enDAQ Documentation

Release 1.4.0

Feb 24, 2022

CONTENTS:

The endaq package is a comprehensive, user-centric Python API for working with enDAQ™ data and devices.

CHAPTER

ONE

ENDAQ API REFERENCE

1.1 endaq.ide

The submodule `endaq.ide` contains utility functions for accessing data in *Mide Instrumentation Data Exchange* (`.IDE`) files, the native format generated by enDAQ data recorders.

1.1.1 endaq.ide.measurement

The module *endaq.ide.measurement* provides an abstract representation of measurement types for easily retrieving specific data from `.ide` files. Several functions in *endaq.ide* accept combinations of `MeasurementType` constants for filtering datasets by sensor type.

Measurement types are represented by a set of singleton instances of `MeasurementType`.

```
import endaq.ide
from endaq.ide.measurement import *

doc = endaq.ide.get_doc("https://info.endaq.com/hubfs/data/surgical-instrument.ide")
endaq.ide.get_channels(doc, ACCELERATION)
endaq.ide.get_channels(doc, TEMPERATURE+PRESSURE)
```

Strings may also be used, either in combination with or instead of, the `MeasurementType` instances. The strings can be abbreviated, but no shorter than three characters.

```
endaq.ide.get_channels(doc, "acceleration")
endaq.ide.get_channels(doc, "accel")
endaq.ide.get_channels(doc, ACCELERATION+"temp")
```

endaq.ide.measurement.**ACCELERATION = <MeasurementType: Acceleration>**
Marker object for filtering channels with acceleration data

endaq.ide.measurement.**ANY = <MeasurementType: Any/all>**
Marker object for matching any/all measurement types

endaq.ide.measurement.**AUDIO = <MeasurementType: Audio>**
Marker object for filtering channels with sound level data

endaq.ide.measurement.**HUMIDITY = <MeasurementType: Relative Humidity>**
Marker object for filtering channels with (relative) humidity data

endaq.ide.measurement.**LIGHT = <MeasurementType: Light>**
Marker object for filtering channels with light intensity data

endaq.ide.measurement.**LOCATION = <MeasurementType: Location>**
Marker object for filtering channels with location data

endaq.ide.measurement.**ORIENTATION = <MeasurementType: Orientation>**
Marker object for filtering channels with rotation/orientation data

endaq.ide.measurement.**PRESSURE = <MeasurementType: Pressure>**
Marker object for filtering channels with air pressure data

endaq.ide.measurement.**ROTATION = <MeasurementType: Rotation>**
Marker object for filtering channels with angular change rate data

endaq.ide.measurement.**SPEED = <MeasurementType: Speed>**
Marker object for filtering channels with rate-of-speed data

endaq.ide.measurement.**TEMPERATURE = <MeasurementType: Temperature>**
Marker object for filtering channels with temperature data

endaq.ide.measurement.**TIME = <MeasurementType: Time>**
Marker object for filtering channels with time data

1.1.2 `endaq.ide` Usage Examples

Note: For brevity, the following examples assume everything has been imported from *endaq.ide*:

```
from endaq.ide import *
```

Opening IDE files: `endaq.ide.get_doc()`

endaq.ide includes a convenient shortcut for importing IDE data: *get_doc()*. It can load data from local files, or read data directly from a URL.

```
doc = get_doc("tests/test.ide")
doc1 = get_doc("https://info.endaq.com/hubfs/data/surgical-instrument.ide")
```

IDE files can be retrieved directly from Google Drive using a Drive 'sharable link' URL. The file must be set to allow access to "Anyone with the link."

```
doc2 = get_doc("https://drive.google.com/file/d/1t3JqbZGhuZbIK9agH24YZIdVE26-NOF5/view?
↪usp=sharing")
```

Whether opening a local file or a URL, *get_doc()* can be used to import only a specific interval by way of its `start` and `end` parameters:

```
doc3 = get_doc("tests/test.ide", start="5s", end="10s")
```

Summarizing IDE files: `endaq.ide.get_channel_table()`

Once an IDE file has been loaded, *`get_channel_table()`* will retrieve basic summary information about its contents.

```
get_channel_table(doc)
```

The results can be filtered by *measurement type*:

```
get_channel_table(doc, ACCELERATION)
```

Measurement types can be combined to retrieve more than one:

```
get_channel_table(doc, ACCELERATION+TEMPERATURE)
```

Information about a specific interval can be retrieved by using the `start` and/or `end` arguments. Note that due to different sampling rates, the start and end times for slower channels may not precisely match the specified `start` or end.

```
get_channel_table(doc, ACCELERATION+TEMPERATURE, start="0:05", end="0:10")
```

Extracting intervals: `endaq.ide.extract_time()`

A portion of an IDE file can be saved to another, new IDE. The source can be a local filename or an opened IDE (from a file or URL).

```
extract_time("tests/test.ide", "doc_extracted.ide", start="0:05", end="0:10")
extract_time(doc1, "doc1_extracted.ide", start="0:05", end="0:10")
```

Additional sample IDE recording files

Here are a number of example IDE files, which may be used with *`endaq.ide`*:

```
file_urls = ['https://info.endaq.com/hubfs/data/surgical-instrument.ide',
             'https://info.endaq.com/hubfs/data/97c3990f-Drive-Home_70-1616632444.ide',
             'https://info.endaq.com/hubfs/data/High-Drop.ide',
             'https://info.endaq.com/hubfs/data/HiTest-Shock.ide',
             'https://info.endaq.com/hubfs/data/Drive-Home_01.ide',
             'https://info.endaq.com/hubfs/data/Tower-of-Terror.ide',
             'https://info.endaq.com/hubfs/data/Punching-Bag.ide',
             'https://info.endaq.com/hubfs/data/Gun-Stock.ide',
             'https://info.endaq.com/hubfs/data/Seat-Base_21.ide',
             'https://info.endaq.com/hubfs/data/Seat-Top_09.ide',
             'https://info.endaq.com/hubfs/data/Bolted.ide',
             'https://info.endaq.com/hubfs/data/Motorcycle-Car-Crash.ide',
             'https://info.endaq.com/hubfs/data/train-passing.ide',
             'https://info.endaq.com/hubfs/data/baseball.ide',
             'https://info.endaq.com/hubfs/data/Clean-Room-VC.ide',
             'https://info.endaq.com/hubfs/data/enDAQ_Cropped.ide',
             'https://info.endaq.com/hubfs/data/Drive-Home_07.ide',
             'https://info.endaq.com/hubfs/data/ford_f150.ide',
             'https://info.endaq.com/hubfs/data/Drive-Home.ide',
```

(continues on next page)

(continued from previous page)

```
            'https://info.endaq.com/hubfs/data/Mining-Data.ide',
            'https://info.endaq.com/hubfs/data/Mide-Airport-Drive-Lexus-Hybrid-Dash-W8.
↪ide']
```

These can be directly read from `endaq.com` using *endaq.ide.get_doc()*, as previously described.

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

Efficiently extract data within a certain interval from an IDE file, writing it to another 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** (*typing.Union[idelib.dataset.Dataset, list]*) – 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** (*typing.Union[int, float, str, datetime.datetime, datetime.timedelta]*) – The starting time. Defaults to the start of the recording.
- **end** (*typing.Optional[int, float, str, datetime.datetime, datetime.timedelta]*) – The ending time. Defaults to the end of the recording.
- **formatting** (*typing.Optional[dict]*) – A dictionary of additional style/formatting items (see *pandas.DataFrame.style.format()*). If *False*, no additional formatting is applied.
- **index** (*bool*) – If *True*, show the index column on the left.
- **precision** (*int*) – The default decimal precision to display. Can be changed later.
- **timestamps** (*bool*) – 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*, *headers=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 received 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.
- **headers** – Additional URL request headers. 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 `MeasurementType` objects (one for each child) if a *Channel* was supplied.

endaq.ide.**get_primary_sensor_data**(*name=''*, *doc=None*, *measurement_type=<MeasurementType: Any/all>*, *criteria='samples'*, *time_mode='datetime'*, *least=False*, *force_data_return=False*)

Get the data from the primary sensor in a given .ide file using *to_pandas()*

Parameters

- **name** (*str*) – The file location to pull the data from, see *get_doc()* for more. This can be a local file location or a URL.
- **doc** (*Optional[idelib.dataset.Dataset]*) – An open *Dataset* object, see *get_doc()* for more. If one is provided it will not attempt to use *name* to load a new one.
- **measurement_type** (*Union[str, endaq.ide.measurement.MeasurementType]*) – The sensor type to return data from, see *measurement* for more. The default is *"any"*, but to get the primary accelerometer for example, set this to *"accel"*.
- **criteria** (*Literal['samples', 'rate', 'duration']*) – How to determine the "primary" sensor using the summary information provided by *get_channel_table()*:
  - *"sample"* - the number of samples, default behavior
  - *"rate"* - the sampling rate in Hz
  - *"duration"* - the duration from start to the end of data from that sensor
- **time_mode** (*Literal['seconds', 'timedelta', 'datetime']*) – 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
- **least** (*bool*) – If set to *True* it will return the channels ranked lowest by the given criteria.
- **force_data_return** (*bool*) – If set to *True* and the specified *measurement_type* is not included in the file, it will return the data from any sensor instead of raising an error which is the default behavior.

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

Return type pandas.core.frame.DataFrame

Here are some examples:

```
#Get sensor with the most samples, may return data of mixed units
data = get_primary_sensor_data('https://info.endaq.com/hubfs/data/All-Channels.ide')
```

(continues on next page)

(continued from previous page)

```
#Instead get just the primary accelerometer data defined by number of samples
accel = get_primary_sensor_data('https://info.endaq.com/hubfs/data/All-Channels.ide
↪', measurement_type='accel')
```

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** (*Literal['seconds', 'timedelta', 'datetime']*) – 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

endaq.calc is a module comprising a collection of common calculations for vibration analysis. It leverages the standard Python scientific stack (NumPy, SciPy, Pandas) in order to enable engineers to perform domain-specific calculations in a few lines of code, without having to first master Python and its scientific stack in their entireties.

1.2.1 endaq.calc Usage Examples

Filters

```
import plotly.express as px

import endaq
endaq.plot.utilities.set_theme('endaq_light')


# get accelerometer data from https://info.endaq.com/hubfs/100Hz_shake_cal.ide
df_accel = endaq.ide.to_pandas(endaq.ide.get_doc(
        'https://info.endaq.com/hubfs/100Hz_shake_cal.ide').channels[8].subchannels[2],
        time_mode='seconds',
    )

# create a new dataframe filtered with a high-pass butterworth with a cutoff frequency␣
↪of 1 Hz
df_accel_highpass = endaq.calc.filters.butterworth(df_accel, low_cutoff=1, high_
↪cutoff=None)
df_accel_highpass.columns = ['1Hz high-pass filter']
```

(continues on next page)

(continued from previous page)

```
# create a new dataframe filtered with a low-pass butterworth with a cutoff frequency of␣
↪100 Hz
df_accel_lowpass = endaq.calc.filters.butterworth(df_accel, low_cutoff=None, high_
↪cutoff=100)
df_accel_lowpass.columns = ['100Hz low-pass filter']

# merge the data into a single dataframe for plotting
df_accel = df_accel.join(df_accel_highpass, how='left')
df_accel = df_accel.join(df_accel_lowpass, how='left')

# plot everything on the same axes
fig1 = px.line(
        df_accel,
        x=df_accel.index,
        y=df_accel.columns,
        labels=
            {
                "timestamp": "time [s]",
                "value": "Acceleration [g]",
            },
    )
fig1.show()
```

Integration

```
import plotly.express as px

import endaq
endaq.plot.utilities.set_theme('endaq_light')


# get accelerometer data from https://info.endaq.com/hubfs/100Hz_shake_cal.ide and␣
↪convert from g to m/s^2
df_accel = endaq.ide.to_pandas(endaq.ide.get_doc(
        'https://info.endaq.com/hubfs/100Hz_shake_cal.ide').channels[8].subchannels[2],
        time_mode='seconds',
    )*9.81  # g to m/s^2

# generate the velocity and position by integrating after filtering out low-frequency␣
↪content below 1 Hz
dfs_integrate = endaq.calc.integrate.integrals(df_accel, n=2, highpass_cutoff=1.0, tukey_
↪percent=0.05)[1]
dfs_integrate_2 = endaq.calc.integrate.integrals(df_accel, n=2, highpass_cutoff=1.0,␣
↪tukey_percent=0.05)[2]
df_accel.columns = ['acceleration']
dfs_integrate.columns = ['velocity']
dfs_integrate_2.columns = ['displacement']

# combine dataframes for plotting and adjust the scale so everything fits well
```

(continues on next page)

(continued from previous page)

```
df_accel = df_accel.join(dfs_integrate*1e3, how='left')  # m/s -> mm/s
df_accel = df_accel.join(dfs_integrate_2*1e6, how='left')  # m -> m

# plot everything on the same axes
fig1 = px.line(
        df_accel,
        x=df_accel.index,
        y=df_accel.columns[::-1],
        labels=
            {
                "timestamp": "time [s]",
                "value": "Acceleration [m/s^2], Velocity [mm/s], Displacement [m]",
            },
    )
fig1.show()
```

PSD

Linearly & Octave Spaced

This presents some data from bearing tests explained in more detail in our blog on calculating vibration metrics.

```
import plotly.express as px
import pandas as pd
import endaq
endaq.plot.utilities.set_theme('endaq_light')

#Get Acceleration Data
bearing = pd.read_csv('https://info.endaq.com/hubfs/Plots/bearing_data.csv', index_col=0)

#Calculate PSD with 1 Hz Bin Width
psd = endaq.calc.psd.welch(bearing, bin_width=1)

#Plot PSD
fig1 = px.line(psd[10:5161]).update_layout(
    title_text='1 Hz PSD of Bearing Vibration',
    yaxis_title_text='Acceleration (g^2/Hz)',
    xaxis_title_text='Frequency (Hz)',
    xaxis_type='log',
    yaxis_type='log',
    legend_title_text='',
)
fig1.show()

#Calculate 1/3 Octave Spaced PSD
oct_psd = endaq.calc.psd.to_octave(psd, fstart=4, octave_bins=3)

#Plot Octave PSD
fig2 = px.line(oct_psd[10:5161]).update_layout(
    title_text='1/3 Octave PSD of Bearing Vibration',
    yaxis_title_text='Acceleration (g^2/Hz)',
```

(continues on next page)

(continued from previous page)

```
    xaxis_title_text='Frequency (Hz)',
    xaxis_type='log',
    yaxis_type='log',
    legend_title_text='',
)
fig2.show()
```

RMS from PSD

Calculating RMS of arbitrary frequency ranges is made possible with specifying `scaling='parseval'` in the *welch()* method and then using the *to_jagged()* method. Note that the overall RMS is the collective RMS of the individual ranges.

```
import plotly.express as px
import pandas as pd
import endaq
endaq.plot.utilities.set_theme('endaq_light')

#Get Acceleration Data
accel = pd.read_csv('https://info.endaq.com/hubfs/Plots/bearing_data.csv', index_col=0)

#Compute RMS of Time History
rms_time = np.mean(accel**2)**0.5

#Define Frequency Breakpoints
freqs = [0.1, 65, 300, 1400, 5999]
labels = [str(freqs[i]) + ' to ' + str(freqs[i+1]) for i in range(len(freqs)-1)]

#Compute RMS for Frequency Ranges Using PSD Functions
parseval = endaq.calc.psd.welch(accel, scaling='parseval', bin_width=0.5)
rms_psd_breaks = endaq.calc.psd.to_jagged(parseval, freqs, agg='sum')**0.5

#Plot Bar Chart of Frequency Ranges
rms_psd_breaks.index = pd.Series(labels, name='Range (Hz)')
fig1 = px.bar(rms_psd_breaks, barmode='group').update_layout(
    yaxis_title_text='Acceleration RMS (g)',
    legend_title_text='',
    title_text='RMS in Frequency Ranges'
)
fig1.show()

#Compare the RMS Calculation from Time Domain to One Using PSD
  #Note that the Overall RMS is the Collective RMS of the Individual Ranges
fig2 = px.bar(
    {
     'Time Domain':rms_time,
     'PSD w/ Breaks':np.sum(rms_psd_breaks**2)**0.5
    },
    barmode='group').update_layout(
    xaxis_title_text='',
    yaxis_title_text='Acceleration RMS (g)',
```

(continues on next page)

(continued from previous page)

```
    legend_title_text='',
    title_text='RMS from Time vs from PSD'
)
fig2.show()
```

Derivatives & Integrals

```
df_vel_psd = endaq.calc.psd.differentiate(df_accel_psd, n=-1)
df_jerk_psd = endaq.calc.psd.differentiate(df_accel_psd, n=1)
```

Vibration Criterion (VC) Curves

```
df_accel_vc = endaq.calc.psd.vc_curves(df_accel_psd, fstart=1, octave_bins=3)
```

Shock Analysis

This presents some data from a motorcycle crash test that is explained in more detail in our blog on shock response spectrums.

```
import plotly.express as px
import pandas as pd
import endaq
endaq.plot.utilities.set_theme('endaq_light')

#Get Acceleration Data
doc = endaq.ide.get_doc('https://info.endaq.com/hubfs/data/Motorcycle-Car-Crash.ide')
accel = endaq.ide.to_pandas(doc.channels[8], time_mode='seconds')[1137.4:1137.8]
accel = accel - accel.median()

#Calculate SRS
freqs = endaq.calc.utils.logfreqs(accel, init_freq=1, bins_per_octave=12)
srs = endaq.calc.shock.shock_spectrum(accel, freqs=freqs, damp=0.05, mode='srs')

#Plot SRS
fig1 = px.line(srs).update_layout(
    title_text='Shock Response Spectrum (SRS) of Motorcycle Crash',
    xaxis_title_text="Natural Frequency (Hz)",
    yaxis_title_text="Peak Acceleration (g)",
    legend_title_text='',
    xaxis_type="log",
    yaxis_type="log",
  )
fig1.show()

#Calculate PVSS
pvss = endaq.calc.shock.shock_spectrum(accel, freqs=freqs, damp=0.05, mode='pvss')
```

(continues on next page)

(continued from previous page)

```
#Generate Half Sine Equivalents
half_sine = endaq.calc.shock.enveloping_half_sine(pvss, damp=0.05)
half_sine_pvss = endaq.calc.shock.shock_spectrum(half_sine.to_time_series(tstart=0,
↪tstop=2), freqs=freqs, damp=0.05, mode='pvss')

#Add to PVSS DataFrame
half_sine_pvss.columns = half_sine.amplitude.astype(int).astype(str) + "g, " + np.
↪round(half_sine.duration*1000,1).astype(str) + "ms"
pvss = pd.concat([pvss,half_sine_pvss],axis=1)*9.81*39.37 #convert to in/s

#Plot PVSS
fig2 = px.line(pvss).update_layout(
    title_text='PVSS w/ Half Sine Equivalents',
    xaxis_title_text="Natural Frequency (Hz)",
    yaxis_title_text="Psuedo Velocity (in/s)",
    legend_title_text='',
    xaxis_type="log",
    yaxis_type="log",
  )
fig2.show()
```

1.2.2 endaq.calc.filters

endaq.calc.filters.**band_limited_noise**(*min_freq=0.0*, *max_freq=None*, *duration=1.0*, *sample_rate=1000.0*, *norm='peak'*)

Generate a time series with noise in a defined frequency range.

Parameters

- **min_freq** (*float*) – minimum frequency (Hz) where noise starts, default to 0
- **max_freq** (*float*) – maximum frequency (Hz) where noise ends, default to 1/2 the sample rate
- **duration** (*float*) – the duration of the time series returned, in seconds
- **sample_rate** (*float*) – sample rate (Hz) of the time series
- **norm** (*typing.Literal['rms', 'peak']*) – how to normalize the amplitude so that one of the following is equal to 1: * *"rms"* - root mean square * *"peak"* - peak value, default

Returns a dataframe of the generated time series

Return type pd.DataFrame

See also:

- stack overflow post that inspired this function and shared code we based this function on.

endaq.calc.filters.**bessel**(*df*, *low_cutoff=1.0*, *high_cutoff=None*, *half_order=3*, *tukey_percent=0.0*, *norm='mag'*)

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

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

Parameters

- **df** (*pd.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
- **tukey_percent** (*float*) – the alpha parameter of a preconditioning Tukey filter; if 0 (default), no filter is applied
- **norm** (*typing.Literal['phase', 'delay', 'mag']*) – how to normalize relative to the critical frequency: * *"phase"* - "phase-matched" case which is the default in SciPy & MATLAB * *"delay"* - the "natural" type obtained by solving Bessel polynomials * *"mag"* - gain magnitude is -3 dB at the cutoff frequency, default for this implementation to match Butterworth

Returns the filtered data

Return type pd.DataFrame

See also:

- SciPy Bessel filter design Documentation for the Bessel filter design function.
- SciPy Tukey window Documentation for the Tukey window function used in preprocessing.

endaq.calc.filters.**butterworth**(*df*, *low_cutoff=1.0*, *high_cutoff=None*, *half_order=3*, *tukey_percent=0.0*)

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
- **tukey_percent** (*float*) – the alpha parameter of a preconditioning Tukey filter; if 0 (default), no filter is applied

Returns the filtered data

Return type pandas.core.frame.DataFrame

See also:

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

endaq.calc.filters.**cheby1**(*df*, *low_cutoff=1.0*, *high_cutoff=None*, *half_order=3*, *tukey_percent=0.0*, *rp=3.0*)

Apply a lowpass and/or a highpass Chebyshev type I filter to an array.

This function uses Chebyshev type I 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
- **tukey_percent** (*float*) – the alpha parameter of a preconditioning Tukey filter; if 0 (default), no filter is applied
- **rp** (*float*) – the maximum ripple allowed in the passband, specified in decibels

Returns the filtered data

Return type pandas.core.frame.DataFrame

See also:

- SciPy Cheby1 filter design Documentation for the Chebyshev type I filter design function.
- SciPy Tukey window Documentation for the Tukey window function used in preprocessing.

endaq.calc.filters.**cheby2**(*df*, *low_cutoff=1.0*, *high_cutoff=None*, *half_order=3*, *tukey_percent=0.0*, *rs=30.0*)

Apply a lowpass and/or a highpass Chebyshev type II filter to an array.

This function uses Chebyshev type II 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
- **tukey_percent** (*float*) – the alpha parameter of a preconditioning Tukey filter; if 0 (default), no filter is applied
- **rs** (*float*) – the minimum attenuation allowed in the stopband, specified in decibels

Returns the filtered data

Return type pandas.core.frame.DataFrame

See also:

- SciPy Cheby2 filter design Documentation for the Chebyshev type II filter design function.
- SciPy Tukey window Documentation for the Tukey window function used in preprocessing.

endaq.calc.filters.**ellip**(*df*, *low_cutoff=1.0*, *high_cutoff=None*, *half_order=3*, *tukey_percent=0.0*, *rp=3.0*, *rs=30.0*)

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

This function uses Elliptic 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
- **tukey_percent** (*float*) – the alpha parameter of a preconditioning Tukey filter; if 0 (default), no filter is applied
- **rp** (*float*) – the maximum ripple allowed in the passband, specified in decibels
- **rs** (*float*) – the minimum attenuation allowed in the stopband, specified in decibels

Returns the filtered data

Return type pandas.core.frame.DataFrame

See also:

- SciPy Ellip filter design Documentation for the Elliptic filter design function.
- SciPy Tukey window Documentation for the Tukey window function used in preprocessing.

endaq.calc.filters.**rolling_mean**(*df*, *duration=5.0*)

Remove the rolling mean of an input time series dataframe

Parameters

- **df** (*pandas.core.frame.DataFrame*) – the input data
- **duration** (*float*) – the rolling window size in seconds to use - if *None* is given, the entire mean is removed

Returns a dataframe of the filtered data

Return type pandas.core.frame.DataFrame

1.2.3 endaq.calc.integrate

endaq.calc.integrate.**integrals**(*df*, *n=1*, *zero='start'*, *highpass_cutoff=None*, *tukey_percent=0.0*)

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; defaults to *1*
- **zero** (*Union[Literal['start', 'mean', 'median'], Callable]*) – the output quantity driven to zero by the integration constant; *"start"* (default) chooses an integration constant of `-output[0]`, *"mean"* chooses `-np.mean(output)` & *"median"* chooses `-np.median(output)`
- **highpass_cutoff** (*Optional[float]*) – the cutoff frequency for the initial highpass filter; this is used to remove artifacts caused by DC trends
- **tukey_percent** (*float*) – the alpha parameter of a preconditioning Tukey filter; if 0 (default), no filter is applied

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*, *zero='start'*, *highpass_cutoff=None*, *tukey_percent=0.0*)

Iterate over conditioned integrals of the given original data.

Parameters

- **df** (*pandas.core.frame.DataFrame*) – the input data
- **zero** (*Union[Literal['start', 'mean', 'median'], Callable]*) – the output quantity driven to zero by the integration constant; *"start"* (default) chooses an integration constant of `-output[0]`, *"mean"* chooses `-np.mean(output)` & *"median"* chooses `-np.median(output)`
- **highpass_cutoff** (*Optional[float]*) – the cutoff frequency of a preconditioning highpass filter; if None, no filter is applied
- **tukey_percent** (*float*) – the alpha parameter of a preconditioning Tukey filter; if 0 (default), 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.

1.2.4 endaq.calc.psd

endaq.calc.psd.**differentiate**(*df*, *n=1.0*)

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-differentiated 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=1.0*, *octave_bins=12.0*, ***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=1.0*, *octave_bins=12.0*)

Calculate Vibration Criterion (VC) curves from an acceleration 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=1.0*, *scaling=None*, ***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
- **scaling** (*Optional[Literal[None, 'density', 'spectrum', 'parseval']]*) – the scaling of the output; *"density"* & *"spectrum"* correspond to the same options in `scipy.signal.welch`; *"parseval"* will maintain the "energy" between the input & output, s.t. `welch(df, scaling="parseval").sum(axis="rows")` is roughly equal to `df.abs().pow(2).sum(axis="rows")`; *"unit"* will maintain the units and scale of the input data.
- **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.
- Parseval's Theorem - the theorem relating the RMS of a time-domain signal to that of its frequency spectrum

1.2.5 endaq.calc.fft

endaq.calc.fft.**aggregate_fft**(*df*, ***kwargs*)

Calculate the FFT using `scipy.signal.welch()` with a specified frequency spacing. The data returned is in the same units as the data input.

Parameters

- **df** – The input data
- **bin_width** – 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 of the input data in the same units as the input.

See also:

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

endaq.calc.fft.**dct**(*df*, *nfft=None*, *norm=None*, ***kwargs*)

Calculate the DCT of the data in `df`, using SciPy's DCT method from `scipy.fft.dct()`.

Parameters

- **df** (*pandas.core.frame.DataFrame*) – the input data
- **nfft** (*Optional[int]*) – Length of the transformed axis of the output. If nfft is smaller than the length of the input, the input is cropped. If it is larger, the input is padded with zeros. If n is not given, the length of the input along the axis specified by axis is used.

- **norm** (*Optional[Literal[None, 'unit', 'forward', 'ortho', 'backward']]*) – Normalization mode. Default is "unit", meaning a normalization of 2/n is applied on the forward transform, and a normalization of 1/2 is applied on the `idct`. The "unit" normalization means that the units of the FFT are the same as the units of the data put into it and that a sinusoid of amplitude A will peak with amplitude A in the frequency domain. "forward" instead applies a normalization of 1/n on the forward transforms and no normalization is applied on the `idct`. "backward" applies no normalization on the forward transform and 1/n on the backward. For `norm="ortho"`, both directions are scaled by *1/sqrt(n)*.
- **kwargs** – Further keywords passed to `scipy.fft.dct()`. Note that the nfft parameter of this function is passed to `scipy.fft.dct()` as `n`.

Returns The DCT of each channel in `df`.

Return type pandas.core.frame.DataFrame

See also:

- SciPy DCT method Documentation for the DCT function wrapped internally.

endaq.calc.fft.**dst**(*df*, *nfft=None*, *norm=None*, ***kwargs*)

Calculate the DST of the data in `df`, using SciPy's DST method from `scipy.fft.dst()`.

Parameters

- **df** (*pandas.core.frame.DataFrame*) – the input data
- **nfft** (*Optional[int]*) – Length of the transformed axis of the output. If nfft is smaller than the length of the input, the input is cropped. If it is larger, the input is padded with zeros. If n is not given, the length of the input along the axis specified by axis is used.
- **norm** (*Optional[Literal[None, 'unit', 'forward', 'ortho', 'backward']]*) – Normalization mode. Default is "unit", meaning a normalization of 2/n is applied on the forward transform, and a normalization of 1/2 is applied on the `idst`. The "unit" normalization means that the units of the FFT are the same as the units of the data put into it and that a sinusoid of amplitude A will peak with amplitude A in the frequency domain. "forward" instead applies a normalization of 1/n on the forward transforms and no normalization is applied on the `idst`. "backward" applies no normalization on the forward transform and 1/n on the backward. For `norm="ortho"`, both directions are scaled by *1/sqrt(n)*.
- **kwargs** – Further keywords passed to `scipy.fft.dst()`. Note that the nfft parameter of this function is passed to `scipy.fft.dst()` as `n`.

Returns The DST of each channel in `df`.

Return type pandas.core.frame.DataFrame

See also:

- SciPy DST method Documentation for the DST function wrapped internally.

endaq.calc.fft.**fft**(*df*, *output=None*, *nfft=None*, *norm=None*, *optimize=True*)

Perform the FFT of the data in `df`, using SciPy's FFT method from `scipy.fft.fft()`. If the in `df` is all real, then the output will be symmetrical between positive and negative frequencies, and it is instead recommended that you use the *endaq.calc.fft.fft()* method.

Parameters

- **df** (*pandas.core.frame.DataFrame*) – The input data

- **output** (*Optional[Literal[None, 'magnitude', 'angle', 'complex']]*) – The type of the output of the FFT. Default is "magnitude". "magnitude" will return the absolute value of the FFT, "angle" will return the phase angle in radians, "complex" will return the complex values of the FFT.
- **nfft** (*Optional[int]*) – Length of the transformed axis of the output. If nfft is smaller than the length of the input, the input is cropped. If it is larger, the input is padded with zeros. If n is not given, the length of the input along the axis specified by axis is used.
- **norm** (*Optional[Literal[None, 'unit', 'forward', 'ortho', 'backward']]*) – Normalization mode. Default is "unit", meaning a normalization of 2/n is applied on the forward transform, and a normalization of 1/2 is applied on the `ifft`. The "unit" normalization means that the units of the FFT are the same as the units of the data put into it and that a sinusoid of amplitude A will peak with amplitude A in the frequency domain. "forward" instead applies a normalization of 1/n on the forward transforms and no normalization is applied on the `ifft`. "backward" applies no normalization on the forward transform and 1/n on the backward. For `norm="ortho"`, both directions are scaled by *1/sqrt(n)*.
- **optimize** (*bool*) – If optimize is set to True, the length of the FFT will automatically be padded to a length which can be calculated more quickly. Default is True.
- **kwargs** – Further keywords passed to `scipy.fft.fft()`. Note that the nfft parameter of this function is passed to `scipy.fft.fft()` as `n`.

