number of points to be plotted in each subplot. The number of points will be reduced from it's original sampling rate by applying metrics (e.g. min, max) over sliding windows and then using that information to represent/visualize the data contained within the original data. If less than the desired number of points are present, then a sliding window will NOT be used, and instead the points will be plotted as they were originally recorded (also the subplot will NOT be plotted as a bar based plot even if *plot_as_bars* was set to true).
- **num_rows** (*Optional[int]*) – The number of columns of subplots to be created inside the Plotly figure. If None is given, (then *num_cols* must not be None), then this number will

automatically be determined by what's needed. If more rows are specified than are needed, the number of rows will be reduced to the minimum needed to contain all the subplots

- **num_cols** (*Optional[int]*) – The number of columns of subplots to be created inside the Plotly figure. See the description of the *num_rows* parameter for more details on this parameter, and how the two interact. This also follows the same approach to handling None when given
- **width_for_subplot_row** (*int*) – The width of the area used for a single subplot (in pixels).
- **height_for_subplot_row** (*int*) – The height of the area used for a single subplot (in pixels).
- **subplot_colors** (*Optional[collections.abc.Container]*) – An 'array-like' object of strings containing colors to be cycled through for the subplots. If None is given (which is the default), then the *colorway* variable in Plotly's current theme/template will be used to color the data on each of the subplots uniquely, repeating from the start of the *colorway* if all colors have been used.
- **min_points_to_plot** (*int*) – The minimum number of data points required to be present to create a subplot for a channel/subchannel (NOT including *NaN* values).
- **plot_as_bars** (*bool*) – A boolean value indicating if the plot should be visualized as a set of rectangles, where a shaded rectangle is used to represent the maximum and minimum values of the data during the time window covered by the rectangle. These maximum and minimum values are visualized by the locations of the top and bottom edges of the rectangle respectively, unless the height of the rectangle would be 0, in which case a line segment will be displayed in it's place. If this parameter is *False*, two lines will be plotted for each of the subplots in the figure being created, creating an 'envelope' around the data. An 'envelope' around the data consists of a line plotted for the maximum values contained in each of the time windows, and another line plotted for the minimum values. Together these lines create a boundary which contains all the data points recorded in the originally recorded data.
- **plot_full_single_channel** (*bool*) – If instead of a dashboard of subplots a single plot with multiple sub-channels should be created. If this is True, only one (key, value) pair can be given for the *channel_df_dict* parameter
- **opacity** (*float*) – The opacity to use for plotting bars/lines
- **y_axis_bar_plot_padding** (*float*) – Due to some unknown reason the bar subplots aren't having their y axis ranges automatically scaled so this is the ratio of the total y-axis data range to pad both the top and bottom of the y axis with. The default value is the one it appears Plotly uses as well.

Returns The Plotly Figure containing the subplots of sensor data (the 'dashboard')

Return type plotly.graph_objs._figure.Figure

endaq.plot.dashboards.**rolling_metric_dashboard**(*channel_df_dict*, *desired_num_points=250*, *num_rows=None*, *num_cols=3*, *rolling_metrics_to_plot=('mean', 'std')*, *metric_colors=None*, *width_for_subplot_row=400*, *height_for_subplot_row=400*)

A function to create a dashboard of subplots of the given data, plotting a set of rolling metrics.

Parameters

- **channel_df_dict** (*dict*) – A dictionary mapping channel names to Pandas DataFrames of that channels data

- **desired_num_points** (*int*) – The desired number of points to be plotted in each subplot. The number of points will be reduced from it's original sampling rate by applying metrics (e.g. min, max) over sliding windows and then using that information to represent/visualize the data contained within the original data. If less than the desired number of points are present, then a sliding window will NOT be used, and instead the points will be plotted as they were originally recorded (also the subplot will NOT be plotted as a bar based plot even if *plot_as_bars* was set to true).
- **num_rows** (*Optional[int]*) – The number of columns of subplots to be created inside the Plotly figure. If None is given, (then *num_cols* must not be None), then this number will automatically be determined by what's needed. If more rows are specified than are needed, the number of rows will be reduced to the minimum needed to contain all the subplots
- **num_cols** (*Optional[int]*) – The number of columns of subplots to be created inside the Plotly figure. See the description of the *num_rows* parameter for more details on this parameter, and how the two interact. This also follows the same approach to handling None when given
- **rolling_metrics_to_plot** (*tuple*) – A tuple of strings which indicate what rolling metrics to plot for each subchannel. The options are ['mean', 'std', 'absolute max'] which correspond to the mean, standard deviation, and maximum of the absolute value.
- **metric_colors** (*Optional[collections.abc.Container]*) – An 'array-like' object of strings containing colors to be cycled through for the metrics. If None is given (which is the default), then the *colorway* variable in Plotly's current theme/template will be used to color the metric data, repeating from the start of the *colorway* if all colors have been used. The first value corresponds to the color if not enough points of data exist for a rolling metric, and the others correspond to the metric in *rolling_metrics_to_plot* in the same order they are given
- **width_for_subplot_row** (*int*) – The width of the area used for a single subplot (in pixels).
- **height_for_subplot_row** (*int*) – The height of the area used for a single subplot (in pixels).