Returns The FFT of each channel in `df`.

Return type pandas.core.frame.DataFrame

See also:

- SciPy FFT method Documentation for the FFT function wrapped internally.

endaq.calc.fft.**rfft**(*df*, *output=None*, *nfft=None*, *norm=None*, *optimize=True*)

Perform the real valued FFT of the data in `df`, using SciPy's RFFT method from `scipy.fft.rfft()`.

Parameters

- **df** (*pandas.core.frame.DataFrame*) – The input data
- **output** (*Optional[Literal[None, 'magnitude', 'angle', 'complex']]*) – The type of the output of the FFT. Default is "magnitude". "magnitude" will return the absolute value of the FFT, "angle" will return the phase angle in radians, "complex" will return the complex values of the FFT.
- **nfft** (*Optional[int]*) – Length of the transformed axis of the output. If nfft is smaller than the length of the input, the input is cropped. If it is larger, the input is padded with zeros. If n is not given, the length of the input along the axis specified by axis is used.
- **norm** (*Optional[Literal[None, 'unit', 'forward', 'ortho', 'backward']]*) – Normalization mode. Default is "unit", meaning a normalization of 2/n is applied on the forward transform, and a normalization of 1/2 is applied on the `ifft`. The "unit" normalization means that the units of the FFT are the same as the units of the data put into it and that a sinusoid of amplitude A will peak with amplitude A in the frequency domain. "forward" instead applies a normalization of 1/n on the forward transforms and no normalization is applied on the `ifft`. "backward" applies no normalization on the forward transform and 1/n on the backward. For `norm="ortho"`, both directions are scaled by *1/sqrt(n)*.
- **optimize** (*bool*) – If optimize is set to True, the length of the FFT will automatically be padded to a length which can be calculated more quickly. Default is True.

- **kwargs** – Further keywords passed to `scipy.fft.rfft()`. Note that the nfft parameter of this function is passed to `scipy.fft.rfft()` as `n`.

Returns The RFFT of each channel in `df`.

Return type pandas.core.frame.DataFrame

See also:

- SciPy RFFT method Documentation for the RFFT function wrapped internally.

1.2.6 endaq.calc.shock

class endaq.calc.shock.**HalfSineWavePulse**(*amplitude*, *duration*)

The output data type for *enveloping_half_sine()*.

The significant data members are *amplitude* and *duration*, which can simply be unpacked as if from a plain tuple:

```
ampl, T = enveloping_half_sine(df_pvss)
```

However, users can also elect to use the other methods of this class to generate other kinds of outputs.

Note: This class is not intended to be instantiated manually.

to_time_series(*tstart=None*, *tstop=None*, *dt=None*, *tpulse=None*)

Generate a time-series of the half-sine pulse.

Parameters

- **tstart** (*Optional[float]*) – the starting time of the resulting waveform; if *None* (default), the range starts at *tpulse*
- **tstop** (*Optional[float]*) – the ending time of the resulting waveform; if *None* (default), the range ends at *tpulse* + *duration*
- **dt** (*Optional[float]*) – the sampling period of the resulting waveform; defaults to 1/20th of the pulse duration
- **tpulse** (*Optional[float]*) – the starting time of the pulse within the resulting waveform; if *None* (default), the pulse starts at either:
  - `tstart`, if provided
  - `tstop - self.duration.max()`), if *tstop* is provided
  - `0.0` otherwise

Returns a time-series of the half-sine pulse

Return type pandas.core.frame.DataFrame

endaq.calc.shock.**absolute_acceleration**(*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.
- *ISO 18431-4 Mechanical vibration and shock — Signal processing — Part 4: Shock-response spectrum analysis* Explicit implementations of digital filter coefficients for shock spectra.

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 *endaq.calc.shock.HalfSineWavePulse*

See also:

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

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

Calculate the pseudo-velocity for a SDOF system.

The pseudo-velocity follows the transfer function:

$H(s) = L\{z(t)\}(s) / L\{y''(t)\}(s) = (/s^2)(Z(s)/Y(s))$

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 pseudo-velocity 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.
- *ISO 18431-4 Mechanical vibration and shock — Signal processing — Part 4: Shock-response spectrum analysis* Explicit implementations of digital filter coefficients for shock spectra.

endaq.calc.shock.**relative_displacement**(*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.
- *ISO 18431-4 Mechanical vibration and shock — Signal processing — Part 4: Shock-response spectrum analysis* Explicit implementations of digital filter coefficients for shock spectra.

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

Calculate the relative displacement expressed as equivalent static acceleration for a SDOF system.

The relative displacement as static acceleration follows the transfer function:

$H(s) = L\{^2z(t)\}(s) / L\{y''(t)\}(s) = (^2/s^2)(Z(s)/Y(s))$

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 of the SDOF system expressed as equivalent static acceleration

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.
- *ISO 18431-4 Mechanical vibration and shock — Signal processing — Part 4: Shock-response spectrum analysis* Explicit implementations of digital filter coefficients for shock spectra.

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

Calculate the relative velocity for a SDOF system.

The relative velocity follows the transfer function:

$H(s) = L\{z'(t)\}(s) / L\{y''(t)\}(s) = (1/s)(Z(s)/Y(s))$

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 velocity 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.
- *ISO 18431-4 Mechanical vibration and shock — Signal processing — Part 4: Shock-response spectrum analysis* Explicit implementations of digital filter coefficients for shock spectra.

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.

1.2.7 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* (default), 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 of 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* (default), 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* (default), 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 - official documentation for `df.rolling().mean`
- Pandas Rolling Standard Deviation method - similar to this function, but first removes the windowed mean before squaring

1.2.8 endaq.calc.utils

endaq.calc.utils.**logfreqs**(*df*, *init_freq=None*, *bins_per_octave=12.0*)
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 uniformly spaced

Returns The resampled data in a DataFrame

Return type pandas.core.frame.DataFrame

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

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

Parameters

- **data** (*Union[numpy.ndarray, pandas.core.frame.DataFrame]*) – the input data; either a pandas DataFrame with the samples spaced along its index, or a 1D-array-like of sample times
- **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., $dB = 20 \log_{10}(...)$). 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*

See also:

- `endaq.calc.stats.rms`
- `endaq.calc.stats.rolling_rms`
- `endaq.calc.psd.welch`

1.3 endaq.cloud

This module houses tools and functions to make both interacting with the enDAQ Cloud API and the enDAQ Cloud itself easier.

1.3.1 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.

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`	Output 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*)

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

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>`

1.3.2 *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.

download_all_ide_files(*output_directory=''*, *should_download_file_fn=None*, *force_reload_file_table=False*, *file_limit=100*)
Downloads all IDE files from the enDAQ Cloud (up to a specified file limit).

Parameters

- **output_directory** (*Union[str, pathlib.Path]*) – The directory to download the ide files to
- **should_download_file_fn** (*Optional[Callable]*) – A function which accepts a row of the IDE file table and returns a boolean value which indicates if the IDE file should be downloaded or not. If this function is not given, the default function will always return True.
- **force_reload_file_table** (*bool*) – If the file table to use as reference for what files exist in the cloud should be recomputed even if it is already stored (as *self.file_table*)
- **file_limit** (*int*) – The maximum number of files to download. If the *force_recompute_file_table* parameter is True then this will also be used to limit the number of files put in the file table it creates.

Returns An array of the filenames which were just downloaded

Return type numpy.ndarray

TO-DO:

- Would be nice to have a parameter to get only ones with a certain tag
- Maybe Have a blacklist and/or whitelist parameter

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.

Parameters

- **file_id** (*Union[int, str]*) – The file's cloud ID.
- **local_name** (*Optional[str]*) – The downloaded file's destination pathname; defaults to the file's original basename & located in the directory in which the Python interpreter was launched

Returns The imported file, as an `idelib.dataset.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 Usage Examples

For these examples we assume there is a Pandas DataFrame named `df` which has it's index as time stamps and it's one column being sensor values (e.g. x-axis acceleration, or pressure). It also assumes there is a Pandas DataFrame `attribute_df` which contains all the attribute data about various data files. More information can be found about how to get this data from enDAQ IDE files in the endaq-cloud readme.

```
from endaq.plot import octave_spectrogram, multi_file_plot_attributes, octave_psd_bar_
↪plot
from endaq.plot.utilities import set_theme
```

Setting The Aesthetic Theme

```
set_theme(theme='endaq')
```

Creating Spectrograms With Octave Spaced Frequencies

```
data_df, fig = octave_spectrogram(df, window=.15)
fig.show()
```

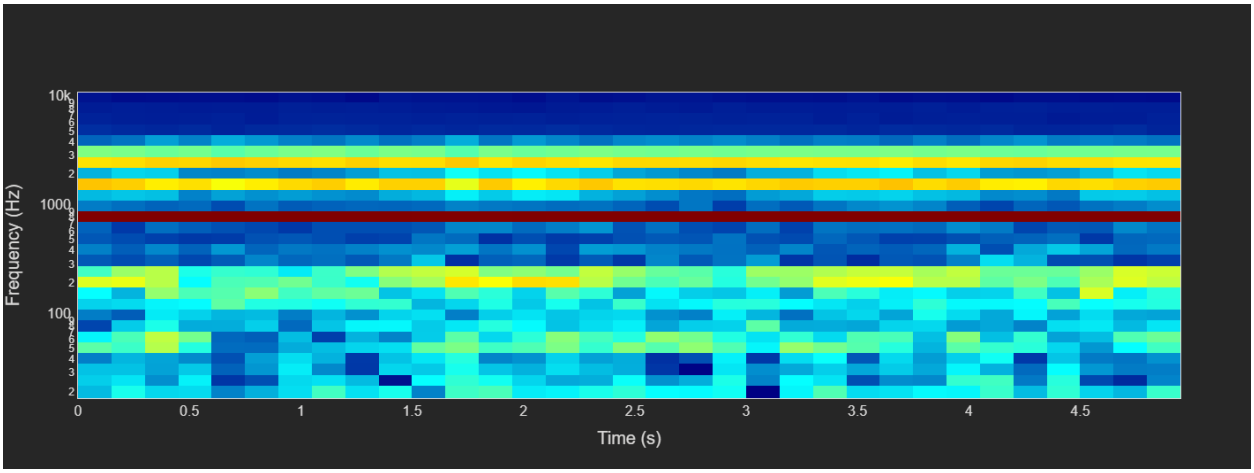


Fig. 1: Spectrogram With Octave Spaced Frequencies

Creating PSD Bar Plots With Octave Spaced Frequencies

```
fig = octave_psd_bar_plot(df, yaxis_title="Magnitude")
fig.show()
```

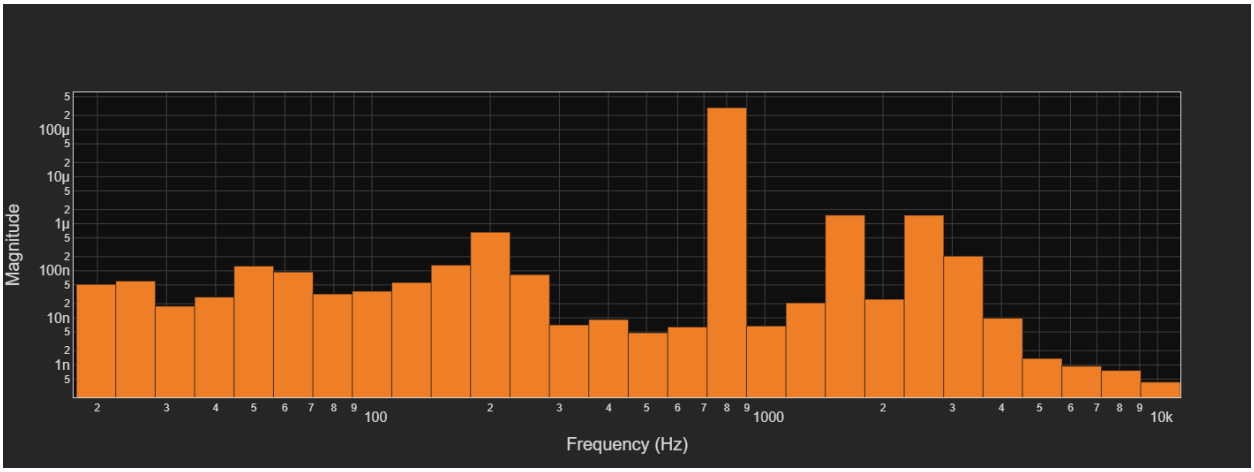


Fig. 2: PSD Bar Plot With Octave Spaced Frequencies

Plot Attributes In Figure With Subplots

```
fig = multi_file_plot_attributes(attribute_df)
fig.show()
```

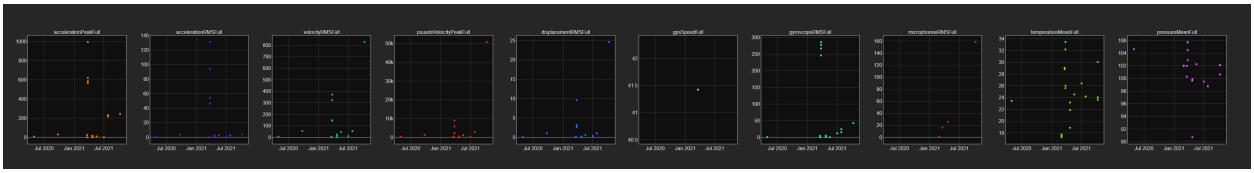


Fig. 3: Attributes Plotted As Subplots

Other Links

- the endaq package - https://github.com/MideTechnology/endaq-python
- the enDAQ homepage - https://endaq.com/

1.4.2 *endaq.plot*

endaq.plot.**animate_quaternion**(*df*, *rate=6.0*, *scale=1.0*)

A function to animate orientation as a set of axis markers derived from quaternion data.

Parameters

- **df** – A dataframe of quaternion data indexed by seconds. Columns must be 'X', 'Y', 'Z', and 'W'. The order of the columns does not matter.
- **rate** – The number of frames per second to animate.
- **scale** – Time-scaling to adjust how quickly data is parsed. Higher speeds will make the file run faster, lower speeds will make the file run slower.

Returns A Plotly figure containing the 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*, *color_col='serial_number_id'*, *color_discrete_map=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
- **color_col** (*str*) – The column name in the given dataframe (as `merged_df`) that is used to color data points with. This is used in combination with the color_discrete_map parameter
- **color_discrete_map** (*Optional[dict]*) – A dictionary which maps the values given to color data points based on (see the color_col parameter description) to the colors that these data points should be
- **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*, *color_col='serial_number_id'*, *color_discrete_map=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
- **color_col** (*str*) – The column name in the given dataframe (as `df`) that is used to color data points with. This is used in combination with the color_discrete_map parameter
- **color_discrete_map** (*Optional[dict]*) – A dictionary which maps the values given to color data points based on (see the color_col parameter description) to the colors that these data points should be

Returns A plotly figure as described in the main function description

Return type plotly.graph_objs._figure.Figure

endaq.plot.**get_tsne_plot**(*df*, *perplexity=20*, *random_seed=0*, *num_iterations=1000*, *learning_rate=500*, *color_col='serial_number_id'*, *color_discrete_map=None*, *momentum=0*)

Estimates a SNE model, and plots the visualization of the given data.

More information about t-SNE visualizations and how to use them effectively can be found here: https://distill.pub/2016/misread-tsne/

(Implementation based around the following: https://nlml.github.io/in-raw-numpy/in-raw-numpy-t-sne/)

Parameters

- **df** (*pandas.core.frame.DataFrame*) – A Pandas Dataframe of the the data to be visualized (e.g. a recording attribute DataFrame)
- **perplexity** (*int*) – A value which can be thought of as determining how many neighbors of a point will be used to update its position in the visualization.
- **random_seed** (*int*) – Integer value to set the seed value for the random
- **num_iterations** (*int*) – The number of iterations to train for

- **learning_rate** (*int*) – The rate at which to update the values in the model
- **color_col** (*str*) – The column name in the given dataframe (as `merged_df`) that is used to color data points with. This is used in combination with the color_discrete_map parameter
- **color_discrete_map** (*Optional[dict]*) – A dictionary which maps the values given to color data points based on (see the color_col parameter description) to the colors that these data points should be
- **momentum** (*int*) – The momentum to be used when applying updates to the model

Returns Plotly figure of the, low-dimensional representation of the given data

Return type plotly.graph_objs._figure.Figure

endaq.plot.**multi_file_plot_attributes**(*multi_file_db*, *attribs_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
- **attribs_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=1.0*, *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

1.4.3 *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=1.0*, *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 its 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 its 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 its 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', 'rms']` which correspond to the mean, standard deviation, maximum of the absolute value, and root-mean-square.
- **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

1.4.4 *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 plotly.graph_objs.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*)

Finds optimal zoom for a plotly mapbox. Must be passed (`lons` & `lats`) or `lonlats`.

Originally based on the following post: https://stackoverflow.com/questions/63787612/plotly-automatic-zooming-for-mapbox-maps

This is a temporary solution awaiting an official implementation: 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 coordinates 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'*)

Set the plot appearances based on a known 'theme'.

Parameters **theme** (*Literal['endaq', 'endaq_light', 'endaq_arial', 'endaq_light_arial']*) – 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 plotly.graph_objs.layout._template.Template

1.5 endaq.batch

class endaq.batch.**GetDataBuilder**(*, *preferred_chs=[]*, *accel_highpass_cutoff*, *accel_start_time=None*, *accel_end_time=None*, *accel_start_margin=None*, *accel_end_margin=None*, *accel_integral_tukey_percent=0*, *accel_integral_zero='start'*)

The main interface for calculations in `endaq.batch`.

This object has two types of functions:

- *configuration functions* - these determine what calculations will be performed on IDE recordings, and pass in any requisite parameters for said calculations. This includes the following functions:

  aggedright

  - *add_psd()*
  - *add_pvss()*
  - *add_pvss_halfsine_envelope()*
  - *add_metrics()*
  - *add_peaks()*
  - *add_vc_curves()*

- *execution functions* - these functions take recording files as parameters, perform the configured calculations on the data therein, and return the calculated data as a *OutputStruct* object that wraps pandas objects.

  This includes the functions `_get_data()` & *aggregate_data()*, which operates on one & multiple file(s), respectively.

A typical use case will look something like this:

```
filenames = [...]

calc_output = (
    GetDataBuilder(accel_highpass_cutoff=1)
    .add_psd(freq_bin_width=1)
    .add_pvss(init_freq=1, bins_per_octave=12)
    .add_pvss_halfsine_envelope()
    .add_metrics()
    .add_peaks(margin_len=100)
    .add_vc_curves(init_freq=1, bins_per_octave=3)
    .aggregate_data(filenames)
)
file_data = calc_output.dataframes
```

Parameters

- **preferred_chs** – a sequence of channels; each channel listed is prioritized over others of the same type of physical measurement (e.g., acceleration, temperature, pressure, etc.)
- **accel_highpass_cutoff** – the cutoff frequency used when pre-filtering acceleration data
- **accel_start_time** – the relative timestamp before which to reject recording data; cannot be used in conjunction with *accel_start_margin*
- **accel_end_time** – the relative timestamp after which to reject recording data; cannot be used in conjunction with *accel_end_margin*
- **accel_start_margin** – the number of samples before which to reject recording data; cannot be used in conjunction with *accel_start_time*
- **accel_end_margin** – the number of samples after which to reject recording data; cannot be used in conjunction with *accel_end_time*

- **accel_integral_tukey_percent** – the alpha parameter of a Tukey window applied to the acceleration before integrating into velocity & displacement; see the *tukey_percent* parameter in *endaq.calc.integrate.integrals()* for details
- **accel_integral_zero** – the output quantity driven to zero when integrating the acceleration into velocity & displacement; see the *zero* parameter in *endaq.calc.integrate.integrals()* for details

add_metrics(*include=[]*, *exclude=[]*)

Add broad channel metrics to the calculation queue.

The output units for each metric are listed below:

aggedright

- *RMS Acceleration*: G
- *RMS Velocity*: $\frac{\text{mm}}{\text{sec}}$
- *RMS Displacement*: mm
- *Peak Absolute Acceleration*: G
- *Peak Pseudo Velocity Shock Spectrum*: $\frac{\text{mm}}{\text{sec}}$
- *GPS Position*: degrees
- *GPS Speed*: $\frac{\text{km}}{\text{hr}}$
- *RMS Angular Velocity*: $\frac{\text{degrees}}{\text{sec}}$
- *RMS Microphone*: Pascals
- *Average Temperature*: °C
- *Average Pressure*: Pascals
- *Average Relative Humidity*:

where *G* is the acceleration of gravity $\left(1\text{G} \approx 9.80665 \frac{\text{m}}{\text{sec}^2}\right)$

add_peaks(*margin_len=1000*)

Add windows about the acceleration's peak value to the calculation queue.

calculation output units: G, where *G* is the acceleration of gravity $\left(1\text{G} \approx 9.80665 \frac{\text{m}}{\text{sec}^2}\right)$

Parameters **margin_len** (*int*) – the number of samples on each side of a peak to include in the windows

add_psd(*freq_bin_width=None*, *freq_start_octave=None*, *bins_per_octave=None*, *window=None*)

Add the acceleration PSD to the calculation queue.

calculation output units: $\frac{\text{G}^2}{\text{Hz}}$, where *G* is the acceleration of gravity $\left(1\text{G} \approx 9.80665 \frac{\text{m}}{\text{sec}^2}\right)$

Parameters

- **freq_bin_width** (*Optional[float]*) – the desired spacing between adjacent PSD samples; a default is provided only if *bins_per_octave* is used, otherwise this parameter is required
- **freq_start_octave** (*Optional[float]*) – the first frequency to use in octave-spacing; this is only used if *bins_per_octave* is set
- **bins_per_octave** (*Optional[float]*) – the number of frequency bins per octave in a log-spaced PSD; if not set, the PSD will be linearly-spaced as specified by *freq_bin_width*
- **window** (*Optional[str]*) – the window type used in the PSD calculation; see the documentation for `scipy.signal.welch` for details

add_pvss(*init_freq=1.0*, *bins_per_octave=3.0*)

Add the acceleration PVSS (Pseudo Velocity Shock Spectrum) to the calculation queue.

calculation output units: $\frac{\text{mm}}{\text{sec}}$

Parameters

- **init_freq** (*float*) – the first frequency sample in the spectrum

- **bins_per_octave** (*float*) – the number of samples per frequency octave

add_pvss_halfsine_envelope(*tstart=None*, *tstop=None*, *dt=None*, *tpulse=None*)
Add the half-sine envelope for the acceleration's PVSS (Pseudo Velocity Shock Spectrum) to the calculation queue.

calculation output units: $\frac{\text{mm}}{\text{sec}}$

add_vc_curves(*init_freq=1.0*, *bins_per_octave=3.0*)
Add Vibration Criteria (VC) Curves to the calculation queue.

calculation output units: $\frac{\text{m}}{\text{sec}}$

Parameters

- **init_freq** (*float*) – the first frequency
- **bins_per_octave** (*float*) – the number of samples per frequency octave

aggregate_data(*filenames*)
Compile configured data from the given files into a dataframe.

Parameters **filenames** – a sequence of paths of recording files to process

class endaq.batch.core.**OutputStruct**(*data*)
A data wrapper class with methods for common export operations.

Objects of this class are generated by *GetDataBuilder.aggregate_data()*. This class is not intended be instantiated manually.

dataframes

to_csv_folder(*folder_path*)
Write data to a folder as CSV's.

Parameters **folder_path** – the output directory path for .CSV files

to_html_plots(*folder_path=None*, *show=False*, *theme='endaq'*)
Generate plots in HTML.

Parameters

- **folder_path** – The output directory for saving .HTML plots. If *None* (default), plots are not saved.
- **show** (*bool*) – Whether to open plots after generation. Defaults to *False*.
- **theme** (*Literal[None, 'endaq', 'endaq_light', 'endaq_arial', 'endaq_light_arial']*) – The enDAQ plotly theme to use; see *endaq.plot.utilities.set_theme()* for details on the supported options. Defaults to *"endaq"*. If *None*, the default Plotly theme is used.

CHAPTER

TWO

EXAMPLE NOTEBOOKS

These notebooks are copied from our "Python for Mechanical Engineers" webinar series

2.1 Intro to Python Acceleration and CSV Analysis

2.1.1 Introduction

This notebook serves as an introduction to Python for a mechanical engineer looking to plot and analyze some acceleration data in a CSV file. Being a Colab, this tool can freely be used without installing anything.

For more information on making the swith to Python see enDAQ's blog, Why and How to Get Started in Python for a MATLAB User.

This is part of our webinar series on Python for Mechanical Engineers:

1. **Get Started with Python**
   - Watch Recording of This
2. Introduction to Numpy & Pandas for Data Analysis
3. Introduction to Plotly for Plotting Data
4. Introduction of the enDAQ Library

2.1.2 Import Data File

We will assume that the first column is time in seconds. Some example files are provided or you can load your own.

Example Files

Here are some example datasets you can use to do some initial testing. If you have uploaded your own data, you'll want to comment this out or not run it!

```
[ ]: filenames = ['https://info.endaq.com/hubfs/data/surgical-instrument.csv',
                  'https://info.endaq.com/hubfs/data/blushift.csv',
                  'https://info.endaq.com/hubfs/Plots/bearing_data.csv', #used in this␣
↪dataset: https://blog.endaq.com/top-vibration-metrics-to-monitor-how-to-calculate-them
                  'https://info.endaq.com/hubfs/data/Motorcycle-Car-Crash.csv', #used in this␣
↪blog: https://blog.endaq.com/shock-analysis-response-spectrum-srs-pseudo-velocity-
↪severity
```

(continues on next page)

(continued from previous page)

```
            'https://info.endaq.com/hubfs/data/Calibration-Shake.csv',
            'https://info.endaq.com/hubfs/data/Mining-Hammer.csv'] #largest dataset
filename = filenames[4]
```

[ ]:
```
filenames[4]
```

```
'https://info.endaq.com/hubfs/data/Calibration-Shake.csv'
```

2.1.3 Install & Import Libraries

First we'll install all libraries we'll need, then import them.

Note that if running this locally you'll only need to install one time, then subsequent runs can just do the import. But colab and anaconda will contain all the libraries we'll need anyways so the install isn't necessary. Here is how the install would be done though: ~~~ !pip install pandas !pip install numpy !pip install matplotlib !pip install plotly !pip install scipy ~~~

You can always check which libraries you have installed by doing: ~~~ !pip freeze ~~~ We do need to upgrade plotly though to work in Colab

[ ]:
```
!pip install --upgrade plotly
```

```
Requirement already satisfied: plotly in /usr/local/lib/python3.7/dist-packages (5.3.1)
Requirement already satisfied: six in /usr/local/lib/python3.7/dist-packages (from␣
↪plotly) (1.15.0)
Requirement already satisfied: tenacity>=6.2.0 in /usr/local/lib/python3.7/dist-packages␣
↪(from plotly) (8.0.1)
```

Now we'll import our libraries we'll use later.

[ ]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import plotly.express as xp
import plotly.io as pio; pio.renderers.default = "iframe"
from scipy import signal
```

2.1.4 Load the CSV, Analyze & Plot

We'll load the data into pandas, display it, do some very basic analysis, and plot the time history in a few ways.

Load the CSV File and Prepare

Remember we are expecting the first column to be time and will set it as the index. This is loading a CSV file, but Pandas supports a lot of other file formats, see: Pandas Input/Output.

If you must/need to use .MAT files, scipy can read these: scipy.io.loadmat

[ ]:
```
df = pd.read_csv(filename) #load the data
df = df.set_index(df.columns[0]) #set the first column as the index
df
```

```
            "X (2000g)"   "Y (2000g)"   "Z (2000g)"
Time
0.004394      -0.122072     -0.122072     -0.061036
0.004594      -0.061036      0.488289     -0.366217
0.004794       0.183108      0.122072     -0.061036
0.004994       0.122072     -0.122072     -0.122072
0.005194       0.122072      0.122072     -0.244144
...                 ...           ...           ...
27.691064     -0.427253     -0.152590     -0.671397
27.691264     -0.122072     -0.335698     -0.305180
27.691464     -0.183108     -0.152590     -0.122072
27.691664     -0.305180      0.030518     -0.244144
27.691864     -0.305180     -0.030518     -0.366217

[138440 rows x 3 columns]
```

Basic Analysis

Once in a Pandas dataframe, doing some basic analysis is SUPER easy as shown. Here's a link to the docs for the .max() function, but notice the many others readily available.

The peak will be applied after first finding the absolute value to ensure we don't ignore large negative values.

$$peak = max(|a|)$$

Then the RMS is a simple square root of the mean of all the values squared.

$$rms = \sqrt{(\frac{1}{n})\sum_{i=1}^{n}(a_i)^2}$$

The crest factor is equal to the peak divided by the RMS.

$$crest = \frac{peak}{rms}$$

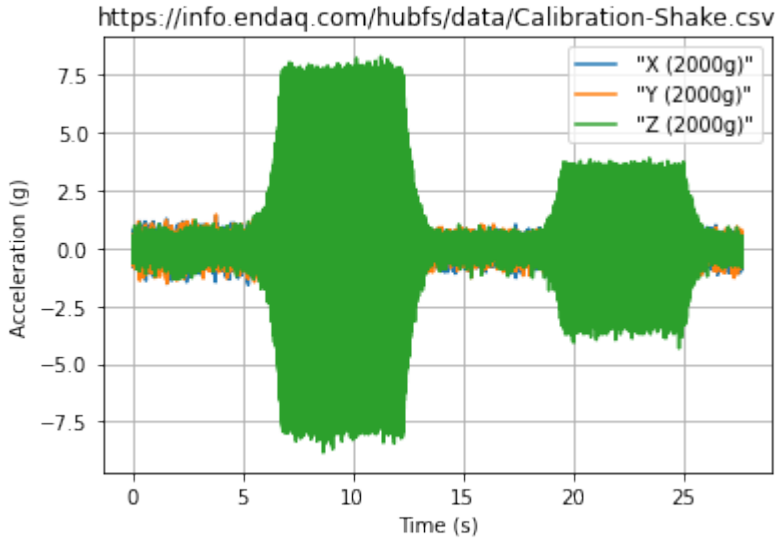
[ ]:
```
abs_max = df.abs().max() #maximum of the absolute values, the peak
std = df.std() #standard deviation (equivalent to the AC coupled RMS value)
crest_factor = abs_max/std #crest factor (peak / RMS)

stats = pd.concat([abs_max, #combine the stats into one table
                   std,
                   crest_factor],
                  axis=1)
stats.columns = ['Peak Acceleration (g)','RMS (g)','Crest Factor'] #set the headers of␣
↪the table
stats
```

```
             Peak Acceleration (g)   RMS (g)  Crest Factor
 "X (2000g)"               1.709010  0.249129      6.859939
 "Y (2000g)"               1.647974  0.279338      5.899566
 "Z (2000g)"               8.850233  2.687501      3.293109
```

2.1.5 Plot Full Time Series

You can create a plot very simply once you have a dataframe by just doing: ~~df.plot()~~ Here we show how to manipulate this plot a bit with axes labels in Matplotlib which has a very similar interface to MATLAB. There are a lot of pretty well documented examples on matplotlib's docs site (but their docs are confusing to navigate).

[ ]:
```
fig, ax = plt.subplots() #create an empty plot

df.plot(ax=ax) #use the dataframe to add plot data, tell it to add to the already␣
↪created axes

ax.set(xlabel='Time (s)',
       ylabel='Acceleration (g)',
       title=filename)
ax.grid() #turn on gridlines

fig.savefig('full-time-history.png')
plt.show()
```



2.1.6 Plot with Plotly

Matplotlib may be familiar, but Plotly offers much more interactivity directly in a browser. They also have really good documentation with a ton of examples online.

The trouble is that plotting too many data points may be sluggish. I've long maintained that plotting 10s of thousands of data points isn't very useful anyways, you just get a shaded mess.

So here we'll plot: - The moving peak value - The moving RMS - The time history around the peak

Moving Peak

This takes advantage of Pandas rolling() function.

```
[ ]: n_steps = 100 #number of points to plot
n = int(df.shape[0]/n_steps) #number of data points to use in windowing
df_rolling_peak = df.abs().rolling(n).max().iloc[::n] #finds the absolute value of every␣
↪datapoint, then does a rolling maximum of the defined window size, then subsamples␣
↪every nth point

df_rolling_peak
```

```
            "X (2000g)"   "Y (2000g)"   "Z (2000g)"
Time
0.004394            NaN           NaN           NaN
0.281203       0.976577      1.159686      0.976577
0.558025       1.159686      1.403830      1.068132
0.834870       1.403830      1.159686      1.129168
1.111689       1.220722      1.281758      1.220722
...                 ...           ...           ...
26.576863      1.159686      1.037613      0.793469
26.853655      0.976577      1.037613      0.854505
27.130463      1.068132      1.068132      0.946059
27.407260      0.976577      1.281758      0.793469
27.684064      1.098650      1.129168      0.915541

[101 rows x 3 columns]
```

```
[ ]: fig = xp.line(df_rolling_peak)
fig.update_layout(
    title="Rolling Peak",
    xaxis_title="Time (s)",
    yaxis_title="Acceleration (g)",
)
fig.show()
fig.write_html('rolling_peak.html',full_html=False,include_plotlyjs='cdn')
```

Moving RMS

Now we'll plot the rolling RMS using the standard deviation. Notice that these rolling value plots make it much easier to compare the datasets than by trying to plot all values which result in a shaded mess.

Also in this example I'm showing how easy it is to change the theme of the plotly figure, see their documentation for more examples and information. You can also make custom themes.

```
[ ]: df_rolling_rms = df.rolling(n).std().iloc[::n] #does a rolling standard deviation of the␣
↪defined window size, then subsamples every nth point

fig = xp.line(df_rolling_rms)
fig.update_layout(
    title="Rolling RMS",
```

(continues on next page)

(continued from previous page)

```
    xaxis_title="Time (s)",
    yaxis_title="Acceleration (g)",
    template="plotly_dark"
)
fig.show()
fig.write_html('rolling_rms.html',full_html=False,include_plotlyjs='cdn')
```

Time History Around Peak

Now let's find the time that had the maximum value and display the time history around that.

[ ]:
```
df.abs().max(axis=1).idxmax()
```

```
8.659936
```

[ ]:
```
peak_time = df.abs().max(axis=1).idxmax() #get the time at which the peak value occurs
d_t = (df.index[-1]-df.index[0])/(len(df.index)-1) #find the average time step
fs = 1/d_t #find the sampling rate

num = 1000 / 2 #total number of datapoints to plot (divide by 2 because it will be two␣
↪sided)
df_peak = df[peak_time - num / fs : peak_time + num / fs ] #segment the dataframe to be␣
↪around that peak value

fig = xp.line(df_peak)
fig.update_layout(
    title="Time History around Peak",
    xaxis_title="Time (s)",
    yaxis_title="Acceleration (g)",
    template="plotly_white"
)
fig.show()
fig.write_html('time_history_peak.html',full_html=False,include_plotlyjs='cdn')
```

2.1.7 PSD

Now using SciPy we can easily compute and plot a PSD using a custom function we'll make to ease the interface to SciPy's Welch function. This is very similar to MATLAB's version.

[ ]:
```
def get_psd(df, bin_width=1.0, window="hanning"):
    d_t = (df.index[-1]-df.index[0])/(len(df.index)-1)
    fs = 1/d_t
    f, psd = signal.welch(
        df.values, fs=fs, nperseg= int(fs / bin_width), window=window, axis=0
    )

    df_psd = pd.DataFrame(psd, columns=df.columns)
    df_psd["Frequency (Hz)"] = f
```

(continues on next page)

(continued from previous page)

```
    df_psd = df_psd.set_index("Frequency (Hz)")
    return df_psd
```

[ ]:
```
df_psd = get_psd(df,bin_width=4) #compute a PSD with a 1 Hz bin width
df_psd.to_csv('psd.csv') #save to a CSV file
df_psd
```

```
                 "X (2000g)"   "Y (2000g)"   "Z (2000g)"
Frequency (Hz)
0.000000            0.000048      0.000049      0.000072
4.000048            0.000294      0.000276      0.000332
8.000095            0.000256      0.000254      0.000287
12.000143           0.000189      0.000206      0.000230
16.000191           0.000170      0.000156      0.000193
...                      ...           ...           ...
2484.029606         0.000007      0.000006      0.000007
2488.029654         0.000007      0.000006      0.000006
2492.029702         0.000007      0.000007      0.000007
2496.029749         0.000007      0.000007      0.000007
2500.029797         0.000003      0.000004      0.000004

[626 rows x 3 columns]
```

[ ]:
```
fig = xp.line(df_psd)
fig.update_layout(
    title="Power Spectral Density (PSD)",
    xaxis_title="Frequency (Hz)",
    yaxis_title="Acceleration (g^2/Hz)",
    xaxis_type="log",
    yaxis_type="log"
)
fig.show()
fig.write_html('psd.html',full_html=False,include_plotlyjs='cdn')
```

2.1.8 Cumulative RMS from PSD

Now that we have the PSD, we can easily compute and plot the overall RMS value. This is partially thanks to the cumulative sum function in Pandas.

The nice thing about a PSD (in addition to the easy control of the bin width) is that the area directly relates to the RMS level in the time domain. The equation is as follows.

$$g_{RMS} = \sqrt{\int PSD(f), df}$$

Let's demonstrate by quickly using the PSD just calculated, integrating, and taking the square root and compare to the values we calculated from the time domain.

[ ]:
```
def rms_from_psd(df_psd):
    d_f = df_psd.index[1] - df_psd.index[0]
```

(continues on next page)

(continued from previous page)

```
    df_rms = df_psd.copy()
    df_rms = df_rms*d_f
    df_rms = df_rms.cumsum()
    return(df_rms**0.5)
```

[ ]:
```
df_rms = rms_from_psd(df_psd)

fig = xp.line(df_rms)
fig.update_layout(
    title="Cumulative RMS",
    xaxis_title="Frequency (Hz)",
    yaxis_title="Acceleration (g RMS)",
    xaxis_type="log",
    #yaxis_type="log"
)
fig.show()
fig.write_html('cum_rms.html',full_html=False,include_plotlyjs='cdn')
```

2.1.9 FFT

Typical FFT (Or Should We Say DFT)

This uses SciPy's discrete Fourier transform function. The trouble here is that this may be very long and therefore plotting a LOT of data.

[ ]:
```
from scipy.fft import fft, fftfreq

def get_fft(df):
    N=len(df)
    fs = len(df)/(df.index[-1]-df.index[0])

    x_plot= fftfreq(N, 1/fs)[:N//2]

    df_fft = pd.DataFrame()
    df_phase = pd.DataFrame()
    for name in df.columns:
        yf = fft(df[name].values)
        y_plot= 2.0/N * np.abs(yf[0:N//2])

        phase = np.unwrap(2 * np.angle(yf)) / 2 * 180/np.pi
        phase = phase[0:N//2]

        df_phase = pd.concat([df_phase,
                              pd.DataFrame({'Frequency (Hz)':x_plot[1:],
                                            name:phase[1:]}).set_index('Frequency (Hz)')],
↪axis=1)
        df_fft = pd.concat([df_fft,
                              pd.DataFrame({'Frequency (Hz)':x_plot[1:],
                                            name:y_plot[1:]}).set_index('Frequency (Hz)')],
↪axis=1)
```

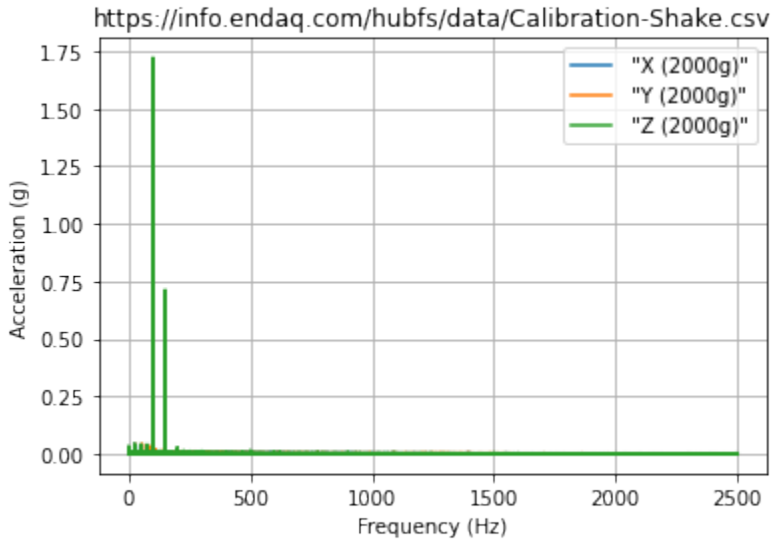
(continues on next page)

(continued from previous page)

```
    return df_fft, df_phase
```

[ ]:
```
df_fft, df_phase = get_fft(df)
```

[ ]:
```
fig, ax = plt.subplots() #create an empty plot

df_fft.plot(ax=ax) #use the dataframe to add plot data, tell it to add to the already␣
↪created axes

ax.set(xlabel='Frequency (Hz)',
       ylabel='Acceleration (g)',
       title=filename)
ax.grid() #turn on gridlines

fig.savefig('fft.png')
plt.show()
```



FFT from PSD

Here we can use the output of a PSD and convet it to a typical DFT. This has the benefit of allowing you to explicitely define the frequency bin width.

[ ]:
```
df
```

```
            "X (2000g)"   "Y (2000g)"   "Z (2000g)"
Time
0.004394      -0.122072     -0.122072     -0.061036
0.004594      -0.061036      0.488289     -0.366217
0.004794       0.183108      0.122072     -0.061036
0.004994       0.122072     -0.122072     -0.122072
0.005194       0.122072      0.122072     -0.244144
...                 ...           ...           ...
```

(continues on next page)

(continued from previous page)

```
27.691064     -0.427253     -0.152590     -0.671397
27.691264     -0.122072     -0.335698     -0.305180
27.691464     -0.183108     -0.152590     -0.122072
27.691664     -0.305180      0.030518     -0.244144
27.691864     -0.305180     -0.030518     -0.366217

[138440 rows x 3 columns]
```

[ ]:
```
def get_fft_from_psd(df,bin_width):
  fs = len(df)/(df.index[-1]-df.index[0])
  f, psd = signal.welch(df.to_numpy(),
                        fs=fs,
                        nperseg=fs/bin_width,
                        window='hanning',
                        axis=0,
                        scaling = 'spectrum'
                        )

  df_psd = pd.DataFrame(psd**0.5,columns=df.columns)
  df_psd.columns
  df_psd['Frequency (Hz)'] = f
  return df_psd.set_index('Frequency (Hz)')
```

[ ]:
```
df_fft_from_psd = get_fft_from_psd(df,.25)

fig = xp.line(df_fft_from_psd)
fig.update_layout(
    title="FFT from PSD",
    xaxis_title="Frequency (Hz)",
    yaxis_title="Acceleration (g)",
    #xaxis_type="log",
    #yaxis_type="log"
)
fig.show()
fig.write_html('fft_from_psd.html',full_html=False,include_plotlyjs='cdn')
```

2.1.10 Moving Peak Frequency

Now we'll introduce a for loop to go through all the columns and find the moving peak frequency.

[ ]:
```
def moving_frequency(df,steps):
  #df.index = np.linspace(df.index[0],df.index[-1],num=df.shape[0]) #resample to have␣
↪uniform sampling
  d_t = (df.index[-1]-df.index[0])/(len(df.index)-1)
  fs = 1/d_t
  length = df.shape[0] #full length of time series data
  nperseg = length/5 #average 5 FFTs per segment
  n = int(length/steps) #the number of data points per step, we will not compute
  df_peak_freqs = pd.DataFrame(columns=df.columns)
  for i in range(0,length-n+1,n):
```

(continues on next page)

(continued from previous page)

```
    df_sub = df.iloc[i:i+n]
    df_psd = get_psd(df_sub, nperseg/fs)
    time = (df_sub.index[-1]-df_sub.index[0])/2+df_sub.index[0]
    df_peak_freqs.loc[time] = df_psd.idxmax()
  return df_peak_freqs
```

[ ]:
```
df_peak_freqs = moving_frequency(df,40) #get rolling peak frequency for 20 different␣
↪steps

fig = xp.line(df_peak_freqs)
fig.update_layout(
    title="Rolling Peak Frequency",
    xaxis_title="Time (s)",
    yaxis_title="Peak Frequency (Hz)",
)
fig.show()
fig.write_html('rolling_peak_frequency.html',full_html=False,include_plotlyjs='cdn')
```

[ ]:
```

```

2.2 Introduction to NumPy and Pandas

2.2.1 Introduction

This notebook and accompanying webinar was developed and released by the enDAQ team. This is the second "chapter" of our series on *Python for Mechanical Engineers*: 1. Get Started with Python * Blog: Get Started with Python: Why and How Mechanical Engineers Should Make the Switch 2. **Introduction to Numpy & Pandas** * Watch Recording of This 3. Introduction to Plotly 4. Introduction of the enDAQ Library

To sign up for future webinars and watch previous ones, visit our webinars page.

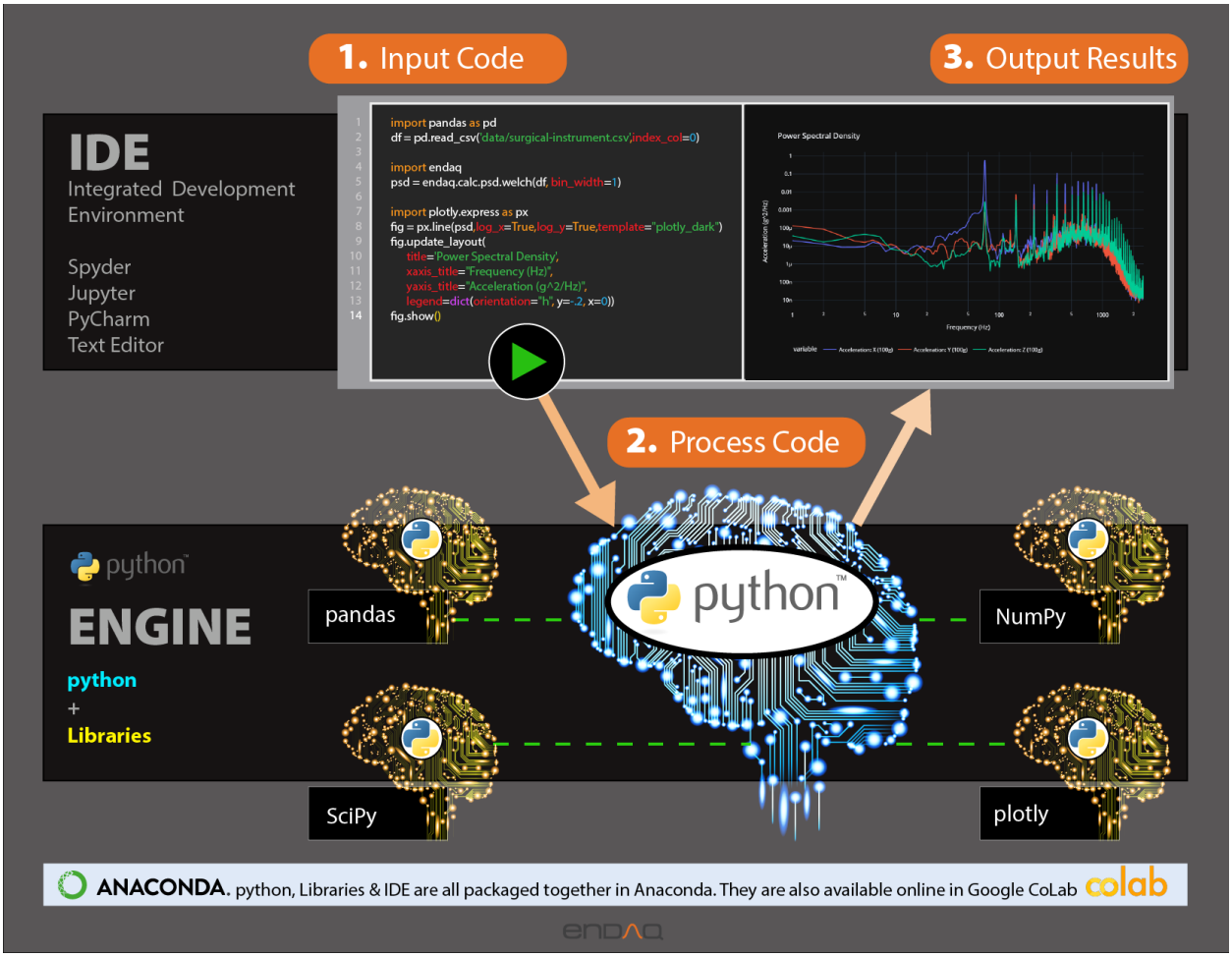
Recap of Python Introduction

1. Python is popular for good reason
2. There are many ways to interact with Python

- Here we are in Google Colab based on Jupyter Notebooks

3. There are many open source libraries to use

- Today we are covering two of the most popular:
  - Numpy
  - Pandas
- And teasing a bit of:
  - Plotly
  - enDAQ

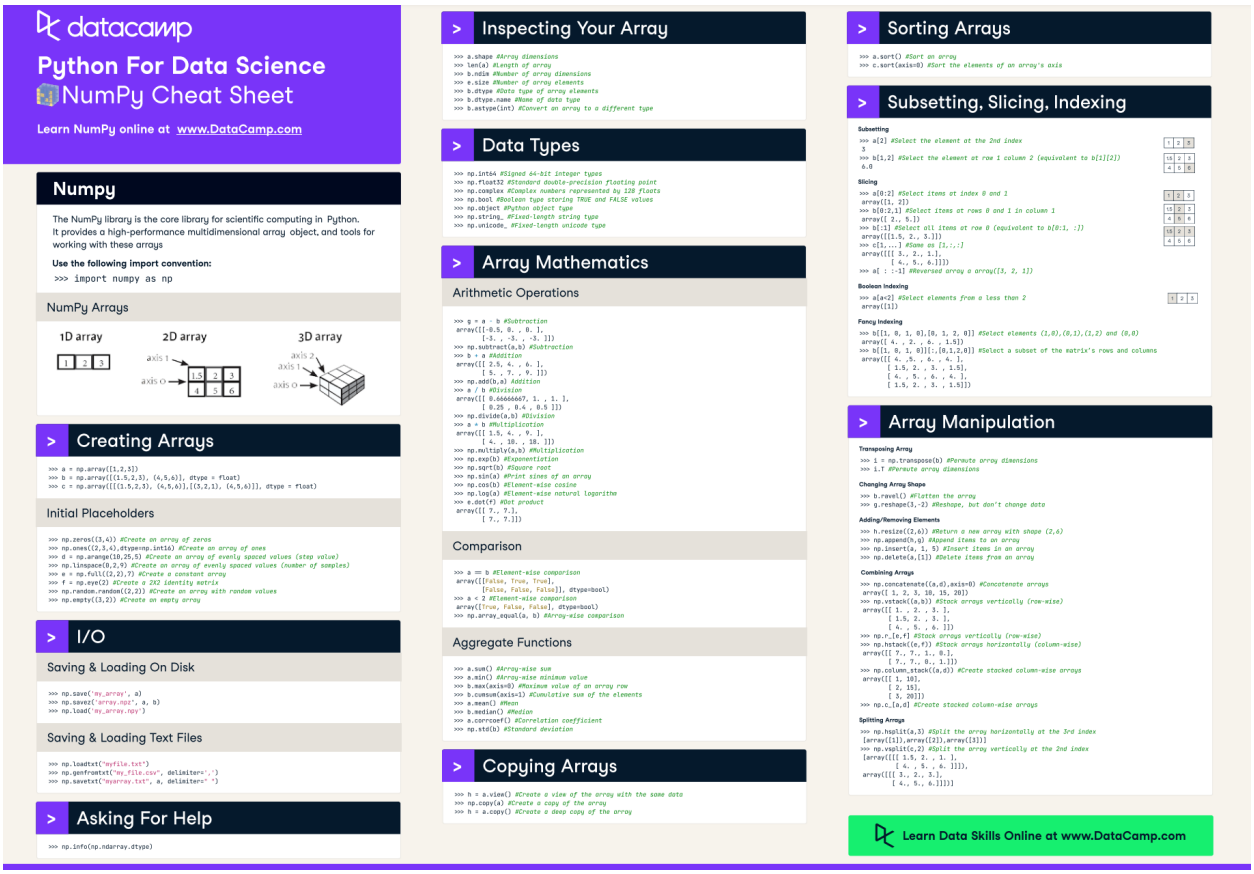


SO MANY Other Resources

Python is incredibly popular and so there are a lot of resources you can tap into online. You can either:

- Just start coding and google specific questions when you get stuck (my preference for learning)
- Follow a few online tutorials (like this first)
- Do both!

Here are a few resources: * Jake VanderPlas Python Data Science Handbook * Buy the book for $35 * Go through a series of well documented Colab Notebooks * **Highly recommended resource!** * Code Academy: Analyze Data with Python * Udemy Academy: Data Analysis with Pandas & NumPy in Python for Beginner * Datacamp: Introduction to Python



datacamp

Python For Data Science
NumPy Cheat Sheet

Learn NumPy online at www.DataCamp.com

Numpy

The NumPy library is the core library for scientific computing in Python. It provides a high-performance multidimensional array object, and tools for working with these arrays

Use the following import convention:

```
>>> import numpy as np
```

NumPy Arrays

1D array

2D array

3D array

Creating Arrays

```
>>> a = np.array([1,2,3])
>>> b = np.array([(1.5,2,3), (4,5,6)], dtype = float)
>>> c = np.array([[(1.5,2,3), (4,5,6)],[(3,2,1), (4,5,6)]], dtype = float)
```

Initial Placeholders

```
>>> np.zeros((3,4)) #Create an array of zeros
>>> np.ones((2,3,4),dtype=np.int16) #Create an array of ones
>>> d = np.arange(10,25,5) #Create an array of evenly spaced values (step value)
>>> np.linspace(0,2,9) #Create an array of evenly spaced values (number of samples)
>>> e = np.full((2,2),7) #Create a constant array
>>> f = np.eye(2) #Create a 2X2 identity matrix
>>> np.random.random((2,2)) #Create an array with random values
>>> np.empty((3,2)) #Create an empty array
```

I/O

Saving & Loading On Disk

```
>>> np.save('my_array', a)
>>> np.savez('array.npz', a, b)
>>> np.load('my_array.npy')
```

Saving & Loading Text Files

```
>>> np.loadtxt("myfile.txt")
>>> np.genfromtxt("my_file.csv", delimiter=',')
>>> np.savetxt("myarray.txt", a, delimiter=" ")
```

Asking For Help

```
>>> np.info(np.ndarray.dtype)
```

Inspecting Your Array

```
>>> a.shape #Array dimensions
>>> len(a) #Length of array
>>> b.ndim #Number of array dimensions
>>> e.size #Number of array elements
>>> b.dtype #Data type of array elements
>>> b.dtype.name #Name of data type
>>> b.astype(int) #Convert an array to a different type
```

Data Types

```
>>> np.int64 #Signed 64-bit integer types
>>> np.float32 #Standard double-precision floating point
>>> np.complex #Complex numbers represented by 128 floats
>>> np.bool #Boolean type storing TRUE and FALSE values
>>> np.object #Python object type
>>> np.string_ #Fixed-length string type
>>> np.unicode_ #Fixed-length unicode type
```

Array Mathematics

Arithmetic Operations

```
>>> g = a - b #Subtraction
 array([[-0.5, 0. , 0. ],
        [-3. , -3. , -3. ]])
>>> np.subtract(a,b) #Subtraction
>>> b + a #Addition
 array([[ 2.5, 4. , 6. ],
        [ 5. , 7. , 9. ]])
>>> np.add(b,a) #Addition
>>> a / b #Division
 array([[ 0.66666667, 1. , 1. ],
        [ 0.25 , 0.4 , 0.5 ]])
>>> np.divide(a,b) #Division
>>> a * b #Multiplication
 array([[ 1.5, 4. , 9. ],
        [ 4. , 10. , 18. ]])
>>> np.multiply(a,b) #Multiplication
>>> np.exp(b) #Exponentiation
>>> np.sqrt(b) #Square root
>>> np.sin(a) #Print sines of an array
>>> np.cos(b) #Element-wise cosine
>>> np.log(a) #Element-wise natural logarithm
>>> e.dot(f) #Dot product
 array([[ 7., 7.],
        [ 7., 7.]])
```

Comparison

```
>>> a == b #Element-wise comparison
 array([[False, True, True],
        [False, False, False]], dtype=bool)
>>> a < 2 #Element-wise comparison
 array([True, False, False], dtype=bool)
>>> np.array_equal(a, b) #Array-wise comparison
```

Aggregate Functions

```
>>> a.sum() #Array-wise sum
>>> a.min() #Array-wise minimum value
>>> b.max(axis=0) #Maximum value of an array row
>>> b.cumsum(axis=1) #Cumulative sum of the elements
>>> a.mean() #Mean
>>> b.median() #Median
>>> a.corrcoef() #Correlation coefficient
>>> np.std(b) #Standard deviation
```

Copying Arrays

```
>>> h = a.view() #Create a view of the array with the same data
>>> np.copy(a) #Create a copy of the array
>>> h = a.copy() #Create a deep copy of the array
```

Sorting Arrays

```
>>> a.sort() #Sort an array
>>> c.sort(axis=0) #Sort the elements of an array's axis
```

Subsetting, Slicing, Indexing

Subsetting

```
>>> a[2] #Select the element at the 2nd index
 3
>>> b[1,2] #Select the element at row 1 column 2 (equivalent to b[1][2])
 6.0
```

Slicing

```
>>> a[0:2] #Select items at index 0 and 1
 array([1, 2])
>>> b[0:2,1] #Select items at rows 0 and 1 in column 1
 array([ 2., 5.])
>>> b[:1] #Select all items at row 0 (equivalent to b[0:1, :])
 array([[1.5, 2., 3.]])
>>> c[1,...] #Same as [1,:,:]
 array([[[ 3., 2., 1.],
         [ 4., 5., 6.]]])
>>> a[ : :-1] #Reversed array a array([3, 2, 1])
```

Boolean Indexing

```
>>> a[a<2] #Select elements from a less than 2
 array([1])
```

Fancy Indexing

```
>>> b[[1, 0, 1, 0],[0, 1, 2, 0]] #Select elements (1,0),(0,1),(1,2) and (0,0)
 array([ 4. , 2. , 6. , 1.5])
>>> b[[1, 0, 1, 0]][:,[0,1,2,0]] #Select a subset of the matrix's rows and columns
 array([[ 4. ,5. , 6. , 4. ],
        [ 1.5, 2. , 3. , 1.5],
        [ 4. , 5. , 6. , 4. ],
        [ 1.5, 2. , 3. , 1.5]])
```

Array Manipulation

Transposing Array

```
>>> i = np.transpose(b) #Permute array dimensions
>>> i.T #Permute array dimensions
```

Changing Array Shape

```
>>> b.ravel() #Flatten the array
>>> g.reshape(3,-2) #Reshape, but don't change data
```

Adding/Removing Elements

```
>>> h.resize((2,6)) #Return a new array with shape (2,6)
>>> np.append(h,g) #Append items to an array
>>> np.insert(a, 1, 5) #Insert items in an array
>>> np.delete(a,[1]) #Delete items from an array
```

Combining Arrays

```
>>> np.concatenate((a,d),axis=0) #Concatenate arrays
 array([ 1, 2, 3, 10, 15, 20])
>>> np.vstack((a,b)) #Stack arrays vertically (row-wise)
 array([[ 1. , 2. , 3. ],
        [ 1.5, 2. , 3. ],
        [ 4. , 5. , 6. ]])
>>> np.r_[e,f] #Stack arrays vertically (row-wise)
>>> np.hstack((e,f)) #Stack arrays horizontally (column-wise)
 array([[ 7., 7., 1., 0.],
        [ 7., 7., 0., 1.]])
>>> np.column_stack((a,d)) #Create stacked column-wise arrays
 array([[ 1, 10],
        [ 2, 15],
        [ 3, 20]])
>>> np.c_[a,d] #Create stacked column-wise arrays
```

Splitting Arrays

```
>>> np.hsplit(a,3) #Split the array horizontally at the 3rd index
 [array([1]),array([2]),array([3])]
>>> np.vsplit(c,2) #Split the array vertically at the 2nd index
 [array([[[ 1.5, 2. , 1. ],
          [ 4. , 5. , 6. ]]]),
  array([[[ 3., 2., 3.],
          [ 4., 5., 6.]]])]
```

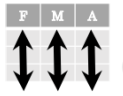
Learn Data Skills Online at www.DataCamp.com

Data Wrangling with pandas Cheat Sheet

http://pandas.pydata.org

Tidy Data – A foundation for wrangling in pandas

In a tidy data set:



Each **variable** is saved in its own **column**

Each **observation** is saved in its own **row**

Tidy data complements pandas's vectorized operations. pandas will automatically preserve observations as you manipulate variables. No other format works as intuitively with pandas.

M * A

Syntax – Creating DataFrames

	a	b	c
1	4	7	10
2	5	8	11
3	6	9	12

```
df = pd.DataFrame(
        {"a" : [4 ,5, 6],
         "b" : [7, 8, 9],
         "c" : [10, 11, 12]},
       index = [1, 2, 3])
```

Specify values for each column.