Returns The Plotly Figure containing the subplots of sensor data (the 'dashboard')

Return type plotly.graph_objs._figure.Figure

1.4.3 *endaq.plot.utilities*

endaq.plot.utilities.**define_theme**(*template_name='endaq_cloud'*, *default_plotly_template='plotly_dark'*, *text_color='#DAD9D8'*, *font_family='Open Sans'*, *title_font_family='Open Sans SemiBold'*, *graph_line_color='#DAD9D8'*, *grid_line_color='#404041'*, *background_color='#262626'*, *plot_background_color='#0F0F0F'*)

Define a Plotly theme (template), allowing completely custom aesthetics

Parameters

- **template_name** (*str*) – The name for the Plotly template being created
- **default_plotly_template** (*str*) – The default Plotly Template (aspects of this will be used if a characteristic isn't set elsewhere)
- **text_color** (*str*) – The color of the text
- **font_family** (*str*) – The font family to use for text (not including the title)
- **title_font_family** (*str*) – The font family to use for the title

- **graph_line_color** (*str*) – The line color used when plotting line plots
- **grid_line_color** (*str*) – The color of the grid lines on the plot
- **background_color** (*str*) – The background color of the figure
- **plot_background_color** (*str*) – The background color of the plot

Returns The plotly template which was just created

Return type go.layout._template.Template

endaq.plot.utilities.**determine_plotly_map_zoom**(*lons=None*, *lats=None*, *lonlats=None*, *projection='mercator'*, *width_to_height=2.0*, *margin=1.2*)

Originally based on the following post: https://stackoverflow.com/questions/63787612/plotly-automatic-zooming-for-mapbox-maps Finds optimal zoom for a plotly mapbox. Must be passed (lons & lats) or lonlats.

Temporary solution awaiting official implementation, see: https://github.com/plotly/plotly.js/issues/3434

Parameters

- **lons** (*Optional[tuple]*) – tuple, optional, longitude component of each location
- **lats** (*Optional[tuple]*) – tuple, optional, latitude component of each location
- **lonlats** (*Optional[tuple]*) – tuple, optional, gps locations
- **projection** (*str*) – str, only accepting 'mercator' at the moment, raises *NotImplementedError* if other is passed
- **width_to_height** (*float*) – float, expected ratio of final graph's with to height, used to select the constrained axis.
- **margin** (*float*) – The desired margin around the plotted points (where 1 would be no-margin)

Returns The zoom scaling for the Plotly map

Return type float

Note: This implementation could be potentially problematic. By simply averaging min/max coorindates you end up with situations such as the longitude lines -179.99 and 179.99 being almost right next to each other, but their center is calculated at 0, the other side of the earth.

---

endaq.plot.utilities.**get_center_of_coordinates**(*lats*, *lons*, *as_list=False*, *as_degrees=True*)

Inputs and outputs are measured in degrees.

Parameters

- **lats** (*numpy.ndarray*) – An ndarray of latitude points
- **lons** (*numpy.ndarray*) – An ndarray of longitude points
- **as_list** (*bool*) – If True, return a length 2 list of the latitude and longitude coordinates. If not return a dictionary of format {"lon": lon_center, "lat": lat_center}
- **as_degrees** (*bool*) – A boolean value representing if the 'lats' and 'lons' parameters are given in degrees (as opposed to radians). These units will be used for the returned values as well.

Returns The latitude and longitude values as either a dictionary or a list, which is determined by the value of the *as_list* parameter (see the *as_list* docstring for details on the formatting of this return value

Return type Union[list, dict]

`endaq.plot.utilities.set_theme(theme='endaq')`
Sets the plot appearances based on a known 'theme'.

Parameters **theme** (*str*) – A string denoting which plot appearance color scheme to use. Current options are 'endaq', 'endaq_light', 'endaq_arial' and 'endaq_light_arial'.

Returns The plotly template which was set

Return type go.layout._template.Template

1.5 *endaq.cloud* API Wrapper

The API Wrapper provides a simple command-line interface for accessing basic file and device information from the enDAQ Cloud API. Output of all commands except `account` and *attributes* are in `csv` files in the `output` folder.

To access the cloud, this tool requires an API key associated with a user's enDAQ Cloud account, which can be provided in two ways:

- (recommended) add to the `endaq.cloud` project directory a `.env` file, formatted like so:

```
API_KEY=<Your Key>
```

- pass in an API key through the command line using the `--key` option

Warning: For security reasons, it is generally **discouraged** to make an authentication key visible on-screen or accessible through the clipboard, such as when using the `--key` option; we provide the `key` option solely as a convenience.

Runs on Python 3.6 and higher.

1.5.1 Commands

`files`	Outputs file information for selected number of files and attributes
`file-id`	Outputs file information for file with specified ID to output file
`devices`	Ouputs device information for selected number of files
`device-id`	Output device information for device with specified ID to output file
`account`	Prints out account information
`attributes`	Adds an attribute to a specified file
`set-env`	Creates a .env file with passed-in API key (*NOT SECURE*)

1.5.2 Parameters

-h	Command Line Help
--id, -i	File or Device ID
--limit, -l	File or Device output limit; Max 100 default 50
--key, -k	API Key
--attributes, -a	Attributes to be outputted; options = all or att1,att2…; default is none
--name, -n	Attribute Name
--type, -t	Attribute Type; options = int, float, string, boolean
--value, -v	Attribute Value
--verbose, -V	Prints out URL API calls

1.5.3 Usages

- `endaq-cloud set-env -k <API_KEY>`
- `endaq-cloud download -i <FILE_ID> -o <OUTPUT_FOLDER>`
- `endaq-cloud files -a <ATTRIBUTES_TO_GET> -l <FILE_OR_DEVICE_OUTPUT_LIMIT>`
- `endaq-cloud file-id -i <FILE_ID>`
- `endaq-cloud devices -l <FILE_OR_DEVICE_OUTPUT_LIMIT>`
- `endaq-cloud device-id -i <DEVICE_ID>`
- `endaq-cloud account`
- `endaq-cloud attribute -n <ATTRIBUTE_NAME> -t <ATTRIBUTE_TYPE> -v <ATTRIBUTE_VALUE> -i <FILE_ID>`

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

e

INDEX

O

R

S

T

V

W