```
df = pd.DataFrame(
    [[4, 7, 10],
     [5, 8, 11],
     [6, 9, 12]],
    index=[1, 2, 3],
    columns=['a', 'b', 'c'])
```

Specify values for each row.

		a	b	c
n	v			
d	1	4	7	10
	2	5	8	11
e	2	6	9	12

```
df = pd.DataFrame(
        {"a" : [4 ,5, 6],
         "b" : [7, 8, 9],
         "c" : [10, 11, 12]},
index = pd.MultiIndex.from_tuples(
        [('d',1),('d',2),('e',2)],
          names=['n','v']))
```

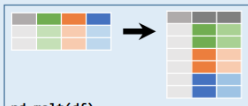
Create DataFrame with a MultiIndex

Method Chaining

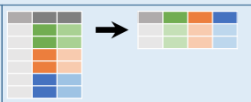
Most pandas methods return a DataFrame so that another pandas method can be applied to the result. This improves readability of code.

```
df = (pd.melt(df)
      .rename(columns={
              'variable' : 'var',
              'value' : 'val'})
      .query('val >= 200')
     )
```

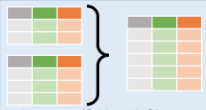
Reshaping Data – Change the layout of a data set



`pd.melt(df)`
Gather columns into rows.



`df.pivot(columns='var', values='val')`
Spread rows into columns.



`pd.concat([df1,df2])`
Append rows of DataFrames

`pd.concat([df1,df2], axis=1)`
Append columns of DataFrames

`df.sort_values('mpg')`
Order rows by values of a column (low to high).

`df.sort_values('mpg',ascending=False)`
Order rows by values of a column (high to low).

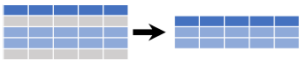
`df.rename(columns = {'y':'year'})`
Rename the columns of a DataFrame

`df.sort_index()`
Sort the index of a DataFrame

`df.reset_index()`
Reset index of DataFrame to row numbers, moving index to columns.

`df.drop(columns=['Length','Height'])`
Drop columns from DataFrame

Subset Observations (Rows)



`df[df.Length > 7]`
Extract rows that meet logical criteria.

`df.drop_duplicates()`
Remove duplicate rows (only considers columns).

`df.head(n)`
Select first n rows.

`df.tail(n)`
Select last n rows.

`df.sample(frac=0.5)`
Randomly select fraction of rows.

`df.sample(n=10)`
Randomly select n rows.

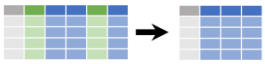
`df.iloc[10:20]`
Select rows by position.

`df.nlargest(n, 'value')`
Select and order top n entries.

`df.nsmallest(n, 'value')`
Select and order bottom n entries.

Logic in Python (and pandas)			
<	Less than	!=	Not equal to
>	Greater than	`df.column.isin(values)`	Group membership
==	Equals	`pd.isnull(obj)`	Is NaN
<=	Less than or equals	`pd.notnull(obj)`	Is not NaN
>=	Greater than or equals	`&,|,~,^,df.any(),df.all()`	Logical and, or, not, xor, any, all

Subset Variables (Columns)



`df[['width','length','species']]`
Select multiple columns with specific names.

`df['width']` *or* `df.width`
Select single column with specific name.

`df.filter(regex='regex')`
Select columns whose name matches regular expression *regex*.

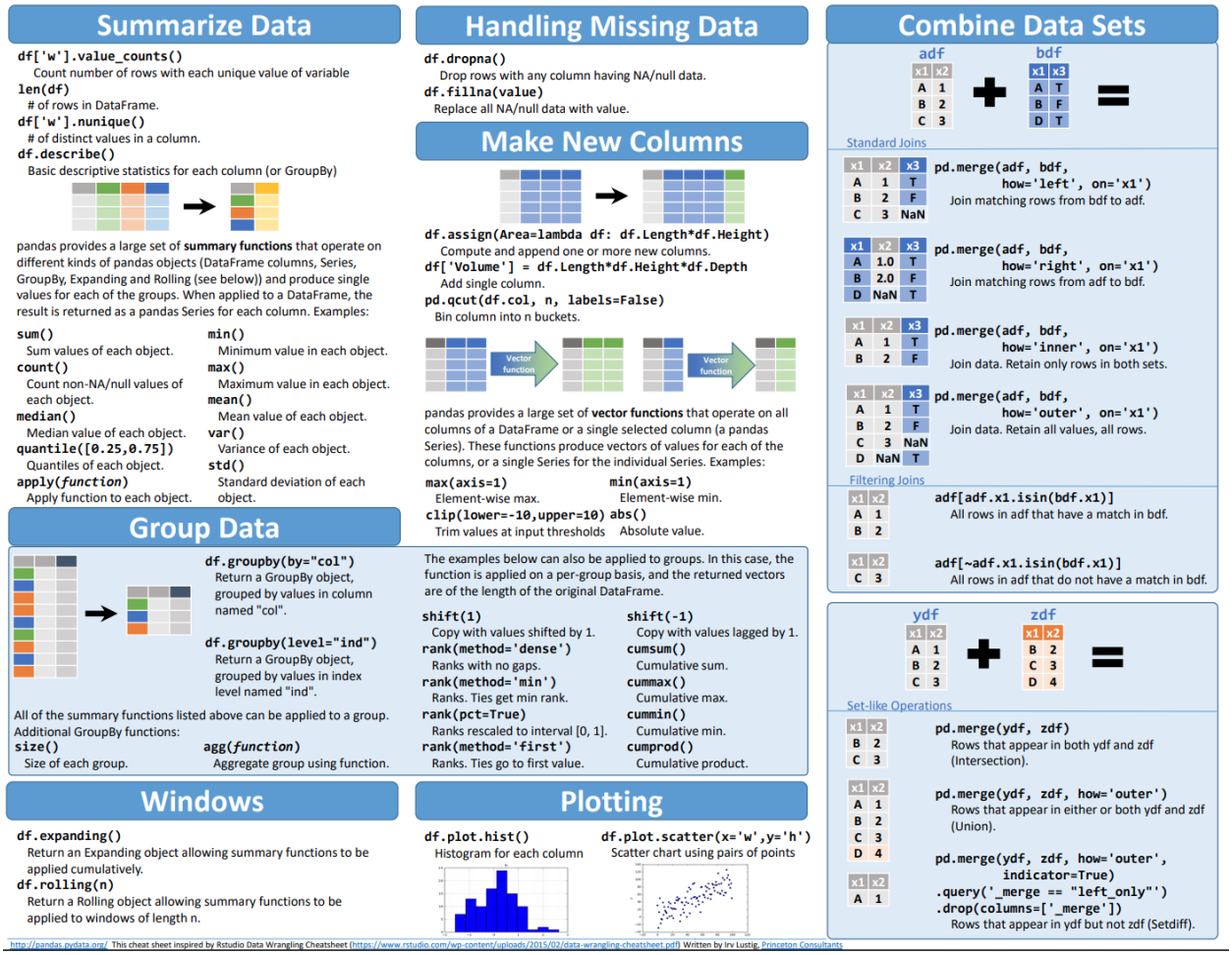
regex (Regular Expressions) Examples	
`'\.'`	Matches strings containing a period '.'
`'Length$'`	Matches strings ending with word 'Length'
`'^Sepal'`	Matches strings beginning with the word 'Sepal'
`'^x[1-5]$'`	Matches strings beginning with 'x' and ending with 1,2,3,4,5
`'^(?!Species$).*'`	Matches strings except the string 'Species'

`df.loc[:,'x2':'x4']`
Select all columns between x2 and x4 (inclusive).

`df.iloc[:,[1,2,5]]`
Select columns in positions 1, 2 and 5 (first column is 0).

`df.loc[df['a'] > 10, ['a','c']]`
Select rows meeting logical condition, and only the specific columns .

http://pandas.pydata.org/ This cheat sheet inspired by Rstudio Data Wrangling Cheatsheet (https://www.rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf) Written by Irv Lustig, Princeton Consultants



2.2.2 NumPy

NumPy provides an in depth overview of what exactly NumPy is. Quoting them: *>At the core of the NumPy package, is the ndarray object. This encapsulates n-dimensional arrays of homogeneous data types, with many operations being performed in compiled code for performance.*

Let's get started by importing it, typically shortened to `np` because of how frequently it's called. This will come standard in most Python distributions such as Anaconda. If you need to install it simply: ~~~ !pip install numpy ~~~

[ ]:
```
import numpy as np
```

Creating Arrays

Manually or from Lists

Manually create a list and demonstrate that operations on that list are difficult.

[ ]:
```
lst = [0, 1, 2, 3]
lst * 2
```

```
[0, 1, 2, 3, 0, 1, 2, 3]
```

That isn't what we expected! For lists, we need to loop through all elements to apply a function to them which can be incredibly time consuming.

[ ]:
```
[i * 2 for i in lst]
```

```
[0, 2, 4, 6]
```

Now lets make a numpy array and once in an array, let's show how intuitive operations are now that they are performed element by element.

[ ]:
```
array = np.array(lst)
print(array)
print(array * 2)
print(array + 2)
```

```
[0 1 2 3]
[0 2 4 6]
[2 3 4 5]
```

Let's create a 2D matrix of 32 bit floats.

[ ]:
```
np.array([[1, 0, 0],
          [0, 1, 0],
          [0, 0, 1]], dtype=np.float32)
```

```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]], dtype=float32)
```

Using Functions

We'll go through a few here, but for more in depth examples and options see NumPy's Array Creation Routines. These first few examples are for very basic arrays/matrices.

[ ]:
```
np.zeros(5)
```

```
array([0., 0., 0., 0., 0.])
```

[ ]:
```
np.zeros([2, 3])
```

```
array([[0., 0., 0.],
       [0., 0., 0.]])
```

[ ]:
```
np.eye(3)
```

```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

Now let's start making sequences.

[ ]:
```
np.arange(10)
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

[ ]:
```
np.arange(0,  #start
          10) #stop (not included in array)
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

[ ]:
```
np.arange(0,  #start
          10, #stop (not included in array)
          2)  #step size
```

```
array([0, 2, 4, 6, 8])
```

[ ]:
```
np.linspace(0,  #start
            10, #stop (default to be included, can pass in endpoint=False)
            6)  #number of data points evenly spaced
```

```
array([ 0.,  2.,  4.,  6.,  8., 10.])
```

[ ]:
```
np.logspace(0, #output starts being raised to this value
            3, #ending raised value
            4) #number of data points
```

```
array([   1.,   10.,  100., 1000.])
```

Logspace is the equivalent of rasing a base by a linspaced array.

[ ]:
```
10 ** np.linspace(0,3,4)
```

```
array([   1.,   10.,  100., 1000.])
```

[ ]:
```
np.logspace(0, 4, 5, base=2)
```

```
array([ 1.,  2.,  4.,  8., 16.])
```

If you don't want to have to do the mental math to know what exponent to raise the values to, you can use geomspace but this only helps for base of 10.

[ ]:
```
np.geomspace(1, 1000, 4)
```

```
array([   1.,   10.,  100., 1000.])
```

Random numbers!

[ ]:
```
np.random.rand(5)
```

```
array([0.05057259, 0.19699801, 0.46727495, 0.67357217, 0.60387782])
```

[ ]:
```
np.random.rand(2, 3)
```

```
array([[0.76299702, 0.06301467, 0.92031266],
       [0.42219446, 0.73721929, 0.39450975]])
```

Indexing

Let's first create a simple array.

```
[ ]: array = np.arange(1, 11, 1)
array
```

```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

Now index the first item.

```
[ ]: array[0]
```

```
1
```

Index the last item.

```
[ ]: array[-1]
```

```
10
```

Grab every 2nd item.

```
[ ]: array[::2]
```

```
array([1, 3, 5, 7, 9])
```

Grab every second item starting at index of 1 (the second value)

```
[ ]: array[1::2]
```

```
array([ 2,  4,  6,  8, 10])
```

Start from the second item, going to the 7th but skipping every other. (Our first time using the full `array[start (inclusive): stop (exclusive): step]` array 'slicing' notation)

```
[ ]: array[1:6:2]
```

```
array([2, 4, 6])
```

Reverse the order.

```
[ ]: array[::-1]
```

```
array([10,  9,  8,  7,  6,  5,  4,  3,  2,  1])
```

Boolean operations to index the array.

```
[ ]: array[array < 5]
```

```
array([1, 2, 3, 4])
```

```
[ ]: array[array * 2 < 5]
```

```
array([1, 2])
```

Integer list can also index arrays.

[ ]:
```
array[[1,3,5]]
```

```
array([2, 4, 6])
```

Now let's create a slightly more complicated, 2 dimensional array.

[ ]:
```
array_2d = np.arange(10).reshape((2, 5))
array_2d
```

```
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
```

Indexing the second dimension.

[ ]:
```
array_2d[:, 1]
```

```
array([1, 6])
```

Any combination of these indexing methods works as well.

[ ]:
```
array_2d[[True, False], 1::2]
```

```
array([[1, 3]])
```

[ ]:
```
array_2d[1, [0,1,4]]
```

```
array([5, 6, 9])
```

Operations

[ ]:
```
array
```

```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

[ ]:
```
array + 100
```

```
array([101, 102, 103, 104, 105, 106, 107, 108, 109, 110])
```

[ ]:
```
array * 2
```

```
array([ 2,  4,  6,  8, 10, 12, 14, 16, 18, 20])
```

To raise all the elements in an array to an exponent we have to use the notation ** not ^.

[ ]:
```
array ** 2
```

```
array([  1,   4,   9,  16,  25,  36,  49,  64,  81, 100])
```

Use that shape to create a new array matching it to do operations with.

[ ]:
```
array2 = np.arange(array.shape[0]) * 5
array2
```

```
array([ 0,  5, 10, 15, 20, 25, 30, 35, 40, 45])
```

[ ]:
```
array2 + array
```

```
array([ 1,  7, 13, 19, 25, 31, 37, 43, 49, 55])
```

Stats of an Array

[ ]:
```
print(array)
print(array_2d)
```

```
[ 1  2  3  4  5  6  7  8  9 10]
[[0 1 2 3 4]
 [5 6 7 8 9]]
```

[ ]:
```
print(array.shape)
print(array_2d.shape)
```

```
(10,)
(2, 5)
```

[ ]:
```
print(len(array))
print(len(array_2d))
```

```
10
2
```

[ ]:
```
print(array.max())
print(array_2d.max())
```

```
10
9
```

[ ]:
```
print(array.min())
print(array_2d.min())
```

```
1
0
```

[ ]:
```
array.std()
```

```
2.8722813232690143
```

[ ]:
```
array.cumsum()
```

```
array([ 1,  3,  6, 10, 15, 21, 28, 36, 45, 55])
```

[ ]:
```
array.cumprod()
```

```
array([      1,       2,       6,      24,     120,     720,    5040,
         40320,  362880, 3628800])
```

Constants

[ ]:
```
np.pi
```
```
3.141592653589793
```

[ ]:
```
np.e
```
```
2.718281828459045
```

[ ]:
```
np.inf
```
```
inf
```

Functions

[ ]:
```
np.sin(np.pi / 2)
```
```
1.0
```

[ ]:
```
np.cos(np.pi)
```
```
-1.0
```

[ ]:
```
np.log(np.e)
```
```
1.0
```

[ ]:
```
np.log10(100)
```
```
2.0
```

[ ]:
```
np.log2(64)
```
```
6.0
```

To demonstrate rounding, let's first make a new array with decimals.

[ ]:
```
array = np.arange(4) / 3
print(array)
np.around(array, 2)
```
```
[0.         0.33333333 0.66666667 1.        ]
```
```
array([0.  , 0.33, 0.67, 1.  ])
```

Looping vs Vectorization

As mentioned in the beginning, NumPy uses machine code with their ndarray objects which is what leads to the performance improvements. Let's demonstrate this by constructing a simple sine wave.

[ ]:
```
fs = 500 #sampling rate in Hz
d_t = 1 / fs #time steps in seconds
n_step = 1000 #number of steps (there will be n_step+1 data points)

amp = 1 #amplitude of sine wave
f = 2 #frequency of sine wave

time = np.linspace(0,              #start time
                   n_step * d_t,     #end time
                   num=n_step + 1)  #number of data points, one more than the steps to␣
↪include an endpoint
time.shape
```

```
(1001,)
```

First we make a function to loop through each element and calculate the amplitude.

[ ]:
```
def sine_wave_with_loop(time, amp, f, phase=0):
  length = time.shape[0]
  wave = np.zeros(length)

  for i in range(length-1):
    wave[i] = np.sin(2 * np.pi * f * time[i] + phase * np.pi / 180)*amp
  return wave
```

Now let's time how quickly that executes for our time array of 1,001 data points.

[ ]:
```
%timeit sine_wave_with_loop(time, amp, f)
```

```
The slowest run took 7.52 times longer than the fastest. This could mean that an␣
↪intermediate result is being cached.
100 loops, best of 5: 2.55 ms per loop
```

Now let's do the same using NumPy's sine function and vectorization.

[ ]:
```
def sine_wave_with_numpy(time, amp, f, phase=0):
  """Takes in a time array and sine wave parameters, returns an array of the sine wave␣
↪amplitude."""
  return np.sin(2 * np.pi * f * time + phase * np.pi / 180) * amp
```

Notice my docstrings!

[ ]:
```
help(sine_wave_with_numpy)
```

```
Help on function sine_wave_with_numpy in module __main__:

sine_wave_with_numpy(time, amp, f, phase=0)
    Takes in a time array and sine wave parameters, returns an array of the sine wave␣
↪amplitude.
```

[ ]:
```
%timeit sine_wave_with_numpy(time, amp, f)
```

```
The slowest run took 4.34 times longer than the fastest. This could mean that an␣
↪intermediate result is being cached.
10000 loops, best of 5: 27.9 µs per loop
```

Using vectorization is about **100x faster!** And this increases the longer the loops are.

Why Vectorization Works So Much Faster

The above example highlights that NumPy is much faster, but why? Because it is using compiled machine code under the hood for it's operations.

Python has the Numba package which can be used to do this compilation which we will do to highlight just why NumPy is faster (and recommended!).

[ ]:
```
from numba import njit

numba_sine_wave_with_loop = njit(sine_wave_with_loop)
numba_sine_wave_with_numpy = njit(sine_wave_with_numpy)
```

[ ]:
```
%timeit numba_sine_wave_with_loop(time, amp, f)
%timeit numba_sine_wave_with_numpy(time, amp, f)
```

```
The slowest run took 14426.53 times longer than the fastest. This could mean that an␣
↪intermediate result is being cached.
1 loop, best of 5: 22.8 µs per loop
The slowest run took 11783.25 times longer than the fastest. This could mean that an␣
↪intermediate result is being cached.
1 loop, best of 5: 22.1 µs per loop
```

Let's combine this into a DataFrame we'll discuss next in more detail, but here's a preview.

[ ]:
```
import pandas as pd

time_data = pd.DataFrame({'Time (us)':[2.77*100, 27.5, 23.1, 22.6],
                          'Method':['Loop','NumPy','Loop','NumPy'],
                          'Numba?':['w/o','w/o','w','w']}
)
time_data
```

```
   Time (us) Method Numba?
0      277.0   Loop    w/o
1       27.5  NumPy    w/o
2       23.1   Loop      w
3       22.6  NumPy      w
```

Now let's plot it and preview Plotly!

[ ]:
```
!pip install --upgrade -q plotly
import plotly.express as px
import plotly.io as pio; pio.renderers.default = "iframe"

fig = px.bar(time_data,
```

(continues on next page)

(continued from previous page)

```
            x="Method",
            y="Time (us)",
            color="Numba?",
            title="Compute Time of a Sine Wave for 1000 Elements",
            barmode='group')
fig.show()
```

```
     || 23.9 MB 1.5 MB/s
```

2.2.3 Pandas

Pandas is built *on top of* NumPy meaning that the data is stored still as NumPy ndarray objects under the hood. But it exposes a much more intuitive labeling/indexing architecture and allows you to link arrays of different types (strings, floats, integers etc.) to one another.

To quote Jake VanderPlas: >*At the very basic level, Pandas objects can be thought of as enhanced versions of NumPy structured arrays in which the rows and columns are identified with labels rather than simple integer indices.*

To start, import pandas as pd, again this will come standard in virtually all Python distributions such as Anaconda. But to install is simply: ~~~ !pip install pandas ~~~

[ ]:
```
import pandas as pd
```

There are three types of Pandas objects, we'll only focus on the first two: 1. **Series** – 1D labeled homogeneous array, sizeimmutable 2. **Data Frames** – 2D labeled, size-mutable tabular structure with heterogenic columns 3. **Panel** – 3D labeled size mutable array.

Creating a Series

First let's create a few numpy arrays.

[ ]:
```
amplitude = sine_wave_with_numpy(time, amp, f, 180)
print(time)
print(amplitude)
```

```
[0.    0.002 0.004 ... 1.996 1.998 2.   ]
[ 1.22464680e-16 -2.51300954e-02 -5.02443182e-02 ...  5.02443182e-02
  2.51300954e-02  1.10218212e-15]
```

Now let's see what a series looks like made from one of the arrays.

[ ]:
```
pd.Series(amplitude)
```

```
0       1.224647e-16
1      -2.513010e-02
2      -5.024432e-02
3      -7.532681e-02
4      -1.003617e-01
            ...
996     1.003617e-01
997     7.532681e-02
```

(continues on next page)

(continued from previous page)

```
998     5.024432e-02
999     2.513010e-02
1000    1.102182e-15
Length: 1001, dtype: float64
```

This type of series has some value, but you really start to see it when you add in an index.

[ ]:
```
series = pd.Series(data=amplitude,
                   index=time,
                   name='Amplitude')
series
```

```
0.000    1.224647e-16
0.002   -2.513010e-02
0.004   -5.024432e-02
0.006   -7.532681e-02
0.008   -1.003617e-01
            ...
1.992    1.003617e-01
1.994    7.532681e-02
1.996    5.024432e-02
1.998    2.513010e-02
2.000    1.102182e-15
Name: Amplitude, Length: 1001, dtype: float64
```

Here's where Pandas shines - indexing is much more intuitive (and inclusive) to specify based on labels, not those confusing integer locations. We'll come back to this when we have the dataframe next too.

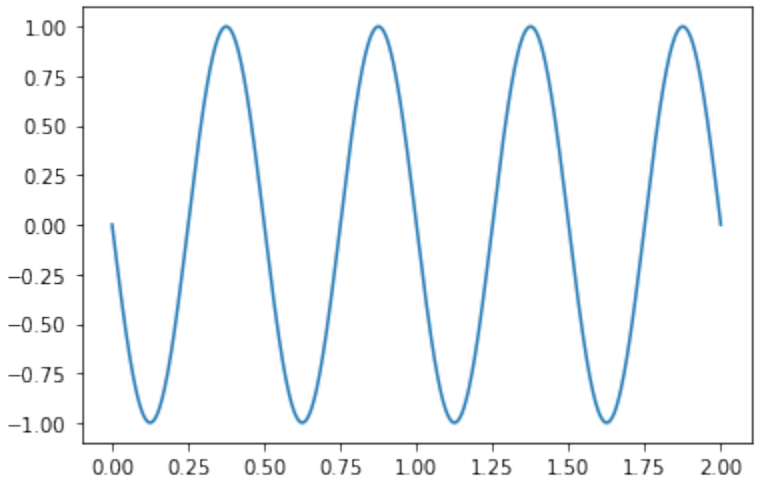
[ ]:
```
series[0: 0.01]
```

```
0.000    1.224647e-16
0.002   -2.513010e-02
0.004   -5.024432e-02
0.006   -7.532681e-02
0.008   -1.003617e-01
0.010   -1.253332e-01
Name: Amplitude, dtype: float64
```

Being able to plot quickly is also a plus!

[ ]:
```
series.plot()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f51188d8310>
```



Remember we never left the NumPy array, it is still here and can be accessed with the following.

[ ]:
```
series.values
```

```
array([ 1.22464680e-16, -2.51300954e-02, -5.02443182e-02, ...,
        5.02443182e-02,  2.51300954e-02,  1.10218212e-15])
```

[ ]:
```
series.to_numpy()
```

```
array([ 1.22464680e-16, -2.51300954e-02, -5.02443182e-02, ...,
        5.02443182e-02,  2.51300954e-02,  1.10218212e-15])
```

Creating a DataFrame

A DataFrame is basically a sequence of aligned series objects, and by aligned I mean they share a common index or label. This let's us mix and match types easily among other benefits.

First we'll start creating dataframes using what is called a "dictionary" with keys and values.

[ ]:
```
df = pd.DataFrame({"Phase 0": sine_wave_with_numpy(time, amp, f, 00),
                   "Phase 90": sine_wave_with_numpy(time, amp, f, 90),
                   "Phase 180": sine_wave_with_numpy(time, amp, f, 180),
                   "Phase 270": sine_wave_with_numpy(time, amp, f, 270)},
                  index=time)
df
```

```
            Phase 0  Phase 90     Phase 180  Phase 270
0.000  0.000000e+00  1.000000  1.224647e-16  -1.000000
0.002  2.513010e-02  0.999684 -2.513010e-02  -0.999684
0.004  5.024432e-02  0.998737 -5.024432e-02  -0.998737
0.006  7.532681e-02  0.997159 -7.532681e-02  -0.997159
0.008  1.003617e-01  0.994951 -1.003617e-01  -0.994951
...             ...       ...           ...        ...
1.992 -1.003617e-01  0.994951  1.003617e-01  -0.994951
1.994 -7.532681e-02  0.997159  7.532681e-02  -0.997159
1.996 -5.024432e-02  0.998737  5.024432e-02  -0.998737
1.998 -2.513010e-02  0.999684  2.513010e-02  -0.999684
```

(continues on next page)

(continued from previous page)

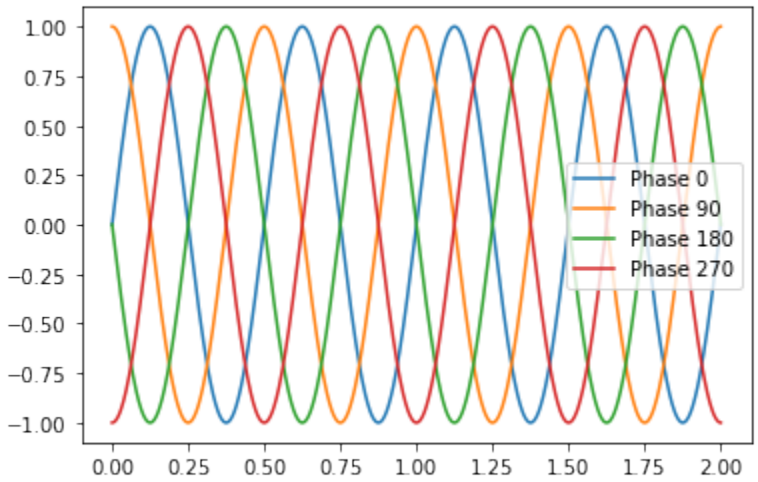
```
2.000 -9.797174e-16  1.000000  1.102182e-15  -1.000000

[1001 rows x 4 columns]
```

Plotting (Preview)

Dataframes also wrap around Matplotlib to allow for plotting directly from the dataframe object itself. This can also be done from the Pandas Series object too like we showed earlier.

[ ]:
```
df.plot()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f51187bd810>
```



[ ]:
```
df['Max'] = df.max(axis=1)
df['Min'] = df.min(axis=1)
df
```

```
           Phase 0  Phase 90     Phase 180  Phase 270       Max       Min
0.000  0.000000e+00  1.000000  1.224647e-16  -1.000000  1.000000 -1.000000
0.002  2.513010e-02  0.999684 -2.513010e-02  -0.999684  0.999684 -0.999684
0.004  5.024432e-02  0.998737 -5.024432e-02  -0.998737  0.998737 -0.998737
0.006  7.532681e-02  0.997159 -7.532681e-02  -0.997159  0.997159 -0.997159
0.008  1.003617e-01  0.994951 -1.003617e-01  -0.994951  0.994951 -0.994951
...             ...       ...           ...        ...       ...       ...
1.992 -1.003617e-01  0.994951  1.003617e-01  -0.994951  0.994951 -0.994951
1.994 -7.532681e-02  0.997159  7.532681e-02  -0.997159  0.997159 -0.997159
1.996 -5.024432e-02  0.998737  5.024432e-02  -0.998737  0.998737 -0.998737
1.998 -2.513010e-02  0.999684  2.513010e-02  -0.999684  0.999684 -0.999684
2.000 -9.797174e-16  1.000000  1.102182e-15  -1.000000  1.000000 -1.000000

[1001 rows x 6 columns]
```

This will be the topic of the next webinar, plotting with Plotly!

Note that I need to install an upgraded version of Plotly in Colab because the default Plotly Express version doesn't work in Colab (but their more advanced graph objects does).

[ ]:
```
!pip install --upgrade -q plotly
import plotly.express as px
```

[ ]:
```
px.line(df).show()
```

Load from CSV

This dataset was discussed in a blog on vibration metrics and used bearing data as an example.

Note you don't *have* to use a CSV. They have a lot of other file formats natively supported (see full list): * hdf * feather * pickle

But I know everyone likes CSVs!

[ ]:
```
df = pd.read_csv('https://info.endaq.com/hubfs/Plots/bearing_data.csv', index_col=0)
df
```

```
           Fault_021  Fault_014  Fault_007    Normal
Time
0.000000   -0.105351  -0.074395   0.053116  0.046104
0.000083    0.132888   0.056365   0.116628 -0.037134
0.000167   -0.056535   0.201257   0.083654 -0.089496
0.000250   -0.193178  -0.024528  -0.026477 -0.084906
0.000333    0.064879  -0.072284   0.045319 -0.038594
...              ...        ...        ...       ...
9.999667    0.095754   0.145055  -0.098923  0.064254
9.999750   -0.123083   0.092263  -0.067573  0.070721
9.999833   -0.036508  -0.168120   0.005685  0.103265
9.999917    0.097006  -0.035898   0.093400  0.124335
10.000000  -0.008762   0.165846   0.130923  0.114947

[120000 rows x 4 columns]
```

Save CSV

Like reading data, there are a host of native formats we can save data from a dataframe. See documentation.

[ ]:
```
df.to_csv('bearing-data.csv')
```

Simple Analysis

[ ]:
```
df.describe()
```

```
           Fault_021      Fault_014      Fault_007         Normal
count  120000.000000  120000.000000  120000.000000  120000.000000
mean        0.012251       0.002729       0.002953       0.010755
std         0.198383       0.157761       0.121272       0.065060
min        -1.037862      -1.338628      -0.650390      -0.269114
25%        -0.107020      -0.096649      -0.072284      -0.032544
50%         0.011682       0.001299       0.004548       0.013351
```

(continues on next page)

(continued from previous page)

```
75%         0.132054       0.100872       0.080081       0.056535
max         0.917908       1.124376       0.594025       0.251382
```

[ ]:
```
df.std()
```

```
Fault_021    0.198383
Fault_014    0.157761
Fault_007    0.121272
Normal       0.065060
dtype: float64
```

[ ]:
```
df.max()
```

```
Fault_021    0.917908
Fault_014    1.124376
Fault_007    0.594025
Normal       0.251382
dtype: float64
```

Note that these built in Pandas functions are using NumPy to process and are the equivalent of doing the following.

[ ]:
```
np.max(df)
```

```
Fault_021    0.917908
Fault_014    1.124376
Fault_007    0.594025
Normal       0.251382
dtype: float64
```

[ ]:
```
df.quantile(0.25)
```

```
Fault_021   -0.107020
Fault_014   -0.096649
Fault_007   -0.072284
Normal      -0.032544
Name: 0.25, dtype: float64
```

[ ]:
```
df['abs(max)'] = df.abs().max(axis=1)
df
```

```
           Fault_021  Fault_014  Fault_007    Normal  abs(max)
Time
0.000000   -0.105351  -0.074395   0.053116  0.046104  0.105351
0.000083    0.132888   0.056365   0.116628 -0.037134  0.132888
0.000167   -0.056535   0.201257   0.083654 -0.089496  0.201257
0.000250   -0.193178  -0.024528  -0.026477 -0.084906  0.193178
0.000333    0.064879  -0.072284   0.045319 -0.038594  0.072284
...              ...        ...        ...       ...       ...
9.999667    0.095754   0.145055  -0.098923  0.064254  0.145055
9.999750   -0.123083   0.092263  -0.067573  0.070721  0.123083
9.999833   -0.036508  -0.168120   0.005685  0.103265  0.168120
9.999917    0.097006  -0.035898   0.093400  0.124335  0.124335
10.000000  -0.008762   0.165846   0.130923  0.114947  0.165846
```

(continues on next page)

(continued from previous page)

```
[120000 rows x 5 columns]
```

Indexing

Here is where indexing in Python gets a whole lot more intuitive! A dataframe with an index let's use index values (time in this case) to slice the dataframe, not rely on the nth element in the arrays.

[ ]:
```
df[0: 0.05]
```

```
          Fault_021  Fault_014  Fault_007    Normal  abs(max)
Time
0.000000  -0.105351  -0.074395   0.053116  0.046104  0.105351
0.000083   0.132888   0.056365   0.116628 -0.037134  0.132888
0.000167  -0.056535   0.201257   0.083654 -0.089496  0.201257
0.000250  -0.193178  -0.024528  -0.026477 -0.084906  0.193178
0.000333   0.064879  -0.072284   0.045319 -0.038594  0.072284
...             ...        ...        ...       ...       ...
0.049584   0.131010   0.129136  -0.014619  0.021487  0.131010
0.049667   0.437675  -0.221399  -0.025340  0.021070  0.437675
0.049750   0.095754  -0.120689   0.033137  0.035256  0.120689
0.049834  -0.137269   0.275977   0.023716  0.044226  0.275977
0.049917   0.150203   0.019167  -0.044670  0.005424  0.150203

[600 rows x 5 columns]
```

We can also use the same convention as before by adding in a step definition, in this case we'll grab every 100th point.

[ ]:
```
df[0: 0.05: 100]
```

```
          Fault_021  Fault_014  Fault_007    Normal  abs(max)
Time
0.000000  -0.105351  -0.074395   0.053116  0.046104  0.105351
0.008333  -0.481067  -0.137745  -0.010558 -0.012934  0.481067
0.016667  -0.265985  -0.073746  -0.140669  0.027329  0.265985
0.025000  -0.192343   0.203856  -0.168120 -0.029832  0.203856
0.033334   0.018984   0.154476  -0.072933  0.076353  0.154476
0.041667  -0.289975  -0.227409   0.088202  0.016898  0.289975
```

There are ways to use the integer based indexing if you so desire.

[ ]:
```
df.iloc[0:10]
```

```
          Fault_021  Fault_014  Fault_007    Normal  abs(max)
Time
0.000000  -0.105351  -0.074395   0.053116  0.046104  0.105351
0.000083   0.132888   0.056365   0.116628 -0.037134  0.132888
0.000167  -0.056535   0.201257   0.083654 -0.089496  0.201257
0.000250  -0.193178  -0.024528  -0.026477 -0.084906  0.193178
0.000333   0.064879  -0.072284   0.045319 -0.038594  0.072284
0.000417   0.214874   0.034761   0.060751  0.025451  0.214874
0.000500  -0.076353   0.094212  -0.174130  0.040680  0.174130
0.000583  -0.065922  -0.070010  -0.229521  0.042558  0.229521
```

(continues on next page)

(continued from previous page)

```
0.000667   0.206529  -0.079431   0.045482  0.038177  0.206529
0.000750   0.021487   0.092426   0.027452  0.044018  0.092426
```

Rolling

I love the rolling method which allows for easy rolling window calculations, something you'll do frequently with time series data.

[ ]:
```
n = int(df.shape[0] / 100)
df.rolling(n).max()
```

```
           Fault_021  Fault_014  Fault_007   Normal  abs(max)
Time
0.000000         NaN        NaN        NaN      NaN       NaN
0.000083         NaN        NaN        NaN      NaN       NaN
0.000167         NaN        NaN        NaN      NaN       NaN
0.000250         NaN        NaN        NaN      NaN       NaN
0.000333         NaN        NaN        NaN      NaN       NaN
...              ...        ...        ...      ...       ...
9.999667    0.748721   0.768318   0.408362  0.18525  0.933137
9.999750    0.748721   0.768318   0.408362  0.18525  0.933137
9.999833    0.748721   0.768318   0.408362  0.18525  0.933137
9.999917    0.748721   0.768318   0.408362  0.18525  0.933137
10.000000   0.748721   0.768318   0.408362  0.18525  0.933137

[120000 rows x 5 columns]
```

[ ]:
```
df.rolling(n).max()[::n]
```

```
          Fault_021  Fault_014  Fault_007    Normal  abs(max)
Time
0.000000        NaN        NaN        NaN       NaN       NaN
0.100001   0.726190   0.710166   0.443448  0.184416  0.813809
0.200002   0.696150   0.641131   0.411773  0.204443  0.740376
0.300003   0.491289   0.846612   0.492178  0.172525  0.846612
0.400003   0.578699   0.608482   0.524828  0.204860  0.608482
...             ...        ...        ...       ...       ...
9.500079   0.731823   0.500950   0.474798  0.206112  0.760820
9.600080   0.697818   0.713253   0.470412  0.190466  0.713253
9.700081   0.677791   0.547244   0.415996  0.229060  0.677791
9.800082   0.625220   0.569498   0.391469  0.176489  0.653801
9.900083   0.749555   0.949271   0.332342  0.176489  0.949271

[100 rows x 5 columns]
```

[ ]:
```
px.line(df.rolling(n).max()[::n]).show()
```

Datetime Data (Yay Finance!)

Let's use Yahoo Finance and stock data as a relatable example of data with datetimes.

[ ]:
```
!pip install -q yfinance
import yfinance as yf
```

```
     || 6.3 MB 56.8 MB/s
  Building wheel for yfinance (setup.py) ... done
```

[ ]:
```
df =  yf.download(["SPY", "AAPL", "MSFT", "AMZN", "GOOGL"],
                  start='2019-01-01',
                  end='2021-09-24')
df
```

```
[**********************100%**********************]  5 of 5 completed
```

	Adj Close				...	Volume		
	AAPL	AMZN	GOOGL		...	GOOGL	MSFT	SPY
Date					...			
2019-01-02	38.382229	1539.130005	1054.680054		...	1593400	35329300	126925200
2019-01-03	34.559078	1500.280029	1025.469971		...	2098000	42579100	144140700
2019-01-04	36.034370	1575.390015	1078.069946		...	2301100	44060600	142628800
2019-01-07	35.954170	1629.510010	1075.920044		...	2372300	35656100	103139100
2019-01-08	36.639565	1656.579956	1085.369995		...	1770700	31514400	102512600
...	...	...	...		...	...	...	...
2021-09-17	146.059998	3462.520020	2816.000000		...	2666800	41309300	118220200
2021-09-20	142.940002	3355.729980	2774.389893		...	2325900	38278700	166445500
2021-09-21	143.429993	3343.629883	2780.659912		...	1266600	22364100	92526100
2021-09-22	145.850006	3380.050049	2805.669922		...	1252800	26626300	102350100
2021-09-23	146.830002	3416.000000	2824.320068		...	1047600	18604600	76396000

[688 rows x 30 columns]

Let's compare not the price, but the relative performance.

[ ]:
```
df = df['Adj Close']
df = df / df.iloc[0]
df
```

	AAPL	AMZN	GOOGL	MSFT	SPY
Date					
2019-01-02	1.000000	1.000000	1.000000	1.000000	1.000000
2019-01-03	0.900393	0.974758	0.972304	0.963212	0.976137
2019-01-04	0.938830	1.023559	1.022177	1.008010	1.008834
2019-01-07	0.936740	1.058721	1.020139	1.009296	1.016788
2019-01-08	0.954597	1.076309	1.029099	1.016614	1.026341
...	...	...	...	...	...
2021-09-17	3.805407	2.249661	2.670004	3.061106	1.849225
2021-09-20	3.724119	2.180277	2.630551	3.004247	1.818391
2021-09-21	3.736885	2.172416	2.636496	3.009351	1.816673
2021-09-22	3.799936	2.196078	2.660210	3.047938	1.834395

(continues on next page)

(continued from previous page)

```
2021-09-23  3.825468  2.219436  2.677893  3.057942  1.856682

[688 rows x 5 columns]
```

[ ]:
```
px.line(df).show()
```

The rolling function will play very nicely with datetime data as shown here when I get the moving average over a 40 day period. And this can handle unevenly sampled date easily.

[ ]:
```
px.line(df.rolling('40d').mean()).show()
```

Indexing with datetime data though will require a slightly extra step, but then it is easy.

[ ]:
```
from datetime import date
start = date(2021, 4, 1)
end = date(2021, 4, 30)
```

[ ]:
```
df[start:end]
```

```
                AAPL      AMZN      GOOGL      MSFT       SPY
Date
2021-04-01  3.194388  2.053758  2.019361  2.463520  1.667522
2021-04-05  3.269703  2.096464  2.103918  2.531830  1.691456
2021-04-06  3.277754  2.094573  2.094721  2.519530  1.690457
2021-04-07  3.321644  2.130678  2.122947  2.540267  1.692414
2021-04-08  3.385532  2.143614  2.133756  2.574320  1.700447
2021-04-09  3.454094  2.190978  2.152947  2.600749  1.712810
2021-04-12  3.408386  2.195649  2.128247  2.601359  1.713434
2021-04-13  3.491232  2.209040  2.137549  2.627585  1.718512
2021-04-14  3.428903  2.165509  2.125678  2.598106  1.712644
2021-04-15  3.493050  2.195455  2.166771  2.637852  1.731041
2021-04-16  3.484220  2.208676  2.164400  2.650457  1.736827
2021-04-19  3.501880  2.190855  2.171047  2.630127  1.728294
2021-04-20  3.456951  2.166607  2.160854  2.625247  1.715640
2021-04-21  3.467079  2.184364  2.160229  2.648830  1.731874
2021-04-22  3.426566  2.149942  2.135738  2.614167  1.716057
2021-04-23  3.488376  2.170629  2.180690  2.654624  1.734663
2021-04-26  3.498764  2.214888  2.190171  2.658690  1.738284
2021-04-27  3.490193  2.220365  2.172204  2.662960  1.737910
2021-04-28  3.469157  2.247049  2.236735  2.587636  1.737410
2021-04-29  3.466560  2.255372  2.268707  2.566798  1.748482
2021-04-30  3.414100  2.252844  2.231482  2.563443  1.736994
```

[ ]:
```
pd.date_range(start='1/1/2019', end='08/31/2021', freq='M')
```

```
DatetimeIndex(['2019-01-31', '2019-02-28', '2019-03-31', '2019-04-30',
               '2019-05-31', '2019-06-30', '2019-07-31', '2019-08-31',
               '2019-09-30', '2019-10-31', '2019-11-30', '2019-12-31',
               '2020-01-31', '2020-02-29', '2020-03-31', '2020-04-30',
               '2020-05-31', '2020-06-30', '2020-07-31', '2020-08-31',
               '2020-09-30', '2020-10-31', '2020-11-30', '2020-12-31',
```

(continues on next page)

(continued from previous page)

```
               '2021-01-31', '2021-02-28', '2021-03-31', '2021-04-30',
               '2021-05-31', '2021-06-30', '2021-07-31', '2021-08-31'],
              dtype='datetime64[ns]', freq='M')
```

[ ]:
```
df.resample(rule='Q').max()
```

```
                AAPL      AMZN     GOOGL      MSFT       SPY
Date
2019-03-31  1.240671  1.182005  1.172043  1.193962  1.143113
2019-06-30  1.346619  1.275045  1.228998  1.373424  1.187797
2019-09-30  1.435250  1.313073  1.181344  1.410902  1.218385
2019-12-31  1.887424  1.214842  1.291833  1.595237  1.315273
2020-03-31  2.108057  1.410030  1.445813  1.893692  1.377995
2020-06-30  2.367841  1.796086  1.388762  2.053599  1.324073
2020-09-30  3.473547  2.294446  1.628352  2.343208  1.471860
2020-12-31  3.544629  2.237387  1.730354  2.281494  1.551179
2021-03-31  3.712409  2.196046  2.008780  2.484634  1.649707
2021-06-30  3.562980  2.277546  2.323662  2.765187  1.787611
2021-09-30  4.082358  2.424363  2.753736  3.115720  1.892556
```

Sorting & Filtering on Tabular Data

To highlight filtering in DataFrames, we'll use a dataset with a bunch of different columns/series of different types. This data was pulled directly from the enDAQ cloud API off some example recording files.

[ ]:
```
df = pd.read_csv('https://info.endaq.com/hubfs/data/endaq-cloud-table.csv')
df
```

```
    Unnamed: 0  ...                                    psdResultant1Hz
0            0  ...                                [0. 0. 0. ... 0. 0. 0.]
1            1  ...                            [ 0.  0.  0. ... nan nan nan]
2            2  ...          [0.    0.001 0.002 ... 0.002 0.001 0.001]
3            3  ...                            [ 0.  0.  0. ... nan nan nan]
4            4  ...          [0.001 0.002 0.002 ...   nan   nan   nan]
5            5  ...                            [ 0.  0.  0. ... nan nan nan]
6            6  ...          [0.157 0.304 0.42  ... 0.001 0.01  0.013]
7            7  ...          [0.007 0.021 0.055 ...   nan   nan   nan]
8            8  ...                            [ 0.  0.  0. ... nan nan nan]
9            0  ...                                                    []
10           1  ...          [0.02  0.071 0.138 ...   nan   nan   nan]
11           2  ...          [0.021 0.074 0.137 ...   nan   nan   nan]
12           3  ...                                                    []
13           4  ...                                                    []
14           5  ...          [0.006 0.016 0.04  ...   nan   nan   nan]
15           6  ...                                                    []
16           7  ...  [2.250e-01 8.390e-01 2.025e+00 ... 1.000e-03 1...
17           8  ...          [0.001 0.001 0.003 ...   nan   nan   nan]
18           9  ...    [ 2.09   8.631 18.196 ... 69.565 59.225 60.801]
19          10  ...    [ 0.072  0.302  0.816 ... 52.226 45.804 52.81 ]
20          11  ...    [ 0.162  0.587  1.495 ... 45.639 43.86  40.177]
21          12  ...    [ 0.371  0.409  0.787 ... 27.577 24.778 37.822]
22          13  ...          [0.008 0.02  0.03  ...   nan   nan   nan]
```

(continues on next page)

(continued from previous page)

```
[23 rows x 29 columns]
```

[ ]:
```
df.columns
```

```
Index(['Unnamed: 0', 'tags', 'id', 'serial_number_id', 'file_name',
       'file_size', 'recording_length', 'recording_ts', 'created_ts',
       'modified_ts', 'device', 'gpsLocationFull', 'velocityRMSFull',
       'gpsSpeedFull', 'pressureMeanFull', 'samplePeakStartTime',
       'displacementRMSFull', 'psuedoVelocityPeakFull', 'accelerationPeakFull',
       'psdResultantOctave', 'samplePeakWindow', 'temperatureMeanFull',
       'accelerometerSampleRateFull', 'psdPeakOctaves', 'microphonoeRMSFull',
       'gyroscopeRMSFull', 'pvssResultantOctave', 'accelerationRMSFull',
       'psdResultant1Hz'],
      dtype='object')
```

There's a lot of data here! So we'll focus on just a handful of columns and convert the time in seconds to a datetime object.

[ ]:
```
df = df[['serial_number_id', 'file_name', 'file_size', 'recording_length', 'recording_ts
↪',
         'accelerationPeakFull', 'psuedoVelocityPeakFull', 'accelerationRMSFull',
         'velocityRMSFull', 'displacementRMSFull', 'pressureMeanFull',
↪'temperatureMeanFull']].copy()

df['recording_ts'] = pd.to_datetime(df['recording_ts'], unit='s')
df = df.sort_values(by=['recording_ts'], ascending=False)
df
```

```
    serial_number_id  ... temperatureMeanFull
11             11456  ...              24.175
10             11456  ...              24.180
22              9695  ...              26.410
7              11162  ...              24.545
17             11071  ...              21.889
8              10916  ...              18.874
2              10118  ...              23.172
20              9680  ...              26.031
19              9680  ...              32.202
18              9680  ...              33.452
21              9680  ...              25.616
12             11046  ...              28.832
5              11046  ...              29.061
14             10030  ...                 NaN
0               9695  ...              23.432
15              9695  ...                 NaN
4               9295  ...              17.820
1               9316  ...              20.133
16              7530  ...              26.989
6                  0  ...               9.538
13              5120  ...                 NaN
9               9874  ...              24.540
3              10309  ...              21.806
```

(continues on next page)

(continued from previous page)

```
[23 rows x 12 columns]
```

Filtering is made simple with boolean expressions that can be combined. There is also a method to sort_values by columns/series.

[ ]:
```
mask = df.recording_ts > pd.to_datetime('2021-01-01')
df[mask].sort_values(by=['serial_number_id'], ascending=False)
```

```
    serial_number_id  ... temperatureMeanFull
11              11456  ...               24.175
10              11456  ...               24.180
7               11162  ...               24.545
17              11071  ...               21.889
12              11046  ...               28.832
5               11046  ...               29.061
8               10916  ...               18.874
2               10118  ...               23.172
22               9695  ...               26.410
20               9680  ...               26.031
19               9680  ...               32.202
18               9680  ...               33.452
21               9680  ...               25.616

[13 rows x 12 columns]
```

[ ]:
```
mask = (df.recording_ts > pd.to_datetime('2021-01-01')) & (df.accelerationPeakFull > 100)
df[mask].sort_values(by=['accelerationPeakFull'], ascending=False)
```

```
    serial_number_id  ... temperatureMeanFull
21               9680  ...               25.616
18               9680  ...               33.452
19               9680  ...               32.202
20               9680  ...               26.031
11              11456  ...               24.175
10              11456  ...               24.180

[6 rows x 12 columns]
```

Another preview to plotly, but visualizing dataframes is made easy, even with mixed types.

[ ]:
```
px.scatter(df,
           x="recording_ts",
           y="accelerationRMSFull",
           size="recording_length",
           color="serial_number_id",
           hover_name="file_name",
           log_y=True,
           size_max=60).show()
```

Plotly automatically made my colors a colorbar because I specified it based on a numeric value. If instead I change the type to string and replot, we'll see discrete series for each device.

```
[ ]: df['device'] = df["serial_number_id"].astype(str)

px.scatter(df,
           x="recording_ts",
           y="accelerationRMSFull",
           size="recording_length",
           color="device",
           hover_name="file_name",
           log_y=True,
           size_max=60).show()
```

2.2.4 Preview of enDAQ Library

Installation

The code is live on GitHub, PyPI, and cleaner documentation is in process that will eventually live on a subdomain of endaq.com.

It can easily be installed with pip.

```
[ ]: !pip install -q endaq
```

```
  Installing build dependencies ... done
  Getting requirements to build wheel ... done
    Preparing wheel metadata ... done
  Installing build dependencies ... done
  Getting requirements to build wheel ... done
    Preparing wheel metadata ... done
  Installing build dependencies ... done
  Getting requirements to build wheel ... done
    Preparing wheel metadata ... done
  Installing build dependencies ... done
  Getting requirements to build wheel ... done
    Preparing wheel metadata ... done
     |▏ 92 kB 1.1 MB/s
     |▏ 81 kB 11.1 MB/s
  Building wheel for endaq (PEP 517) ... done
  Building wheel for endaq-calc (PEP 517) ... done
  Building wheel for endaq-cloud (PEP 517) ... done
  Building wheel for endaq-ide (PEP 517) ... done
```

We are using newer versions of some of these libraries so we need to update them then reset the runtime.

```
[ ]: ! python -m pip install -U -q numpy scipy plotly pandas
exit()
```

```
     |▏ 15.7 MB 417 kB/s
     |▏ 28.5 MB 1.3 MB/s
     |▏ 11.3 MB 28.8 MB/s
ERROR: pip's dependency resolver does not currently take into account all the packages␣
↪that are installed. This behaviour is the source of the following dependency conflicts.
tensorflow 2.6.0 requires numpy~=1.19.2, but you have numpy 1.21.2 which is incompatible.
google-colab 1.0.0 requires pandas~=1.1.0; python_version >= "3.0", but you have pandas␣
↪1.3.3 which is incompatible.
datascience 0.10.6 requires folium==0.2.1, but you have folium 0.8.3 which is␣
↪incompatible.
albumentations 0.1.12 requires imgaug<0.2.7,>=0.2.5, but you have imgaug 0.2.9 which is␣
↪incompatible.
```

(continues on next page)

(continued from previous page)

[ ]:
```
import numpy as np
import pandas as pd
import scipy

import plotly.express as px
import plotly.graph_objects as go

import endaq
```

Access Data from IDE

[ ]:
```
def get_doc_and_table(file_location, display_table=True, display_recorder_info=True):
  """Given an IDE file location, return the object and table summary"""
  dataset = endaq.ide.get_doc(file_location, quiet=True)
  table = endaq.ide.get_channel_table(dataset)

  if display_recorder_info:
    display(pd.Series(dataset.recorderInfo))
  if display_table:
    display(table)

  return dataset, table.data
```

[ ]:
```
[doc,table] = get_doc_and_table('https://info.endaq.com/hubfs/data/Bolted.ide')
```

```
CalibrationDate                          1583263964
CalibrationExpiry                        1614799964
CalibrationSerialNumber                        1999
RecorderTypeUID                                   1
RecorderSerial                                10118
HwRev                                             1
FwRev                                             5
FwRevStr                                     2.0.36
ProductName                                W8-E100D40
PartNumber                                 W8-E100D40
DateOfManufacture                          1588954110
UniqueChipID                     3782319969872279
McuType                    EFM32GG11B820F2048GL120
RecorderName                    Steve's Microphone
dtype: object
```

```
<pandas.io.formats.style.Styler at 0x7feec8874750>
```

[ ]:
```
def get_primary_accel(dataset, table, preferred_ch=None, time_mode='seconds'):
  """Given an IDE object and summary table, return a dataframe of the accelerometer with␣
↪the most samples unless a channel number is explicitely defined."""
  if preferred_ch:
    df = endaq.ide.to_pandas(doc.channels[preferred_ch],time_mode=time_mode)
```

(continues on next page)

(continued from previous page)

```
  else:
    table = table.sort_values('samples',ascending=False)
    channel = table.loc[table.type=='Acceleration','channel'].iloc[0].parent
    df = endaq.ide.to_pandas(channel,time_mode=time_mode)

  return df.drop(["Mic"], axis=1, errors="ignore")
```

[ ]:
```
df = get_primary_accel(doc, table)
df
```

```
           X (100g)  Y (100g)  Z (100g)
timestamp
0.381011  -1.268025 -2.824445 -2.262913
0.381061  -1.292439 -1.902800 -4.362554
0.381111  -1.438926 -1.164263 -4.740978
0.381161  -1.274128 -0.914015 -3.483635
0.381211  -0.730907 -1.115435 -1.927214
...             ...       ...       ...
29.667738 -1.036088 -1.817350 -2.207980
29.667788 -1.383993 -1.585412 -2.092012
29.667838 -1.438926 -1.933318 -1.512169
29.667888 -1.219196 -2.738994 -0.682078
29.667938 -0.865187 -3.526360  0.111391

[585728 rows x 3 columns]
```

PSD

We created a wrapper to SciPy's Welch's method to make the interface rely on dataframes.

[ ]:
```
df_psd = endaq.calc.psd.welch(df, bin_width=10)

fig = px.line(df_psd)

fig.update_layout(
    title='Power Spectral Density',
    xaxis_title="Frequency (Hz)",
    yaxis_title="Acceleration (g^2/Hz)",
    xaxis_type="log",
    yaxis_type="log",
    legend=dict(
        orientation="h",
        yanchor="top",
        y=-.2,
        xanchor="right",
        x=1
    )
  )

fig.show()
```

We also provide a function to convert a linear spaced PSD to an octave spaced one.

[ ]:
```
df_psd_oct = endaq.calc.psd.to_octave(df_psd, fstart=4, octave_bins=3)

fig = px.line(df_psd_oct)

fig.update_layout(
    title='Octave Spaced Power Spectral Density',
    xaxis_title="Frequency (Hz)",
    yaxis_title="Acceleration (g^2/Hz)",
    xaxis_type="log",
    yaxis_type="log",
    legend=dict(
        orientation="h",
        yanchor="top",
        y=-.2,
        xanchor="right",
        x=1
    )
  )

fig.show()
```

Shock

For some shock data, we'll use the motorcycle crash test data discussed in our blog post on pseudo velocity.

[ ]:
```
[doc,table] = get_doc_and_table('https://info.endaq.com/hubfs/data/Motorcycle-Car-Crash.
↪ide',
                                display_recorder_info=False,
                                display_table=False)
df = get_primary_accel(doc, table)
df
```

```
             X (500g)  Y (500g)  Z (500g)
timestamp
1024.088653  0.563998 -5.142559  0.575260
1024.088753  0.409059 -5.045741  0.847571
1024.088853  0.389691 -5.142559  1.275489
1024.088953  0.234751 -5.007014  0.905924
1024.089053  0.196017 -5.181287  0.633613
...               ...       ...       ...
1152.098232  0.796408 -5.452379  0.497457
1152.098332  0.835142 -5.607288  0.205695
1152.098432  1.048184 -5.549197  0.205695
1152.098532  1.145022 -5.936471  0.478006
1152.098632  1.280594 -5.859016  0.419654

[1280087 rows x 3 columns]
```

Get Peak Time & Plot

[ ]:
```
def get_peak_time(df,num):
  """Get a subset of a dataframe around the peak value"""
  peak_time = df.abs().max(axis=1).idxmax() #get the time at which the peak value occurs
  d_t = (df.index[-1]-df.index[0])/(len(df.index)-1) #find the average time step
  fs = 1/d_t #find the sampling rate

  num = num / 2 #total number of datapoints to plot (divide by 2 because it will be two␣
↪sided)
  df_peak = df[peak_time - num / fs : peak_time + num / fs ] #segment the dataframe to␣
↪be around that peak value

  return df_peak
```

[ ]:
```
df_peak = get_peak_time(df,5000)

fig = px.line(df_peak)
fig.update_layout(
    title="Time History around Peak",
    xaxis_title="Time (s)",
    yaxis_title="Acceleration (g)",
    template="plotly_white",
    legend=dict(
        orientation="h",
        yanchor="top",
        y=-.2,
        xanchor="right",
        x=1
    )
)
fig.show()
```

Filter & Plot

[ ]:
```
df_filtered = df - df.median() #For shock events it's sometimes best to avoid high pass␣
↪filters because they can introduce weird artifacts
df_filtered = endaq.calc.filters.butterworth(df_filtered, low_cutoff=None, high_
↪cutoff=150, half_order=3)
df_peak_filtered = get_peak_time(df_filtered,5000)

fig = px.line(df_peak_filtered)
fig.update_layout(
    title="Filtered Time History around Peak",
    xaxis_title="Time (s)",
    yaxis_title="Acceleration (g)",
    template="plotly_white",
    legend=dict(
        orientation="h",
        yanchor="top",
```

(continues on next page)

(continued from previous page)

```
        y=-.2,
        xanchor="right",
        x=1
    )
)
fig.show()
```

Integrate to Velocity & Displacement

[ ]:
```
[df_accel, df_vel, df_disp] = endaq.calc.integrate.integrals(df_peak_filtered, n=2,␣
↪highpass_cutoff=None, tukey_percent=0)

df_vel = df_vel-df_vel.iloc[0] #forced the starting velocity to 0
df_vel = df_vel*9.81*39.37 #convert to in/s
```

[ ]:
```
fig = px.line(df_vel)
fig.update_layout(
    title="Integrated Velocity Time History",
    xaxis_title="Time (s)",
    yaxis_title="Velocity (in/s)",
    template="plotly_dark",
    legend=dict(
        orientation="h",
        yanchor="top",
        y=-.2,
        xanchor="right",
        x=1
    )
)
fig.show()
```

Shock Response Spectrum

[ ]:
```
freqs=2 ** np.arange(0, 12, 1/12)

pvss = endaq.calc.shock.pseudo_velocity(df_peak-df_peak.median(), freqs=freqs, damp=0.05)
pvss = pvss*9.81*39.37 #convert to in/s
```

[ ]:
```
fig = px.line(pvss)
fig.update_layout(
    title='Psuedo Velocity Shock Spectrum (PVSS)',
    xaxis_title="Natural Frequency (Hz)",
    yaxis_title="Psuedo Velocity (in/s)",
    template="plotly_dark",
    xaxis_type="log",
    yaxis_type="log",
    legend=dict(
```

(continues on next page)

(continued from previous page)

```
        orientation="h",
        yanchor="top",
        y=-.2,
        xanchor="right",
        x=1
    )
)
```

2.3 Introduction to Plotly

2.3.1 Introduction

This notebook and accompanying webinar was developed and released by the enDAQ team. This is the third "chapter" of our series on *Python for Mechanical Engineers*: 1. Get Started with Python * Blog: Get Started with Python: Why and How Mechanical Engineers Should Make the Switch 2. Introduction to Numpy & Pandas for Data Analysis 3. **Introduction to Plotly for Plotting Data** * Watch Recording of This 4. Introduction of the enDAQ Library

To sign up for future webinars and watch previous ones, visit our webinars page.

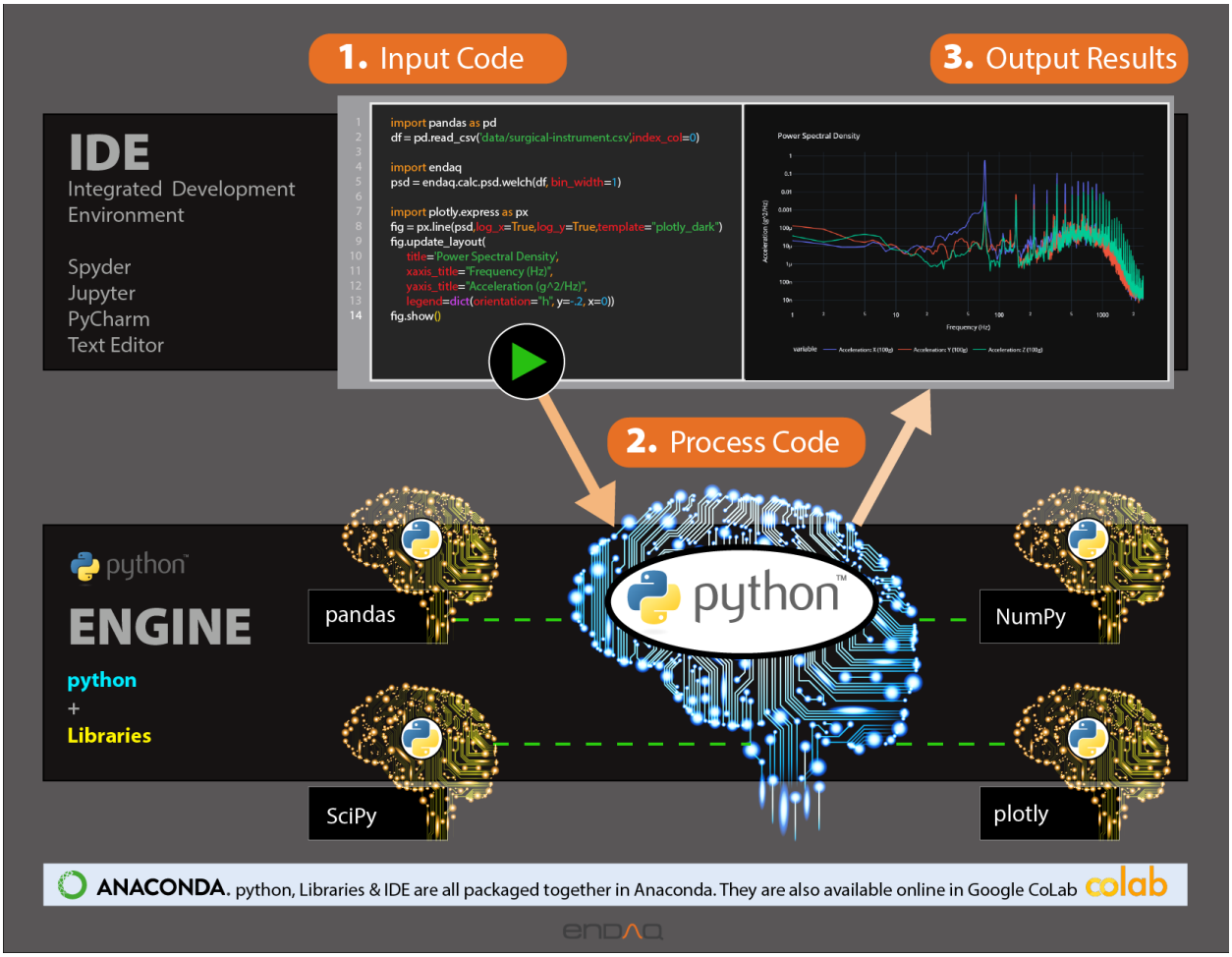
Recap of Python Introduction

1. Python is popular for good reason
2. There are many ways to interact with Python

- Here we are in Google Colab based on Jupyter Notebooks

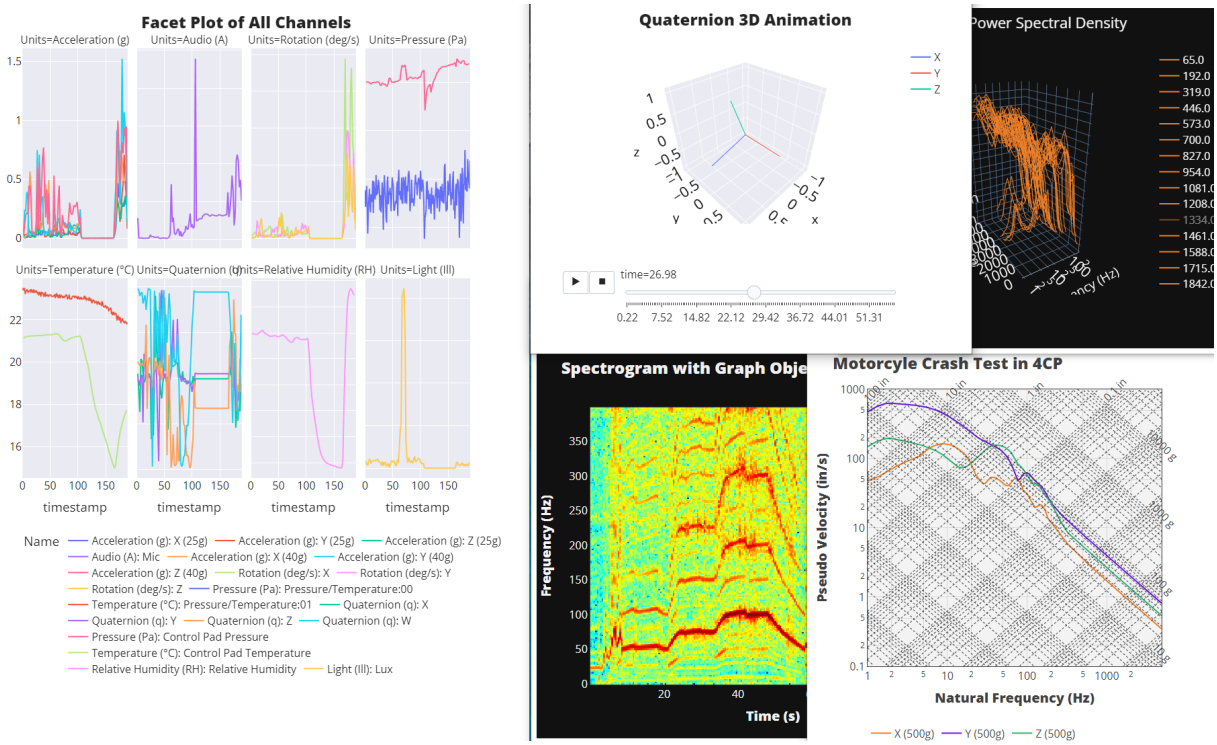
3. There are many open source libraries to use

- Today we are covering Plotly for plotting



2.3.2 Why Do We Plot?

- To Understand Relationships, Make an Observation
  - Interactivity Matters!
- To Share & Present
  - Beautiful Matters!



2.3.3 Overview of Python Plotting Libraries

Like everything in Python, there are a few options for plotting data! We'll focus on Plotly but quickly cover: - Matplotlib - Seaborn - ggplot - bokeh - Plotly

Data Source

For some shock data, we'll use the motorcycle crash test data discussed in our blog post on pseudo velocity. This was filtered with a 150 low pass filter first to clean the signal using the endaq library (which uses SciPy under the hood).

[ ]:
```
import pandas as pd

df = pd.read_csv('https://info.endaq.com/hubfs/data/motorcycle-crash.csv',index_col=0)
```

[ ]:
```
df
```

```
           X (500g)  Y (500g)  Z (500g)
timestamp
0.00000   -0.055582  0.033337  0.032217
0.00010   -0.056731  0.041513  0.032720
0.00020   -0.056972  0.049888  0.033106
0.00030   -0.056273  0.058399  0.033419
0.00040   -0.054605  0.066982  0.033706
...             ...       ...       ...
0.19952    0.383610  0.117662 -0.292337
0.19962    0.411270  0.135782 -0.275515
0.19972    0.438324  0.153832 -0.258591
0.19982    0.464640  0.171657 -0.241783
```

(continues on next page)

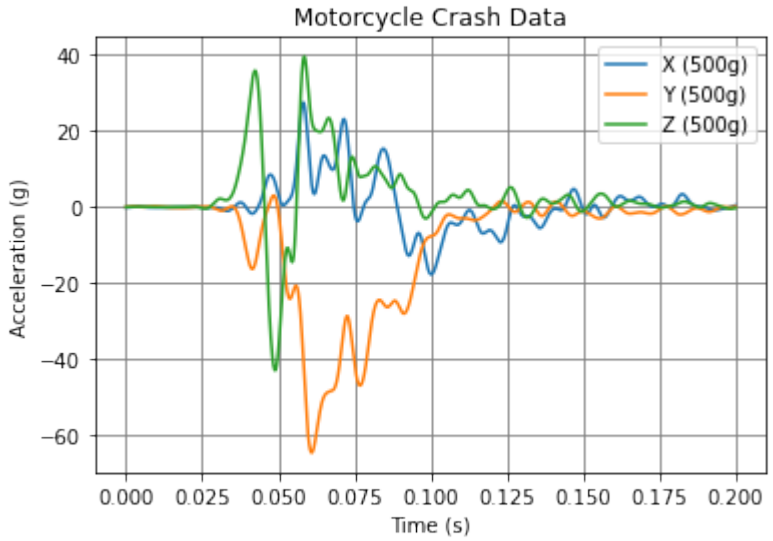
(continued from previous page)

```
0.19992    0.490098  0.189112 -0.225302

[2000 rows x 3 columns]
```

Matplotlib

The most popular plotting library and the default go-to. But… there is a new game in town!

[ ]:
```
import matplotlib.pyplot as plt

plt.plot(df)
plt.title('Motorcycle Crash Data')
plt.ylabel('Acceleration (g)')
plt.xlabel('Time (s)')
plt.legend(df.columns)
plt.grid(color='grey')
plt.show()
```

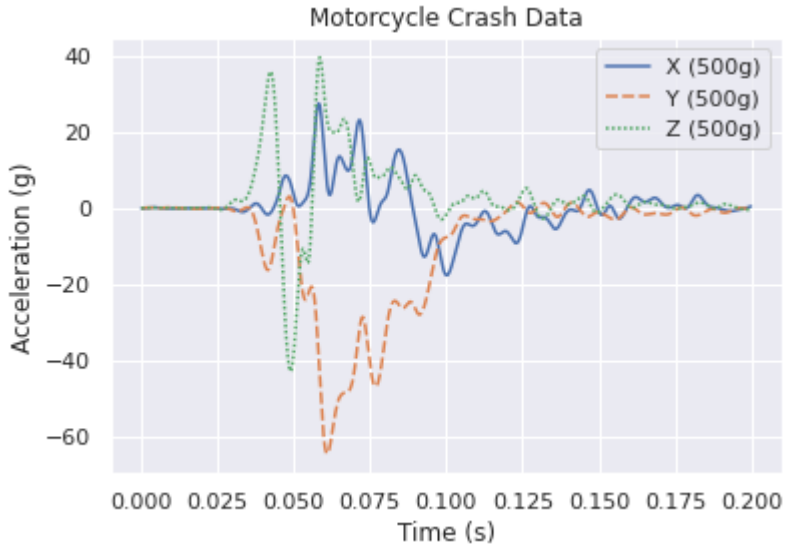


Seaborn

A wrapper around Matplotlib to simplify the interface, beautify the plots, and support more stats-based analysis. Here is their documentation specifically on aesthetics, they care too!

[ ]:
```
import seaborn as sns

sns.set_theme()

p = sns.lineplot(data=df)
p.set_title('Motorcycle Crash Data')
p.set_ylabel('Acceleration (g)')
p.set_xlabel('Time (s)')
```

```
Text(0.5, 0, 'Time (s)')
```

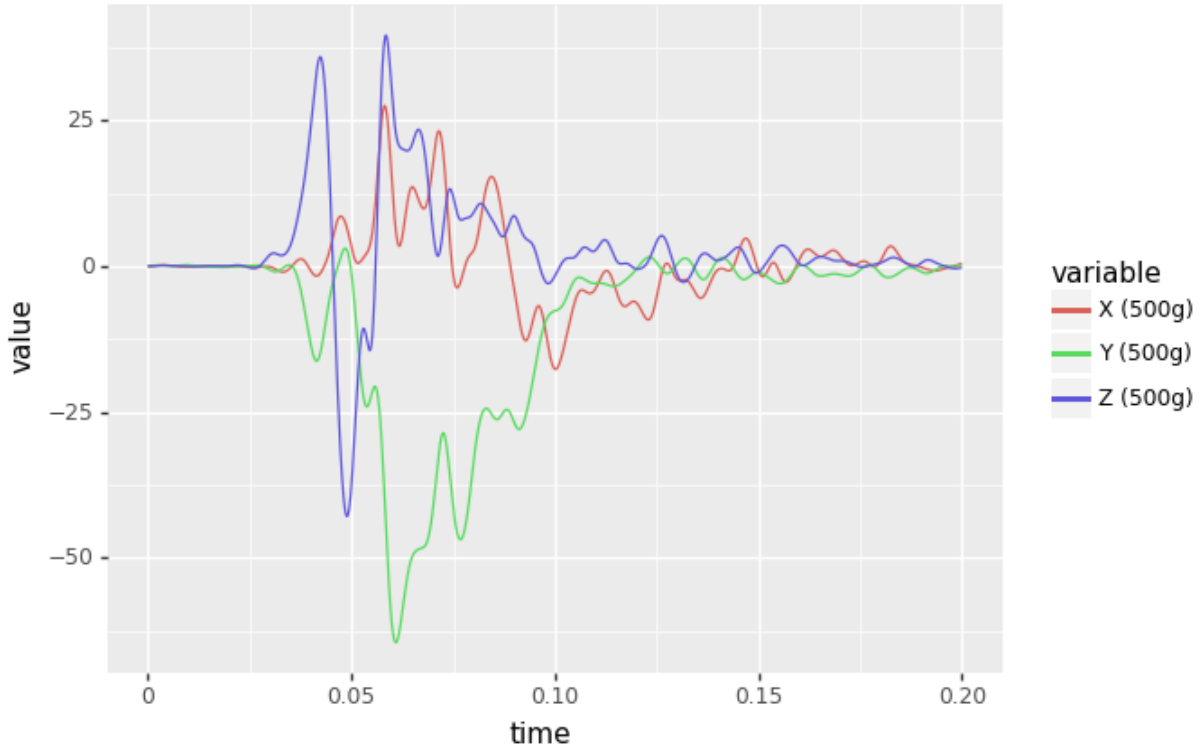


ggplot

Introduces a "grammer of graphics" logic to plotting data which allows explicit mapping of data to the visual representation. This is something plotly express excels at.

[ ]:
```
from plotnine import ggplot, aes, geom_line

df_ggplot = df.copy()
df_ggplot['time'] = df.index
df_ggplot = pd.melt(df_ggplot, id_vars='time')
df_ggplot

(
    ggplot(df_ggplot)  # What data to use
    + aes(x="time", y='value',color='variable')  # What variable to use
    + geom_line()  # Geometric object to use for drawing
)
```

```
/usr/local/lib/python3.7/dist-packages/plotnine/utils.py:1246: FutureWarning: is_
↪categorical is deprecated and will be removed in a future version.  Use is_categorical_
↪dtype instead
  if pdtypes.is_categorical(arr):
```




```
<ggplot: (8785480354521)>
```

Bokeh

Finally, we have interactivity!

[ ]:
```
from bokeh.plotting import figure, show
from bokeh.io import output_notebook

output_notebook()

p = figure(title='Motorcycle Crash Data',
           x_axis_label='Time (s)',
           y_axis_label='Acceleration (g)')

colors = ['red','green','blue']
for c,color in zip(df.columns,colors):
  p.line(df.index, df[c], legend_label = c, line_color = color, line_width = 2)

show(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_load.v0+json

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

Plotly

Interactive, beautiful, easy!

[ ]:
```
!pip install -U -q plotly
```

```
     || 23.9 MB 1.5 MB/s
```

[ ]:
```
import plotly.express as px
import plotly.io as pio; pio.renderers.default = "iframe"

fig = px.line(df)
fig.update_layout(
    title_text = 'Motorcycle Crash Data',
    xaxis_title_text = "Time (s)",
    yaxis_title_text = "Acceleration (g)")
fig.show()
```

2.3.4 How Does Plotly Work?

There are three main components to how/why Plotly works: 1. Python Library 2. Figure Objects 3. JavaScript Library

To illustrate this relationship, let's make a Plotly Treemap!

[ ]:
```
df_tree = pd.DataFrame({
    'names': ["Plotly","Python", "Figure Object", "JavaScript", "Express", "Graph Objects
↪","Data","Layout",'Frames',"Plotly.js"],
    'parents' : ["", "Plotly", "Plotly", "Plotly", "Python",'Express',"Figure Object",
↪"Figure Object","Figure Object","JavaScript"]
})
```

[ ]:
```
fig = px.treemap(
    names = df_tree.names,
    parents = df_tree.parents
)
fig.update_traces(root_color="lightgrey")
fig.update_layout(
  font_family='Open Sans',
  font_size=32)
fig.show()
```

Plotly Express was introduced in May 2019 and is a GAME CHANGER! Here is the introduction article.

✨ Introducing Plotly Express ✨

plotly Follow

Mar 20, 2019 · 11 min read

Plotly Express is a new high-level Python visualization library: it's a wrapper for Plotly.py that exposes a simple syntax for complex charts. Inspired by Seaborn and ggplot2, it was specifically designed to have a terse, consistent and easy-to-learn API: with just a single import, you can make richly interactive plots in just a single function call, including faceting, maps, animations, and trendlines. It comes with on-board datasets, color scales and themes, and just like Plotly.py, Plotly Express is *totally free*: with its permissive open-source MIT license, you can use it however you like (yes, even in commercial products!). Best of all, Plotly Express is fully compatible with the rest of Plotly ecosystem: use it in your Dash apps, export your figures to almost any file format using Orca, or edit them in a GUI with the JupyterLab Chart Editor!

If you're the **TL;DR** type, just `pip install plotly` and head on over to our walkthrough notebook or gallery or reference documentation to start playing around, otherwise read on for an overview of what makes Plotly Express special. If you have any feedback or want to check out the code, it's all up on Github.

2.3.5 Shoutout to Plotly's Docs

Before we proceed, I want to really stress how good the docs are from Plotly. They have a TON of examples, and very good documentation. We'll be focusing on Plotly Express.

They also have their own community forum that is pretty rich with examples and people helping each other out.

2.3.6 Styling Figures

Example Data

Here, to mix it up, we'll calculate a shock response spectrum from the acceleration data to then be plotting on a log log scale to help show how to style that.

[ ]:
```
!pip install -q endaq
import endaq
import numpy as np
```

[ ]:
```
def get_log_freqs(df,init_freq=1,bins_per_octave=12):
  """Given a timebased dataframe, return a log spaced frequency array up to the Nyquist␣
↪frequency"""
  fs = (df.shape[0]-1)/(df.index[-1]-df.index[0])
  return 2 ** np.arange(np.log2(init_freq),
                        np.log2(fs/2),
                        1/bins_per_octave)
```

[ ]:
```
df_pvss = endaq.calc.shock.pseudo_velocity(df,
                                           get_log_freqs(df,init_freq=1,bins_per_
↪octave=12),
                                           damp=0.05, two_sided=False)
df_pvss = df_pvss*9.81*39.37 #convert to in/s
```

[ ]:
```
df_pvss
```

```
                 X (500g)    Y (500g)    Z (500g)
frequency (Hz)
1.000000        47.804683  473.402407  148.658683
1.059463        49.240508  493.307994  154.812945
1.122462        50.532563  513.039780  160.893921
1.189207        51.637517  532.374454  166.829370
1.259921        52.551121  551.046158  172.533775
...                   ...         ...         ...
3866.109185      0.437852    1.025865    0.681822
4096.000000      0.413256    0.968276    0.643536
4339.560834      0.390047    0.913923    0.607404
4597.604550      0.368145    0.862623    0.573305
4870.992343      0.347478    0.814205    0.541124

[148 rows x 3 columns]
```

Manually Defining the Theme

First let's start by plotting with the standard theme, modifying only the titles. We'll also define the scale on the x and y axis to be a log scale.

[ ]:
```
fig = px.line(df_pvss)
fig.update_layout(
    title_text = 'Motorcycle Crash Data',
    xaxis_title_text = "Natural Frequency (Hz)",
    yaxis_title_text = "Pseudo Velocity (in/s)",
    xaxis_type = "log",
    yaxis_type = "log")
fig.show()
```

Now let's get crazy and customize all the "common" settings. But note that there are a LOT of different parameters that can be explicitely defined. Remember, Plotly has very thorough documentation, so check it out! * Figure Layout * X Axis * Y Axis

You will need to download Open Sans if you like the font like me too! To pick colors, I suggest Color Hex.

[ ]:
```
fig = px.line(df_pvss)
fig.update_layout(
  font_family='Open Sans',
  font_size=16,
  font_color='#404041',

  title_text = 'Motorcycle Crash Data',
  title_font_family = 'Showcard Gothic',
  title_font_size = 32,
  title_font_color = '#e77025',
  title_x = 0.5,

  xaxis_title_text = "Natural Frequency (Hz)",
  xaxis_title_font_family = 'Algerian',
  xaxis_title_font_size = 24,
  xaxis_title_font_color = '#7f3f98',
  xaxis_type = 'log',

  yaxis_title_text ="Pseudo Velocity (in/s)",
  yaxis_title_font_family = 'Playbill',
  yaxis_title_font_size = 24,
  yaxis_title_font_color = '#be1e2d',
  yaxis_type = 'log',

  legend_bgcolor = 'yellow',
  legend_title_text = 'Legend',
  legend_title_font_size = 24,
  legend_orientation = 'v',
  legend_y = 1.0,
  legend_yanchor = 'top',
  legend_x = 1.0,
  legend_xanchor = 'right',

  plot_bgcolor = '#f3f3f3',
```

(continues on next page)

(continued from previous page)

```
  width = 800,
  height = 600)
fig.show()
```

One thing that Plotly does which is REALLY cool is that they use "magic underscore notation" which means that this: ~~~ yaxis_title_font_family = 'Open Sans ExtraBold' ~~~

is the equivalent of: ~~~ {'yaxis': {'title': {'font': {'family': 'Open Sans ExtraBold'} } } } ~~~

Now I am particular about my plots and how they look, I think aesthetics matter! So I can create a few custom themes that can be added to figures when making them.

[ ]:
```
template_light = dict(
  template="presentation",

  font_family='Open Sans',
  font_size=16,
  font_color='#404041',

  title_font_family = 'Open Sans ExtraBold',
  title_font_size = 24,
  title_x = 0.5,

  xaxis_title_font_family = 'Open Sans ExtraBold',
  xaxis_title_font_size = 20,

  yaxis_title_font_family = 'Open Sans ExtraBold',
  yaxis_title_font_size = 20,

  legend_title='',
  legend_orientation='h',
  legend_y = -0.2,

  plot_bgcolor = '#f3f3f3',
  yaxis_gridcolor = '#dad9d8',
  yaxis_linecolor = '#404041',
  yaxis_mirror = True,
  xaxis_gridcolor = '#dad9d8',
  xaxis_linecolor = '#404041',
  xaxis_mirror = True,
  )

template_dark = template_light.copy()
template_dark['template'] = "plotly_dark"
template_dark['font_color'] = '#f3f3f3'
template_dark['plot_bgcolor'] = "#111111"
template_dark['yaxis_linecolor'] = "#404041"
template_dark['xaxis_linecolor'] = "#404041"
template_dark['yaxis_gridcolor'] = "#404041"
template_dark['xaxis_gridcolor'] = "#404041"
```

I also tend to make a lot of similar plots, so it can be helpful to define some axes labels and types as variables.

[ ]:
```
template_pvss = dict(
  xaxis_title_text = "Natural Frequency (Hz)",
  xaxis_type = 'log',
  yaxis_title_text ="Pseudo Velocity (in/s)",
  yaxis_type = 'log'
)

template_psd = dict(
  xaxis_title_text = "Frequency (Hz)",
  xaxis_type = 'log',
  yaxis_title_text ="Acceleration (g^2/Hz)",
  yaxis_type = 'log'
  )

template_accel = dict(
  xaxis_title_text = "Time (s)",
  yaxis_title_text ="Acceleration (g)",
  )
```

[ ]:
```
fig = px.line(df_pvss)
fig.update_layout(
  {**template_light, **template_pvss},
  title_text = 'Custom Light Theme')
fig.show()
```

[ ]:
```
fig = px.line(df_pvss)
fig.update_layout(
  {**template_dark, **template_pvss},
  title_text = 'Custom Dark Theme')
fig.show()
```

Themes

As you may have noticed, I used some templates in my custom theme. There are a few to pick from and you can make your own as well. This is well documented with examples on Plotly's website.

[ ]:
```
themes = ["plotly", "plotly_white", "plotly_dark", "ggplot2", "seaborn", "simple_white",
↪'presentation', "none"]

for theme in themes:
  fig = px.line(df_pvss)
  fig.update_layout(
    template_pvss,
    template = theme,
    title_text = theme,
    )
  fig.show()
```

Colors

Plotly also supports a whole range of different color schemes you can implement. There are a lot of built in ones and remember, great documentation!

[ ]:
```
px.colors.qualitative.swatches().show()
```

[ ]:
```
px.colors.qualitative.Alphabet
```

```
['#AA0DFE',
 '#3283FE',
 '#85660D',
 '#782AB6',
 '#565656',
 '#1C8356',
 '#16FF32',
 '#F7E1A0',
 '#E2E2E2',
 '#1CBE4F',
 '#C4451C',
 '#DEA0FD',
 '#FE00FA',
 '#325A9B',
 '#FEAF16',
 '#F8A19F',
 '#90AD1C',
 '#F6222E',
 '#1CFFCE',
 '#2ED9FF',
 '#B10DA1',
 '#C075A6',
 '#FC1CBF',
 '#B00068',
 '#FBE426',
 '#FA0087']
```

[ ]:
```
px.colors.sequential.swatches_continuous().show()
```

[ ]:
```
px.colors.sequential.Blackbody
```

```
['rgb(0,0,0)',
 'rgb(230,0,0)',
 'rgb(230,210,0)',
 'rgb(255,255,255)',
 'rgb(160,200,255)']
```

Here I'm going to make my own swatch of some kind with a custom color scale.

```
[ ]: colors = ['#EE7F27', '#6914F0', '#2DB473', '#D72D2D', '#3764FF', '#FAC85F','#27eec0','
     ↪#b42d4d','#82d72d','#e35ffa']
     fig = px.bar(x=np.arange(10),
                  y=np.zeros(10)+1,
                  color=colors,
                  color_discrete_sequence=colors,
                  height=200)
     fig.update_layout(template_dark,
                       font_color="#111111",
                       legend_font_color="white")
     fig.show()
```

When creating a figure you can specify the color sequence you'd like as a parameter. You can use a custom list like what I'm showing here or you can call one of the `px.colors.qualitative.` lists.

```
[ ]: fig = px.line(df_pvss,
                   color_discrete_sequence=colors)

     fig.update_layout(
       {**template_dark, **template_pvss},
       title_text = 'Custom Colors & Dashes')

     fig.show()
```

2.3.7 Saving Figures

To save static images, first you need to install kaleido.

```
[ ]: !pip install -U kaleido
```

```
Collecting kaleido
  Downloading kaleido-0.2.1-py2.py3-none-manylinux1_x86_64.whl (79.9 MB)
     || 79.9 MB 85 kB/s
Installing collected packages: kaleido
Successfully installed kaleido-0.2.1
```

In colab only you'll also need to do this install.

```
[ ]: !wget https://github.com/plotly/orca/releases/download/v1.2.1/orca-1.2.1-x86_64.AppImage␣
     ↪-O /usr/local/bin/orca
     !chmod +x /usr/local/bin/orca
     !apt-get install xvfb libgtk2.0-0 libgconf-2-4
```

Now saving an image is easy and they support a few types. More information and options available in their docs.

```
[ ]: fig.write_image("fig.png")
     fig.write_image("fig.jpeg")
     fig.write_image("fig.svg")
     fig.write_image("fig.pdf")
```

You can also keep the interactivity by downloading an HTML file, either with or without the plotly javascript library. You'll need the library if you want to view the plots when you don't have internet access. For more, see their docs on saving HTML files.

[ ]:
```
fig.write_html('fig-full.html')
fig.write_html('fig-small.html',full_html=False,include_plotlyjs='cdn')
```

2.3.8 Plot Types

Remember Plotly has GREAT documentation, and LOTS of plot types. Below is a GIF of their main page on all the different plot types.

Now here are the gallery of plots all in plotly.express meaning they are super easy to create! Again, check out the docs.

Now we'll go through a few relevant examples for mechanical engineers and vibration analysis.

Bubble

[ ]:
```
df_table = pd.read_csv('https://info.endaq.com/hubfs/data/endaq-cloud-table.csv')
df_table = df_table[['serial_number_id', 'file_name', 'file_size', 'recording_length',
↪'recording_ts',
        'accelerationPeakFull', 'psuedoVelocityPeakFull', 'accelerationRMSFull',
        'velocityRMSFull', 'displacementRMSFull', 'pressureMeanFull',
↪'temperatureMeanFull']].copy()

df_table['recording_ts'] = pd.to_datetime(df_table['recording_ts'], unit='s')
df_table = df_table.sort_values(by=['recording_ts'], ascending=False)
df_table
```

```
    serial_number_id  ... temperatureMeanFull
11             11456  ...              24.175
10             11456  ...              24.180
22              9695  ...              26.410
7              11162  ...              24.545
17             11071  ...              21.889
8              10916  ...              18.874
2              10118  ...              23.172
20              9680  ...              26.031
19              9680  ...              32.202
18              9680  ...              33.452
21              9680  ...              25.616
12             11046  ...              28.832
5              11046  ...              29.061
14             10030  ...                 NaN
0               9695  ...              23.432
15              9695  ...                 NaN
4               9295  ...              17.820
1               9316  ...              20.133
16              7530  ...              26.989
6                  0  ...               9.538
13              5120  ...                 NaN
9               9874  ...              24.540
3              10309  ...              21.806
```

(continues on next page)

(continued from previous page)

```
[23 rows x 12 columns]
```

[ ]:
```
fig = px.scatter(df_table,
           x="recording_ts",
           y="accelerationRMSFull",
           size="recording_length",
           color="serial_number_id",
           hover_name="file_name",
           log_y=True,
           size_max=60)

fig.update_layout(
  template_dark,
  title_text ='Scatter Plot with Numeric "Color"',
  xaxis_title_text ="Date of Recording",
  yaxis_title_text ="Acceleration RMS (g)"
)

fig.show()
```

[ ]:
```
df_table['device'] = df_table["serial_number_id"].astype(str)

fig = px.scatter(df_table,
           x="accelerationPeakFull",
           y="velocityRMSFull",
           size="recording_length",
           color="device",
           color_discrete_sequence=px.colors.qualitative.Light24, #I'll want to use a␣
↪list with as many discrete values as I can
           hover_name="file_name",
           log_y=True,
           log_x=True,
           size_max=60)

fig.update_layout(
  template_dark,
  title_text ='Scatter Plot with Text "Color"',
  xaxis_title_text ="Acceleration RMS (g)",
  yaxis_title_text ="Velocity RMS (mm/s)"
)

fig.show()
```

Drop Downs to Define X & Y Axis Columns

This function (thanks Sam!) highlights another layer of interactivity in Plotly.

[ ]:
```
def get_correlation_figure_please(merged_df):
    cols = [col for col, t in zip(merged_df.columns, merged_df.dtypes) if t != object]

    start_dropdown_indices = [0, 0]

    # Create the scatter plot of the initially selected variables
    fig = px.scatter(
        merged_df,
        x=cols[start_dropdown_indices[0]],
        y=cols[start_dropdown_indices[1]]
    )

    # Create the drop-down menus which will be used to choose the desired file␣
↪characteristics for comparison
    drop_downs = []
    for axis in ['x', 'y']:
        drop_downs.append([
            dict(
                method = 'update',
                args = [
                    {axis : [merged_df[cols[k]]]},
                    {'%saxis.title.text'%axis: cols[k]},
                    # {'color':[merged_df['serial_number_id']],'color_discrete_map':SERIALS_
↪TO_INDEX},
                ],
                label = cols[k]) for k in range(len(cols))
            ])

    # Sets up various apsects of the Plotly figure that is currently being produced. ␣
↪This ranges from
    # aethetic things, to setting the dropdown menues as part of the figure
    fig.update_layout(
        title_x=0.4,
        updatemenus=[{
            'active': start_j,
            'buttons': drop_down,
            'x': 1.125,
            'y': y_height,
            'xanchor': 'left',
            'yanchor': 'top',
          } for drop_down, start_j, y_height in zip(drop_downs, start_dropdown_indices,␣
↪[1, .85])])

    return fig
```

[ ]:
```
fig = get_correlation_figure_please(df_table)
fig.update_layout(
  template_light,
  title_text='Selectable Drop Downs',
```

(continues on next page)

(continued from previous page)

```
)
fig.show()
```

Map

[ ]:
```
map_data = pd.read_csv('https://info.endaq.com/hubfs/data/mide-map-gps-data.csv')
map_data
```

```
        timestamp   Latitude  Longitude          Date  Ground Speed
0     4436.395385  42.369258 -71.019592  1.619789e+09         1.014
1     4437.395355  42.369257 -71.019600  1.619789e+09         0.511
2     4438.395324  42.369257 -71.019603  1.619789e+09         0.184
3     4439.395294  42.369256 -71.019608  1.619789e+09         0.413
4     4440.395263  42.369259 -71.019605  1.619789e+09         0.341
...           ...        ...        ...           ...           ...
1905  6343.337073  42.496331 -71.139950  1.619791e+09         0.170
1906  6344.337041  42.496331 -71.139949  1.619791e+09         0.025
1907  6345.337008  42.496330 -71.139949  1.619791e+09         0.166
1908  6346.336976  42.496329 -71.139950  1.619791e+09         0.117
1909  6347.336944  42.496330 -71.139952  1.619791e+09         0.100

[1910 rows x 5 columns]
```

You need a mapbox token here, which is free up to 50,000 "loads" per month.

[ ]:
```
import os
from dotenv import load_dotenv

load_dotenv()

MAPBOX_ACCESS_TOKEN = os.getenv('MAPBOX_ACCESS_TOKEN')
```

[ ]:
```
px.set_mapbox_access_token(MAPBOX_ACCESS_TOKEN)
fig = px.scatter_mapbox(map_data,
                        lat="Latitude",
                        lon="Longitude",
                        color="Ground Speed",
                        zoom=10)
fig.update_layout(
  template_dark,
  title_text ='Mide to Airport Drive',
  coloraxis_showscale=False)
fig.show()
```

3D Line

[ ]:
```
df_rolling_psd = pd.read_csv('https://info.endaq.com/hubfs/data/rolling-psd.csv',index_
↪col=0)
df_rolling_psd
```

```
    frequency (Hz)       X (40g)  ...     Resultant    Time
0         1.000000  1.021500e-04  ...  3.258877e-04    65.0
1         1.259921  1.267597e-04  ...  3.274643e-04    65.0
2         1.587401  1.159243e-04  ...  2.957754e-04    65.0
3         2.000000  2.432255e-04  ...  4.165131e-04    65.0
4         2.519842  1.447308e-04  ...  2.210021e-04    65.0
..             ...           ...  ...           ...     ...
20      101.593667  3.904692e-07  ...  5.634277e-06  6285.0
21      128.000000  1.520778e-07  ...  7.528400e-07  6285.0
22      161.269894  6.667854e-08  ...  1.335149e-07  6285.0
23      203.187335  1.048538e-08  ...  2.896009e-08  6285.0
24      256.000000  6.308070e-09  ...  1.567104e-08  6285.0

[1250 rows x 6 columns]
```

[ ]:
```
fig = px.line_3d(df_rolling_psd,
                 x='frequency (Hz)',
                 y='Time',
                 z='Resultant',
                 color='Time',
                 log_x=True,
                 log_z=True,
                 color_discrete_sequence=['#EE7F27']
           )

fig.update_layout(
  template_dark,
  title_text ='3D PSD',
  legend_orientation='v',
  legend_y = 0.0,
  )
fig.show()
```

Animation

[ ]:
```
fig = px.line(
    df_rolling_psd,
    x="frequency (Hz)",
    y="Resultant",
    animation_frame="Time",
    color_discrete_sequence=['#EE7F27']
)

fig.update_layout(
  {**template_dark, **template_psd},
```

(continues on next page)

(continued from previous page)

```
  title_text = 'Animation of Moving PSD',
  legend_orientation = 'v',
  legend_y = 1.0,
  legend_yanchor = 'top',
  legend_x = 1.0,
  legend_xanchor = 'right')

fig.show()
```

[ ]:
```
import plotly.graph_objects as go

def quantile_from_rolling_psd(rolling_psd,column,quantile):
  """Get the respective quantile of the defined column for all frequencies in the␣
↪rolling psd"""
  freqs = rolling_psd["frequency (Hz)"].drop_duplicates().to_numpy()
  df = pd.Series(index=freqs,name='Quantile',dtype='float64')
  df['Quantile'] = 0
  for f in freqs:
    df_quant = rolling_psd[rolling_psd['frequency (Hz)'] == f].quantile(quantile)
    df.loc[f,'Quantile'] = df_quant[column]
  return df

def add_quantile(rolling_psd,column,quantile,name,color,dash):
  quant = quantile_from_rolling_psd(rolling_psd,column,quantile)
  fig.add_trace(
      go.Scatter(
          x=quant.index,
          y=quant.values,
          name=name,
          line=dict(color=color, dash=dash),
      )
  )

add_quantile(df_rolling_psd,'Resultant',1.0,"Max","#6914F0","solid")
add_quantile(df_rolling_psd,'Resultant',0.5,"Median","#6914F0","dash")
add_quantile(df_rolling_psd,'Resultant',0.0,"Min","#6914F0","solid")
fig.write_html('psd-animation.html',full_html=False,include_plotlyjs='cdn')
fig.show()
```

Large Time Series

The one bad thing about plotly is that it makes your browser load all the data points into memory. It therefore can crash if you are trying to plot too many data points.

[ ]:
```
df_vibe = pd.read_csv('https://info.endaq.com/hubfs/data/motorcycle-vibration-moving-
↪frequency.csv',index_col=0)
df_vibe = df_vibe - df_vibe.median()
df_vibe
```

```
            X (40g)   Y (40g)   Z (40g)
timestamp
0.402069  -0.390691  0.017191 -0.847865
0.402318  -0.172913  0.071447 -0.826096
0.402567   1.089847  1.284786  0.356687
0.402816   0.296012  1.093866 -0.054382
0.403066  -0.111124  0.318515 -0.624797
...             ...       ...       ...
99.726203  0.260407 -0.161978 -0.047653
99.726453 -0.552108 -0.073261 -0.077179
99.726703 -0.280525 -0.001577 -0.057706
99.726953  0.819221 -0.196046  0.157842
99.727203  0.523529 -0.314730  0.136786

[397386 rows x 3 columns]
```

Render an Image

To get around this we can render the figure as an image (see docs for more information).

[ ]:
```
fig = px.line(df_vibe)

fig.update_layout(
  {**template_dark, **template_accel},
  title_text = 'Large Time Series, Render as Image')

fig.show(renderer="svg")
```

The other issue I have is that the order in which we plot matters with how we view and understand the data which isn't good!

[ ]:
```
fig = px.line(df_vibe[df_vibe.columns[::-1]],
              color_discrete_sequence=['#00CC96','#EF553B','#636EFA'])

fig.update_layout(
  {**template_dark, **template_accel},
  title_text = 'Large Time Series, Render as Image')

fig.show(renderer="svg")
```

Moving Metrics

One option is to plot the moving max, min, and mean or standard deviation.

[ ]:
```
def add_metric(fig,df_rolling,colors,dash,names,label):
  for var,color in zip(names,colors):
    fig.add_trace(go.Scatter(
        x=df_rolling.index,
        y=df_rolling[var].to_numpy(),
```

(continues on next page)

(continued from previous page)

```
        mode="lines",
        line = dict(color=color, width=1,dash=dash),
        name=var+label
    ))
  return fig

def moving_window_plot(df,n_steps=100):
  """Generate a plotly figure with the moving peak and standard deviation."""
  n = int(df.shape[0]/n_steps) #number of data points to use in windowing

  df_rolling_max = df.rolling(n).max().iloc[::n]
  df_rolling_min = df.rolling(n).min().iloc[::n]
  df_rolling_rms = df.rolling(n).std().iloc[::n]

  df_rolling_max.columns = [col+': Max' for col in df.columns]
  fig = px.line(df_rolling_max,
                color_discrete_sequence=colors)

  fig = add_metric(fig,df_rolling_min,colors,None,df.columns,': Min')
  fig = add_metric(fig,df_rolling_rms,colors,'dash',df.columns,': STD')

  return fig
```

[ ]:
```
fig = moving_window_plot(df_vibe,1000)

fig.update_layout(
  {**template_dark, **template_accel},
  title_text = 'Rolling Peak & RMS')

fig.show()
```

Using Datashader

There is a library for this (there is one for everything in Python!) to create large datasets into pixels for easy visualization called Datashader.

I modified the code from their example on time series.

[ ]:
```
!pip install -q datashader
```

```
     || 15.8 MB 1.3 kB/s
     || 76 kB 3.5 MB/s
     || 786 kB 38.9 MB/s
     || 125 kB 47.3 MB/s
     || 779 kB 38.6 MB/s
     || 778 kB 51.1 MB/s
     || 776 kB 60.8 MB/s
     || 769 kB 59.3 MB/s
     || 766 kB 51.8 MB/s
     || 1.0 MB 27.9 MB/s
     || 722 kB 57.1 MB/s
```

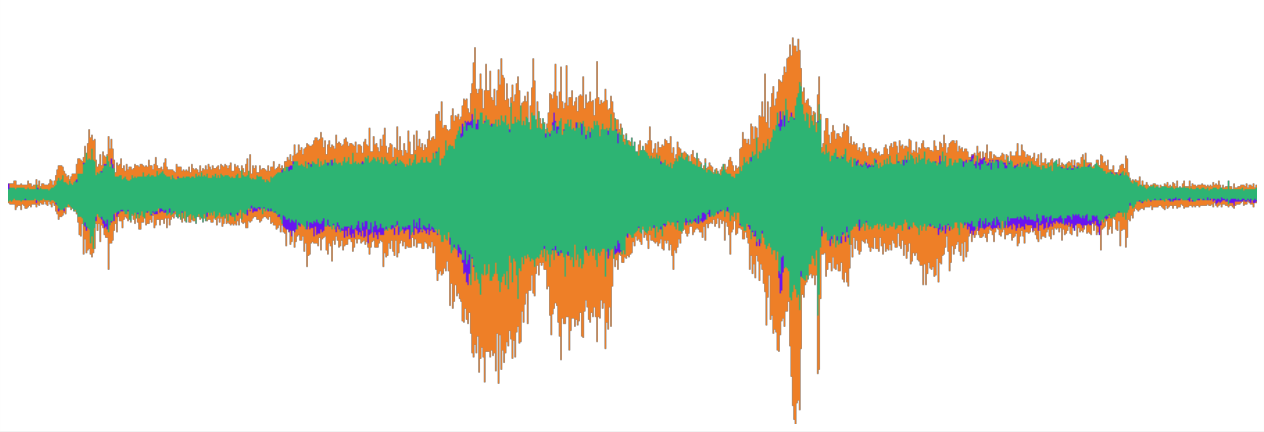
(continues on next page)

(continued from previous page)

```
     || 722 kB 40.9 MB/s
     || 715 kB 50.7 MB/s
     || 705 kB 47.4 MB/s
     || 699 kB 46.8 MB/s
     || 696 kB 44.2 MB/s
     || 684 kB 56.4 MB/s
     || 679 kB 59.6 MB/s
     || 675 kB 48.9 MB/s
     || 675 kB 47.6 MB/s
     || 672 kB 57.3 MB/s
     || 671 kB 50.6 MB/s
     || 669 kB 57.3 MB/s
     || 656 kB 37.9 MB/s
  Building wheel for datashape (setup.py) ... done
ERROR: pip's dependency resolver does not currently take into account all the packages␣
↪that are installed. This behaviour is the source of the following dependency conflicts.
gym 0.17.3 requires cloudpickle<1.7.0,>=1.2.0, but you have cloudpickle 2.0.0 which is␣
↪incompatible.
```

[ ]:
```
import datashader as ds
import datashader.transfer_functions as tf
import xarray as xr
from collections import OrderedDict

def datashader_large_time(df,time_col,colors,height=300,width=900):
  """Use datashader to render an image off a large time series"""
  x_range = (df.index[0], df.index[-1])
  y_range = (df.min(axis=1).min(), 1.2*df.max(axis=1).max())

  cvs = ds.Canvas(x_range=x_range, y_range=y_range, plot_height=height, plot_width=width)

  cols = df.columns
  aggs= OrderedDict((c, cvs.line(df.reset_index(), time_col, c)) for c in cols)

  imgs = [tf.shade(aggs[i], cmap=[c]) for i, c in zip(cols, colors)]

  renamed = [aggs[key].rename({key: 'value'}) for key in aggs]
  merged = xr.concat(renamed, 'cols')
  total = tf.shade(merged.sum(dim='cols'), how='linear')

  return imgs, total
```

[ ]:
```
imgs, total = datashader_large_time(df_vibe,'timestamp',colors)
tf.stack(*imgs)
```



[ ]:
```
def display_all_imgs(imgs,names,colors):
  for i in range(len(imgs)):
    fig = px.imshow(imgs[i],
                    color_continuous_scale  = [ [0, 'rgba(255,255,255,0)'], [1,␣
↪colors[i]] ],
                    origin='lower')
    fig.update_layout(
      {**template_dark, **template_accel},
      title_text = 'Large Time Series with Datashader - '+names[i],
      coloraxis_showscale = False)
    fig.show()
```

[ ]:
```
display_all_imgs(imgs,df_vibe.columns,colors)
```

[ ]:
```
fig = px.imshow(total,
                color_continuous_scale  = [ [0, colors[0]], [1, 'rgba(255,255,255,0)'] ],
                origin='lower')
fig.update_layout(
  {**template_dark, **template_accel},
  title_text = 'Large Time Series with Datashader - Merged',
  coloraxis_showscale = False)
fig.show()
```

That last example was a meager 400,000 data points, what happens when we have 6 million!

[ ]:
```
doc = endaq.ide.get_doc('https://info.endaq.com/hubfs/ford_f150.ide',quiet=True)
df_long = endaq.ide.to_pandas(doc.channels[8],time_mode='seconds')
df_long = df_long - df_long.median()
df_long
```

```
             X (100g)  Y (100g)  Z (100g)
timestamp
0.715484     6.054797  3.714581 -1.500087
0.715684     5.657761  3.765466 -1.354917
0.715884     5.459243  3.612812 -1.500087
0.716084     5.757020  3.969005 -1.645257
```

(continues on next page)

(continued from previous page)

```
0.716284     6.203685  4.070774 -2.032376
...               ...       ...       ...
1207.574669  0.645183  1.984502 -1.306527
1207.574869  0.843701  2.086272 -1.306527
1207.575069  0.496295  2.086272 -1.064578
1207.575269  0.893331  1.730079 -1.161358
1207.575469  0.893331  1.831848 -1.064578

[6034260 rows x 3 columns]
```

[ ]:
```
imgs, total = datashader_large_time(df_long,'timestamp',colors)

fig = px.imshow(total,
                color_continuous_scale  = [ [0, colors[0]], [1, 'rgba(255,255,255,0)'] ],
                origin='lower')
fig.update_layout(
  {**template_dark, **template_accel},
  title_text = 'Large Time Series with Datashader - Merged',
  coloraxis_showscale = False)
fig.show()
```

Now let's get crazy and do a VERY large file with 64 million samples, the file is 385 MB. Note that the free version of Colab will run out of memory if you try this file…

[ ]:
```
doc = endaq.ide.get_doc('https://info.endaq.com/hubfs/data/Mining-Data.ide',quiet=True)
df_very_long = endaq.ide.to_pandas(doc.channels[8],time_mode='seconds')
df_very_long = df_very_long - df_very_long.median()
df_very_long
```

```
                      X         Y         Z
timestamp
22562.056030  -3.604027 -4.576728  0.000000
22562.056080   1.201342 -1.144182  2.341676
22562.056130   2.402685  0.000000 -5.854189
22562.056180   3.604027  0.000000  0.000000
22562.056230  -2.402685  0.000000 -3.512513
...                 ...       ...       ...
25785.220503  -6.006712 -1.144182  0.000000
25785.220553  -9.610740  0.000000 -3.512513
25785.220603 -12.013425  2.288364  0.000000
25785.220653  -7.208055 -2.288364  3.512513
25785.220703  -2.402685 -6.865092  5.854189

[64464451 rows x 3 columns]
```

[ ]:
```
imgs, total = datashader_large_time(df_very_long,'timestamp',colors)

fig = px.imshow(total,
                color_continuous_scale  = [ [0, colors[0]], [1, 'rgba(255,255,255,0)'] ],
                origin='lower')
fig.update_layout(
  {**template_dark, **template_accel},
```

(continues on next page)

(continued from previous page)

```
  title_text = 'VERY Large (64M) Time Series with Datashader - Merged',
  coloraxis_showscale = False)
fig.write_html('large-time-image.html',full_html=False,include_plotlyjs='cdn')
fig.show()
```

2.3.9 Wait, What're Graph Objects?

Adding to a Figure

Because Plotly Express has become so versatile you really only need to use graph_objects when adding to a figure already created by express. Let's demonstrate by adding half sine best fits to our shock response spectrum.

[ ]:
```
fig = px.line(df_pvss,
              color_discrete_sequence=colors)

fig.update_layout(
  {**template_dark, **template_pvss},
  title_text = 'Motorcyle Crash Test with Best Fit Half Sine Pulses')

fig.show()
```

[ ]:
```
damp=0.05
t = np.linspace(0,0.5,num=int(10000))  # NOTE: if there aren't enough samples, low-
↪frequency artifacts will appear!

def half_sine_pulse(t, T):
    """Given a pulse width (T) and total time (t) generate a time series of the pulse␣
↪with an amplitude of 1."""
    result = np.zeros_like(t)
    result[(t > 0) & (t < T)] = np.sin(np.pi*t[(t > 0) & (t < T)] / T)
    df_result = pd.DataFrame({'Time':t,
                              'Pulse':result}).set_index('Time')
    return df_result
```

[ ]:
```
for var,color in zip(df_pvss.columns,colors):
  half_sine_params = endaq.calc.shock.enveloping_half_sine(df_pvss[var]/9.81/39.37,␣
↪damp=damp)

  df_pvss_pulse = endaq.calc.shock.pseudo_velocity(half_sine_params[0] * half_sine_
↪pulse(t, T=half_sine_params[1]),
                                                   freqs=df_pvss.index, damp=damp)

  fig.add_trace(go.Scatter(
      x=df_pvss_pulse.index,
      y=df_pvss_pulse['Pulse'].to_numpy()*9.81*39.37,
      mode="lines",
      line = dict(color=color, width=1,dash='dash'),
      name=var+': Half Sine ('+str(np.round(half_sine_params[1],5))+'s, '+str(np.
↪round(half_sine_params[0],1))+'g)'
```

(continues on next page)

(continued from previous page)

```
  ))

fig.show()
```

Creating a Heatmap

[ ]:
```
from scipy import signal

def spectrogram(df,win,max_freq):
  """Generate a Spectrogram with a defined frequency window size and maximum frequency"""
  fs = len(df)/(df.index[-1]-df.index[0])
  N = int(fs*win) #Number of points in the fft
  w = signal.blackman(N)
  output = []
  figs = []
  for c in df.columns:
    freqs, bins, Pxx = signal.spectrogram(df[c].to_numpy(), fs,window = w,nfft=N,axis=0)
    Pxx = Pxx[:max_freq,:]
    freqs = freqs[:max_freq]
    output.append(Pxx)

    #Generate Figures
    trace = [go.Heatmap(
        x= bins,
        y= freqs,
        z= 10*np.log10(Pxx),
        colorscale='Jet',
        )]
    layout = go.Layout(
        yaxis = dict(title = 'Frequency (Hz)'), # x-axis label
        xaxis = dict(title = 'Time (s)'), # y-axis label
        )
    fig = go.Figure(data=trace, layout=layout)
    figs.append(fig)

  return freqs,bins,output,figs
```

[ ]:
```
freqs,bins,output,figs = spectrogram(df_vibe,0.5,200)

figs[1].update_layout(template_dark,
                      title_text = 'Spectrogram with Graph Objects')
figs[1].write_html('spectrogram.html',full_html=False,include_plotlyjs='cdn')
figs[1].show()
```

4 Coordinate or Tripartite Plot

Let's say I want to make a 4 coordinate plot for shock response spectrums like what I showed in the blog:

This would be a great example of when to use graph_objects to add all these diagonal lines.

Supporting Functions

This will be updated and included in our open enDAQ Library.

[ ]:
```
def log_ticks(start,stop):
  """Create an array of values that would be the equivalent of tick marks in a log␣
↪scaled plot"""
  start = np.floor(np.log10(start))
  stop = np.ceil(np.log10(stop))
  ones = np.linspace(1,9,9)
  output = ones*(10**start)
  for i in np.arange(start+1,stop,1):
    output = np.concatenate((output,
                             ones*(10**i)))
  return np.concatenate((output,np.array([10**stop])))

def build_mult_df(rows,columns,scalar=1,row_exp=1,col_exp=1):
  """Multiply two arrays by each other to form a matrix with defined scalar and exponents
↪"""
  rows2 = np.reshape(rows,(len(rows),1))
  columns2 = np.reshape(columns,(1,len(columns)))
  return pd.DataFrame(data=(rows2**row_exp)*(columns2**col_exp)*scalar,
                      index=rows,
                      columns=columns)

def add_df(fig,df,units):
  """Add lines for each column in a dataframe to an existing figure"""
  for col in df.columns:
    fig.add_trace(go.Scatter(
      x=df.index,
      y=df[col].to_numpy(),
      mode="lines",
      line = dict(color='#404041', width=1,dash='dot'),
      showlegend=False,
      name=str(col)+units
  ))
  return fig
```

[ ]:
```
def add_4cp_pvss(fig,df_pvss):
  """Given a figure and the dataframe of pseudo velocity (in/s) add the 4 coordinate␣
↪plot diagonals"""
  max_vel = np.ceil(np.log10(df_pvss.max(axis=1).max()))
  min_vel = np.floor(np.log10(df_pvss.min(axis=1).min()))

  min_disp = (10**min_vel)/df_pvss.index[-1]/(2*np.pi)
```

(continues on next page)

(continued from previous page)

```
max_disp = (10**max_vel)/df_pvss.index[0]/(2*np.pi)

disps = build_mult_df(df_pvss.index,
                      log_ticks(min_disp,max_disp),
                      scalar=(2*np.pi))
disps = disps[(disps<(10**max_vel)) & (disps>(10**min_vel))]

major_disps = disps.columns[np.remainder(np.log10(disps.columns),1)==0]
for disp in major_disps:
  disp_str = str(disp) + " in"
  if disp>=1:
    disp_str = f"{disp:.0f}" + " in"
  x = (10**max_vel)/(disp*2*np.pi)
  if (x < df_pvss.index[-1]) & (x > df_pvss.index[0]):\
    fig.add_annotation(x=np.log10(x)-.1, y=max_vel,
            text=disp_str,
            textangle=-45,
            showarrow=False)

min_accel = (10**min_vel)*df_pvss.index[0]*(2*np.pi)/(9.81*39.37)
max_accel = (10**max_vel)*df_pvss.index[-1]*(2*np.pi)/(9.81*39.37)

accels = build_mult_df(df_pvss.index,
                      log_ticks(min_accel,max_accel),
                      scalar=9.81*39.37/(2*np.pi),
                      row_exp=-1)
accels = accels[(accels<(10**max_vel)) & (accels>(10**min_vel))]

major_accels = accels.columns[np.remainder(np.log10(accels.columns),1)==0]
for accel in major_accels:
  accel_str = str(accel) + " g"
  if accel>=1:
    accel_str = f"{accel:.0f}" + " g"
  y = accel / (df_pvss.index[-1] * 2 * np.pi) * (9.81*39.37)
  if (y < 10**max_vel) & (y > 10**min_vel):
    fig.add_annotation(x=np.log10(df_pvss.index[-1]), y=np.log10(y)+.1,
            text=accel_str,
            textangle=45,
            showarrow=False)

add_df(fig,disps,' in')
add_df(fig,accels,' g')

fig.update_yaxes(range=[min_vel,max_vel])
fig.update_xaxes(range=np.log10([df_pvss.index[0],df_pvss.index[-1]]))

return fig
```

Motorcycle Crash

[ ]:
```
fig = px.line(df_pvss,
              color_discrete_sequence=colors)

fig = add_4cp_pvss(fig,df_pvss)

fig.update_layout(
  {**template_light, **template_pvss},
  title_text = 'Motorcyle Crash Test in 4CP',
  width=600,
  height=600)
fig.write_html('4cp.html',full_html=False,include_plotlyjs='cdn')
fig.show()
```

El Centro Earthquake

Data credit of good ol' Tom Irvine, the El Centro earthquake in 1940.

[ ]:
```
df_elcentro = pd.read_csv('https://info.endaq.com/hubfs/elcentro_earthquake.csv',index_
↪col=0)

fig = px.line(df_elcentro)
fig.update_layout(
  {**template_light, **template_accel},
  title_text = 'El Centro Earthquake Acceleration')

fig.show()
```

[ ]:
```
df_pvss_elcentro = endaq.calc.shock.pseudo_velocity(df_elcentro,
                                                    get_log_freqs(df_elcentro,init_freq=.
↪1,bins_per_octave=12),
```

(continues on next page)

(continued from previous page)

```
                                                    damp=0.05, two_sided=False)
df_pvss_elcentro = df_pvss_elcentro*9.81*39.37 #convert to in/s
```

[ ]:
```
fig = px.line(df_pvss_elcentro,
              color_discrete_sequence=colors)

fig = add_4cp_pvss(fig,df_pvss_elcentro)

fig.update_layout(
  {**template_light, **template_pvss},
  title_text = 'El Centro Earthquake in 4CP',
  showlegend = False,
  width=600,
  height=600)

fig.show()
```

2.3.10 Sub Plots

Install & Load Libraries

[ ]:
```
!pip install -U -q numpy scipy plotly pandas
exit()
```

```
     || 15.7 MB 5.1 MB/s
     || 28.5 MB 56.8 MB/s
     || 11.3 MB 24.4 MB/s
ERROR: pip's dependency resolver does not currently take into account all the packages␣
↪that are installed. This behaviour is the source of the following dependency conflicts.
tensorflow 2.6.0 requires numpy~=1.19.2, but you have numpy 1.21.2 which is incompatible.
gym 0.17.3 requires cloudpickle<1.7.0,>=1.2.0, but you have cloudpickle 2.0.0 which is␣
↪incompatible.
google-colab 1.0.0 requires pandas~=1.1.0; python_version >= "3.0", but you have pandas␣
↪1.3.3 which is incompatible.
datascience 0.10.6 requires folium==0.2.1, but you have folium 0.8.3 which is␣
↪incompatible.
albumentations 0.1.12 requires imgaug<0.2.7,>=0.2.5, but you have imgaug 0.2.9 which is␣
↪incompatible.
```

[ ]:
```
import numpy as np
import pandas as pd
import scipy

import plotly.express as px
import plotly.graph_objects as go

import endaq
```

(continues on next page)

(continued from previous page)

```
def get_dfs_from_doc(doc, display_table=True, time_mode='seconds'):
  """Extract dataframes from all channels, separating subchannels to their own dataframe␣
↪if they have different units"""
  table = endaq.ide.get_channel_table(doc)
  if display_table:
    display(table)
  for i in table.data.index:
    table.data.loc[i, 'parent'] = table.data['channel'].loc[i].parent.id

  dfs = []
  labels = []
  for ch in doc.channels:
    df = endaq.ide.to_pandas(doc.channels[ch],time_mode=time_mode)
    units = table.data[table.data.parent == doc.channels[ch].id].units.unique()
    types = table.data[table.data.parent == doc.channels[ch].id].type.unique()
    for unit,d_type in zip(units,types):
      columns = table.data[(table.data.parent == doc.channels[ch].id) & (table.data.
↪units == unit)].name
      dfs.append(df[columns])
      labels.append(d_type + ' (' + unit + ')')

  return dfs, labels

def merge_dfs(dfs,labels,plot_points=500,mean_thresh=100):
  """Combine dataframes into one merged one, but downsample to moving RMS for datasets␣
↪with fast sampling rates, and moving mean for slower data sets"""
  master_df = pd.DataFrame()
  for df, label in zip(dfs,labels):
    f_s = dt = (len(df.index) - 1) / (df.index[-1] - df.index[0])
    n = int(df.shape[0]/plot_points) #number of data points to use in windowing
    if n>0:
      if f_s > mean_thresh:
        df = df.rolling(n).std().iloc[::n]
      else:
        df = df.rolling(n).mean().iloc[::n]

    for col in df.columns:
      df_temp = df[col].to_frame()
      df_temp.columns = ['Value']
      df_temp['Name'] = label + ': ' + col
      df_temp['Units'] = label
      df_temp = df_temp.reset_index()
      master_df = pd.concat([master_df,df_temp])
  return master_df
```

[ ]:
```
template_light = dict(
  template="presentation",

  font_family='Open Sans',
  font_size=16,
  font_color='#404041',
```

(continues on next page)

(continued from previous page)

```
  title_font_family = 'Open Sans ExtraBold',
  title_font_size = 24,
  title_x = 0.5,

  xaxis_title_font_family = 'Open Sans ExtraBold',
  xaxis_title_font_size = 20,

  yaxis_title_font_family = 'Open Sans ExtraBold',
  yaxis_title_font_size = 20,

  legend_title='',
  legend_orientation='h',
  legend_y = -0.2,

  plot_bgcolor = '#f3f3f3',
  yaxis_gridcolor = '#dad9d8',
  yaxis_linecolor = '#404041',
  yaxis_mirror = True,
  xaxis_gridcolor = '#dad9d8',
  xaxis_linecolor = '#404041',
  xaxis_mirror = True,
  )

template_dark = template_light.copy()
template_dark['template'] = "plotly_dark"
template_dark['font_color'] = '#f3f3f3'
template_dark['plot_bgcolor'] = "#111111"
template_dark['yaxis_linecolor'] = "#404041"
template_dark['xaxis_linecolor'] = "#404041"
template_dark['yaxis_gridcolor'] = "#404041"
template_dark['xaxis_gridcolor'] = "#404041"

colors = ['#EE7F27', '#6914F0', '#2DB473', '#D72D2D', '#3764FF', '#FAC85F','#27eec0','
↪#b42d4d','#82d72d','#e35ffa']

template_pvss = dict(
  xaxis_title_text = "Natural Frequency (Hz)",
  xaxis_type = 'log',
  yaxis_title_text ="Pseudo Velocity (in/s)",
  yaxis_type = 'log'
)

template_psd = dict(
  xaxis_title_text = "Frequency (Hz)",
  xaxis_type = 'log',
  yaxis_title_text ="Acceleration (g^2/Hz)",
  yaxis_type = 'log'
  )

template_accel = dict(
  xaxis_title_text = "Time (s)",
  yaxis_title_text ="Acceleration (g)",
```

(continues on next page)

(continued from previous page)

```
  )
```

Load Data

[ ]:
```
doc = endaq.ide.get_doc('https://info.endaq.com/hubfs/data/All-Channels.ide', quiet=True)
dfs, labels = get_dfs_from_doc(doc, display_table=True)
```

```
<pandas.io.formats.style.Styler at 0x7f90c0631810>
```

[ ]:
```
merged_dfs = merge_dfs(dfs,labels,plot_points=100)
merged_dfs
```

```
      timestamp       Value                        Name             Units
0      0.338775         NaN  Acceleration (g): X (25g)  Acceleration (g)
1      2.228761    0.012966  Acceleration (g): X (25g)  Acceleration (g)
2      4.118748    0.004471  Acceleration (g): X (25g)  Acceleration (g)
3      6.008682    0.009458  Acceleration (g): X (25g)  Acceleration (g)
4      7.898544    0.010230  Acceleration (g): X (25g)  Acceleration (g)
..          ...         ...                        ...              ...
103  181.239308   88.318638          Light (Ill): Lux       Light (Ill)
104  182.993142  200.759452          Light (Ill): Lux       Light (Ill)
105  184.746975  124.224819          Light (Ill): Lux       Light (Ill)
106  186.500809  113.725342          Light (Ill): Lux       Light (Ill)
107  188.254643  255.837158          Light (Ill): Lux       Light (Ill)

[2217 rows x 4 columns]
```

Facet Plot

[ ]:
```
fig = px.line(merged_dfs,
              x='timestamp',
              y='Value',
              facet_col='Units',
              facet_col_wrap = 4,
              color='Name',
              labels={'Value':''})

fig.update_yaxes(matches=None)
fig.update_layout(
  template="seaborn",
  font_family='Open Sans',
  font_size=16,

  title_text = 'Facet Plot of All Channels',
  title_font_family = 'Open Sans ExtraBold',
  title_font_size = 24,

  legend_orientation = 'h',
  height=1000)
```

(continues on next page)

(continued from previous page)

```
fig.write_html('dashboard.html',full_html=False,include_plotlyjs='cdn')
fig.show()
```

[ ]:
```
fig = px.line(merged_dfs,
              x='timestamp',
              y='Value',
              facet_col='Units',
              facet_col_wrap = 1,
              facet_row_spacing=0.04,
              color='Name',
              labels={'Value':''})

fig.update_yaxes(matches=None)
fig.update_layout(
  template="seaborn",
  font_family='Open Sans',
  font_size=16,

  title_text = 'Facet Plot of All Channels',
  title_font_family = 'Open Sans ExtraBold',
  title_font_size = 24,

  height=1000)
fig.show()
```

2.3.11 Quaternion Animation

Let's pull in some data from an enDAQ sensor that has quaternions on, something that is impossible to understand! Here is what was done to the device:

1. Roll +/- 45
2. Roll +/- 90
3. Pitch +/- 90
4. Yaw +/- 90
5. Roll 720

Quaternion Line

[ ]:
```
doc = endaq.ide.get_doc('https://info.endaq.com/hubfs/data/Roll_Pitch_Yaw.ide',
↪quiet=True)
df_quaternion = endaq.ide.to_pandas(doc.channels[70],time_mode='seconds')
df_quaternion
```

```
                  X         Y         Z         W
timestamp
0.217681   0.000549 -0.000366 -0.000244  1.000000
0.258234   0.001099 -0.000488 -0.000488  1.000000
```

(continues on next page)

(continued from previous page)

```
0.298787   0.001404 -0.000244 -0.000488  1.000000
0.339340   0.035522 -0.044434 -0.000427  0.998352
0.379893   0.035522 -0.044434 -0.000610  0.998352
...             ...       ...       ...       ...
56.789204  0.087402 -0.025391 -0.067383  0.993591
56.829757  0.086060 -0.030212 -0.068787  0.993469
56.870310  0.081360 -0.031860 -0.068665  0.993774
56.910863  0.071289 -0.030334 -0.072021  0.994385
56.951416  0.068848 -0.034607 -0.074097  0.994263

[1400 rows x 4 columns]
```

[ ]:
```
fig = px.line(df_quaternion)
fig.update_layout(
    template_light,
    title_text = 'Quaternion',
    xaxis_title_text = 'Time',
    yaxis_title_text = 'Quaternion'
)
fig.show()
```

Convert to Euler

[ ]:
```
def euler_from_quaternion(df_quaternion):
        """
        Convert a quaternion into euler angles (roll, pitch, yaw)
        roll is rotation around x in radians (counterclockwise)
        pitch is rotation around y in radians (counterclockwise)
        yaw is rotation around z in radians (counterclockwise)
        """

        x = df_quaternion['X'].to_numpy()
        y = df_quaternion['Y'].to_numpy()
        z = df_quaternion['Z'].to_numpy()
        w = df_quaternion['W'].to_numpy()

        t0 = +2.0 * (w * x + y * z)
        t1 = +1.0 - 2.0 * (x * x + y * y)
        roll_x = np.arctan2(t0, t1)

        t2 = +2.0 * (w * y - z * x)
        #t2[t2>1] = 1
        #t2[t2<-1] = -1
        pitch_y = np.arcsin(t2)

        t3 = +2.0 * (w * z + x * y)
        t4 = +1.0 - 2.0 * (y * y + z * z)
        yaw_z = np.arctan2(t3, t4)

        return pd.DataFrame({'Roll':roll_x,
```

(continues on next page)

(continued from previous page)

```
                                'Pitch':pitch_y,
                                'Yaw':yaw_z},
                               index=df_quaternion.index)
```

Euler Line

[ ]:
```
euler = euler_from_quaternion(df_quaternion)*180/np.pi
euler
```

```
                Roll     Pitch       Yaw
timestamp
0.217681    0.062957 -0.041949 -0.028000
0.258234    0.125922 -0.055891 -0.056014
0.298787    0.160879 -0.027898 -0.055992
0.339340    4.085591 -5.088270 -0.230658
0.379893    4.086525 -5.087522 -0.251689
...              ...       ...       ...
56.789204  10.208962 -2.216580 -7.957810
56.829757  10.099325 -2.762193 -8.165903
56.870310   9.573258 -2.989384 -8.155004
56.910863   8.414331 -2.869402 -8.496235
56.951416   8.179713 -3.360261 -8.764316

[1400 rows x 3 columns]
```

[ ]:
```
fig = px.line(euler)
fig.update_layout(
    template_light,
    title_text = 'Euler from Quaternion',
    xaxis_title_text = 'Time',
    yaxis_title_text = 'Angle (Degrees)'
)
fig.show()
```

Animation with Polar

[ ]:
```
df_stacked = pd.DataFrame()
euler_rolled = euler[::10]
for col in euler_rolled.columns:
  df = euler_rolled[col].reset_index()
  df.columns = ['time','value']
  df['time'] = np.round(df['time'],2)
  df['axis'] = col
  df['r'] = 0.5
  df_stacked = pd.concat([df_stacked,df],axis=0)
df_stacked
```

```
      time     value  axis    r
0     0.22  0.062957  Roll  0.5
```

(continues on next page)

(continued from previous page)

```
1     0.62  3.899420  Roll  0.5
2     1.03  1.825449  Roll  0.5
3     1.43 -0.934873  Roll  0.5
4     1.84 -9.934072  Roll  0.5
..     ...       ...   ...  ...
135  54.96  1.261520   Yaw  0.5
136  55.37 -0.815223   Yaw  0.5
137  55.78 -3.869159   Yaw  0.5
138  56.18 -6.322262   Yaw  0.5
139  56.59 -5.541401   Yaw  0.5

[420 rows x 4 columns]
```

[ ]:
```
fig = px.scatter_polar(df_stacked, r='r', theta="value",color='axis',animation_frame=
↪'time')
fig.update_layout(
  template="seaborn",
  font_family='Open Sans',
  font_size=16,
  legend_title_text='',
  title_text = 'Euler Animation',
  title_font_family = 'Open Sans ExtraBold',
  title_font_size = 24,
)
fig.show()
```

In this representation it is quite easy to see what I did!

1. Roll +/- 45
2. Roll +/- 90
3. Pitch +/- 90
4. Yaw +/- 90
5. Roll 720

But… we see some issues with gimbal lock when doing the +/- 90 degree pitch, check out roll and yaw they act like I was spinning around!

Animation in 3D Space

Now with some proper software developers (thank you Connor!), we can get pretty crazy to present this in 3D space with the coordinate system without any gimbal lock issues.

[ ]:
```
rot = scipy.spatial.transform.Rotation.from_quat(np.array([df_quaternion[x] for x in
↪'XYZW']).T)
X = rot.apply([1, 0, 0])
Y = rot.apply([0, 1, 0])
Z = rot.apply([0, 0, 1])
```

(continues on next page)

(continued from previous page)

```
df_trip = pd.DataFrame({letter + a: r[:, idx] for r, letter in zip([X, Y, Z], ['X', 'Y',
↪'Z']) for idx, a in enumerate('xyz')}, index=df_quaternion.index)
df_trip
```

```
                Xx        Xy        Xz  ...        Zx        Zy        Zz
timestamp                               ...
0.217681   1.000000 -0.000489  0.000732  ... -0.000733 -0.001098  0.999999
0.258234   0.999999 -0.000978  0.000975  ... -0.000978 -0.002197  0.999997
0.298787   0.999999 -0.000977  0.000487  ... -0.000490 -0.002807  0.999996
0.339340   0.996051 -0.004010  0.088695  ... -0.088756 -0.070894  0.993527
0.379893   0.996050 -0.004376  0.088682  ... -0.088769 -0.070878  0.993527
...             ...       ...       ...  ...       ...       ...       ...
56.789204  0.989630 -0.138334  0.038675  ... -0.062232 -0.170254  0.983433
56.829757  0.988712 -0.141870  0.048189  ... -0.071867 -0.166833  0.983362
56.870310  0.988539 -0.141667  0.052154  ... -0.074502 -0.157341  0.984730
56.910863  0.987785 -0.147561  0.050060  ... -0.070598 -0.137409  0.987995
56.951416  0.986624 -0.152110  0.058615  ... -0.079021 -0.131779  0.988125

[1400 rows x 9 columns]
```

[ ]:
```
vals = []
for i in range(0, len(X), 10):
  vals.append({'time': df_quaternion.index[i], 'x': 0,       'y': 0,       'z': 0,
↪'channel': 'X' })
  vals.append({'time': df_quaternion.index[i], 'x': X[i, 0], 'y': X[i, 1], 'z': X[i, 2],
↪'channel': 'X'})
  vals.append({'time': df_quaternion.index[i], 'x': 0,       'y': 0,       'z': 0,
↪'channel': 'Y' })
  vals.append({'time': df_quaternion.index[i], 'x': Y[i, 0], 'y': Y[i, 1], 'z': Y[i, 2],
↪'channel': 'Y'})
  vals.append({'time': df_quaternion.index[i], 'x': 0,       'y': 0,       'z': 0,
↪'channel': 'Z' })
  vals.append({'time': df_quaternion.index[i], 'x': Z[i, 0], 'y': Z[i, 1], 'z': Z[i, 2],
↪'channel': 'Z'})
df_rots = pd.DataFrame(vals)
df_rots
```

```
          time         x         y         z channel
0     0.217681  0.000000  0.000000  0.000000       X
1     0.217681  1.000000 -0.000489  0.000732       X
2     0.217681  0.000000  0.000000  0.000000       Y
3     0.217681  0.000488  0.999999  0.001099       Y
4     0.217681  0.000000  0.000000  0.000000       Z
..         ...       ...       ...       ...     ...
835  56.586438  0.995064 -0.096537  0.022999       X
836  56.586438  0.000000  0.000000  0.000000       Y
837  56.586438  0.090656  0.978539  0.185050       Y
838  56.586438  0.000000  0.000000  0.000000       Z
839  56.586438 -0.040370 -0.182052  0.982460       Z

[840 rows x 5 columns]
```

```
[ ]: df_rots['time'] = np.round(df_rots['time'],2)
fig = px.line_3d(df_rots, x='x', y='y', z='z', color='channel', animation_frame='time')
fig.update_layout(
  template="seaborn",
  font_family='Open Sans',
  font_size=16,
  legend_title_text='',
  title_text = 'Quaternion 3D Animation',
  title_font_family = 'Open Sans ExtraBold',
  title_x = 0.5,
  title_font_size = 24,
  scene=dict(
      xaxis=dict(range=[-1.1, 1.1], nticks=5),
      yaxis=dict(range=[-1.1, 1.1], nticks=5),
      zaxis=dict(range=[-1.1, 1.1], nticks=5),
      aspectmode='cube',
  )
)
fig.write_html('quaternion-3d.html',full_html=False,include_plotlyjs='cdn')
fig.show()
```

2.4 Introduction to the enDAQ library

2.4.1 Introduction

This notebook and accompanying webinar was developed and released by the enDAQ team. This is the fourth "chapter" of our series on *Python for Mechanical Engineers*: 1. Get Started with Python * Blog: Get Started with Python: Why and How Mechanical Engineers Should Make the Switch 2. Introduction to Numpy & Pandas for Data Analysis 3. Introduction to Plotly for Plotting Data 4. **Introduction of the enDAQ Library** * Watch Recording of This 5. More Custom Examples

To sign up for future webinars and watch previous ones, visit our webinars page.

2.4.2 Why Did We Develop This?

When analyzing data you'll need: - Customize a bit to meet your specific need - Share the results - Share the *methodology* - Reproduce the analysis for future/other data sets

Writing scripts that produce highly interactive and custom plots addresses all of these needs. And that's why we created the open source enDAQ library - to make analysis more convenient, adaptable and reliable!

2.4.3 Docs

All of our functions are documented at www.docs.endaq.com

The code itself lives on github at: https://github.com/MideTechnology/endaq-python

2.4.4 Installation

Installing is as easy as `pip install endaq` that is needed once when running locally, but everytime in Google Colab.

[1]:
```
!pip install -q endaq
exit() #needed in Colab because they pre-load some libraries
```

```
WARNING: You are using pip version 21.1.2; however, version 22.0.3 is available.
You should consider upgrading via the 'c:\users\cflanigan\documents\endaq\endaq-python\
↪endaq-python\venv\scripts\python.exe -m pip install --upgrade pip' command.
```

[ ]:
```
import endaq

import plotly.express as px
import plotly.io as pio; pio.renderers.default = "iframe"
import plotly.graph_objects as go
import pandas as pd
import numpy as np
import scipy
```

2.4.5 Story 1: PSD of Large Time Series

Accessing Data

This first uses a function `endaq.ide.get_doc()` (see docs) to load in an IDE file. This can accept locations locally, or hosted online.

Please note that Python won't accept backslashes and there are a number of ways around this (libraries!). I typically add an 'r' before the string of the folder like so `r"C:\Users\shanly"+"\\"` which ends up becoming: `'C:\\Users\\shanly\\'`. For more see Python's docs.

[ ]:
```
doc = endaq.ide.get_doc('https://info.endaq.com/hubfs/data/Sys034_1.ide')
```

There are a lot of elements to this object that you can explore with `doc.__dict__` but my favorite is to grab the serial number and part number.

[ ]:
```
print(doc.recorderInfo['PartNumber'])
print(doc.recorderInfo['RecorderSerial'])
```

Once the file is loaded, we are using the `endaq.ide.get_channel_table()` (see docs) which will present the contents of the file.

This file is modestly sized at 140 MB.

[ ]:
```
table = endaq.ide.get_channel_table(doc)
table
```

If you need to parse this `table` dataframe to sort it, find channels of interest, etc., you'll need to grab the data like so.

[ ]:
```
table.data
```

Ok, now that we have IDE file, let's get the data out! This is using the `endaq.ide.to_pandas()` (see docs) to pull out every channel into a dictionary with keys of the channel name.

[ ]:
```
data = {doc.channels[ch].name: endaq.ide.to_pandas(doc.channels[ch], time_mode='datetime
↪') for ch in doc.channels}
data
```

Dashboard of an IDE File

We can pass this into a plot function to display a dashboard! Here are the docs, this accepts a dictionary of dataframes to display a bunch of sub plots for every channel/column.

[ ]:
```
fig = endaq.plot.dashboards.rolling_enveloped_dashboard(
    data,
    desired_num_points=100,
    min_points_to_plot=1,
    plot_as_bars=True,
)
fig.show()
```

New Plotting Theme

We've developed four Plotly themes (although really just two): 1. endaq 2. endaq_light 3. endaq_cloud (Open Sans Font) 4. endaq_cloud_light (Open Sans Font)

The `set_theme()` function creates the above four themes and makes one the default.

This uses a helper function, `define_theme()` which we recommend to those that may want to develop their own theme. Here are the docs.

[ ]:
```
endaq.plot.utilities.set_theme()

fig.update_layout(
    template='endaq'
)
```

Remember these are plotly figures which can be saved as interactive HTML files.

[ ]:
```
fig.write_html('dashboard.html',include_plotlyjs ='cdn')
```

This function allows a lot of customization with how the dashboard is displayed.

[ ]:
```
endaq.plot.dashboards.rolling_enveloped_dashboard(
    data,
    desired_num_points=100,
    min_points_to_plot=1,
    plot_as_bars=False,
    num_rows=1,
    num_cols=None
)
```

This dashboard can be used on any collection of dataframes, not just from an IDE file.

[ ]:
```
csv_data = {
    'crash':pd.read_csv('https://info.endaq.com/hubfs/data/Motorcycle-Car-Crash.csv',
↪index_col=0),
    'moto':pd.read_csv('https://info.endaq.com/hubfs/data/motorcycle-vibration-moving-
↪frequency.csv',index_col=0),
    'instrument':pd.read_csv('https://info.endaq.com/hubfs/data/surgical-instrument.csv',
↪index_col=0),
    'rocket':pd.read_csv('https://info.endaq.com/hubfs/data/blushift.csv',index_col=0),
    'calibration':pd.read_csv('https://info.endaq.com/hubfs/data/Calibration-Shake.csv',
↪index_col=0),
    'baseball':pd.read_csv('https://info.endaq.com/hubfs/data/baseball-throw-
↪acceleration.csv',index_col=0, header=None, names=['X','Y','Z']),
    'volleyball':pd.read_csv('https://info.endaq.com/hubfs/data/volleyball-hit-
↪acceleration.csv',index_col=0, header=None, names=['X','Y','Z']),
    'football':pd.read_csv('https://info.endaq.com/hubfs/data/football-catch-
↪acceleration.csv',index_col=0, header=None, names=['X','Y','Z']),
}
```

[ ]:
```
endaq.plot.dashboards.rolling_enveloped_dashboard(
    csv_data,
    desired_num_points=100,
    min_points_to_plot=1,
```

(continues on next page)

(continued from previous page)

```
    plot_as_bars=True,
)
```

We also have a related function to compute some rolling metrics, not just the envelope. Here I will plot the rolling peak and standard deviation (effectively the RMS). See the docs.

[ ]:
```
endaq.plot.dashboards.rolling_metric_dashboard(
    csv_data,
    desired_num_points=100,
    rolling_metrics_to_plot = ('absolute max', 'std')
)
```

No let's go back to that big dataset and look at our dataframes within it.

[ ]:
```
data.keys()
```

[ ]:
```
data['100g PE Acceleration']
```

If you want to skip this step we just went through (load all data, present dashboard) and you know which channel is of interest, you can directly load in that data with `endaq.ide.to_pandas()` (see docs).

This defaults to present the time with datetime objects which is helpful for synchronization. You can pass in `time_mode='seconds'` to just get the time as seconds if preferred.

[ ]:
```
accel = endaq.ide.to_pandas(doc.channels[8])
accel
```

High Pass Filter

If you noticed, the DC offset on the piezoelectric accelerometer was a bit wonky which should be filtered away. Even if there isn't an egregious DC offset like this example, it is still recommended to apply this filter when doing vibration analysis.

This uses our `endaq.calc.filters.butterworth()` (see docs) to filter intuitively.

[ ]:
```
filtered = endaq.calc.filters.butterworth(accel, low_cutoff=2)
filtered
```

Plotting Large Time Series

Now here's a function similar to the dashboard above that was based off a dictionary of dataframes with each subchannel/column of data getting its own plot.

Here in `rolling_min_max_envelope()` (see docs) though we plot a single dataframe's data on one plot. When using `plot_as_bars` this view will appear identical to loading ALL of the data and plotting it, yet this operation is completed in <10 seconds and will be highly responsive.

We are going to be making a lot of acceleration vs time plots, so I am going to simplify the labeling.

[ ]:
```
accel_time_labels = dict(
    xaxis_title_text='',
    yaxis_title_text='Acceleration (g)',
```

(continues on next page)

(continued from previous page)

```
    legend_title_text=''
)
```

[ ]:
```
fig = endaq.plot.plots.rolling_min_max_envelope(
    filtered,
    desired_num_points=1000,
    plot_as_bars=True,
    opacity=0.7
    )
fig.update_layout(
    accel_time_labels,
    title_text='Filtered Time Series with 13M Points',
)
```

Linear PSD

Now we have the data, let's generate a PSD on the whole thing with `psd.welch()`(see docs), and add in the resultant (PSDs are squared, so the resultant is simply the sum).

[ ]:
```
psd = endaq.calc.psd.welch(filtered, bin_width=1)
psd['Resultant'] = psd.sum(axis=1)
psd
```

Remember this is a pandas dataframe, saving to csv is easy with `to_csv()` (or other file type, see docs)

We're going to make a lot of PSDs so let's make the labeling easy.

[ ]:
```
psd_labels = dict(
    xaxis_title_text='Frequency (Hz)',
    yaxis_title_text='Acceleration (g^2/Hz)',
    legend_title_text='',
    xaxis_type='log',
    yaxis_type='log',
)
```

Now let's plot it in Plotly!

[ ]:
```
fig = px.line(psd)
fig.update_layout(
    psd_labels,
    title_text='Power Spectral Density',
)
```

Log PSD

Once a linear PSD is computed, we have a function to convert it to octave spacing, see docs.

```
[ ]: oct_psd = endaq.calc.psd.to_octave(psd, fstart=4, octave_bins=3)
oct_psd.head()
```

Using Plotly graph_objects, I'll add these lines to the existing plot.

```
[ ]: for c in oct_psd.columns:
  fig.add_trace(go.Scattergl(
      x=oct_psd.index,
      y=oct_psd[c],
      name=c+' Octave',
      line_width=6,
      line_dash='dash'
  ))

fig.show()
```

Spectrogram

Our spectrogram function (see docs) allows for octave spaced frequency bins which drastically reduces the heatmap resolution needed and is arguably a better way to represent the data anyways. This is a spectrogram generated off 13M points and completed in 3 seconds.

```
[ ]: freqs, bins, Pxx, fig = endaq.plot.octave_spectrogram(filtered[['X (100g)']], window=12,␣
↪bins_per_octave=6)
fig.show()
```

Extracting a Subsection of IDE File

With long files there may be subsections we'd like to pull out to save and share. This function does just that, (see docs).

```
[ ]: endaq.ide.extract_time(doc,
                        out='extracted.ide',
                        start=pd.to_datetime('2021-10-22 09:17:15'),
                        end=pd.to_datetime('2021-10-22 09:17:25'))
```

Within Python and with dataframes, remember we can easily "slice" the data to focus on areas of interest. Here I'll generate a PSD on a 10 second period of a fixed operating state.

```
[ ]: psd = endaq.calc.psd.welch(filtered['2021-10-22 09:17:15':'2021-10-22 09:17:25'], bin_
↪width=1)
psd['Resultant'] = psd.sum(axis=1)

fig = px.line(psd)
fig.update_layout(
    psd_labels,
    title_text='Power Spectral Density from 9:17:15 to 9:17:25',
)
```

I knew that to be a particularly interesting time in the file because of the light data this user utilized… clever!

```
[ ]: fig = px.line(data['Light Sensor'][['Lux']][::4])
fig.update_layout(
    showlegend=False,
    xaxis_title_text='',
    yaxis_title_text="Light (Lux)"
)
```

2.4.6 Story 2: Multiple Files

We did a quick test with 3 devices on a shaker.

I know I only care about the secondary accelerometer, so I can load that channel (80) directly on these **THREE** files.

```
[ ]: data = {
    'Shaker' : endaq.ide.to_pandas(endaq.ide.get_doc('https://info.endaq.com/hubfs/data/
↪Transfer_Shaker.ide').channels[80]),
    'Long Beam' : endaq.ide.to_pandas(endaq.ide.get_doc('https://info.endaq.com/hubfs/
↪data/Transfer_Long_Beam.ide').channels[80]),
    'Short Beam' : endaq.ide.to_pandas(endaq.ide.get_doc('https://info.endaq.com/hubfs/
↪data/Transfer_Short_Beam.ide').channels[80])
}
data
```

Dashboard

With these three files, let's generate that dashboard again and compare them all in one figure (with shared x axes).

```
[ ]: fig = endaq.plot.dashboards.rolling_enveloped_dashboard(
    data,
    desired_num_points=500,
    min_points_to_plot=1,
    plot_as_bars=True,
```

(continues on next page)

(continued from previous page)

```
)
fig.update_xaxes(matches='x')
fig.show()
```

Single Plot Comparison

We may want to compare them all in one plot, here I'll combine just the Z axis from each file.

[ ]:
```
time_overall = pd.DataFrame()
for device in data.keys():
  #Get Y axis and filter
  accel = endaq.calc.filters.butterworth(data[device]['Z (40g)'].to_frame(), low_
↪cutoff=2)

  #Rename the column as our test/device name
  accel.columns = [device]

  #Combine Times
  time_overall = pd.concat([time_overall,accel])

time_overall
```

There is still 40,000 data points, so I'll use the rolling_envelope plot to simplify the plot.

[ ]:
```
fig = endaq.plot.plots.rolling_min_max_envelope(
    time_overall,
    desired_num_points=1000,
    plot_as_bars=True,
    opacity=0.7
    )
fig.update_layout(
    accel_time_labels,
    title_text='Comparison of Acceleration in Time Domain',
)
fig.show()
```

PSD Comparison

Now let's compute a PSD for all of these, and combine into one dataframe by rounding to a shared frequency bin.

[ ]:
```
psd_overall = pd.DataFrame()
for device in data.keys():
  #Get Z axis and filter
  accel = endaq.calc.filters.butterworth(data[device]['Z (40g)'].to_frame(), low_
↪cutoff=2)

  #Get PSD
  psd = endaq.calc.psd.welch(accel, bin_width = 0.5)

  #Round to the nearest 0.5 Hz
```

(continues on next page)

(continued from previous page)

```
  psd.index = np.round(psd.index*2,0)/2

  #Rename the PSD column as our test/device name
  psd.columns = [device]

  #Combine PSDs
  psd_overall = pd.concat([psd_overall,psd], axis=1)

psd_overall
```

[ ]:
```
fig = px.line(psd_overall[4:500])
fig.update_layout(
    psd_labels,
    title_text='PSD Comparison',
)
```

Octave PSD

[ ]:
```
oct_psd = endaq.calc.psd.to_octave(psd_overall, fstart=4, octave_bins = 0.5)

fig = px.line(oct_psd['Shaker'][:256])
fig.update_layout(
    psd_labels,
    title_text='PSD Comparison',
    template='endaq_light'
)
```

Transfer Function

I can use the shaker data to compute a transfer function for the other devices on different length beams. This is accomplished by taking the square root of the ratio from the beam PSD to the shaker.

[ ]:
```
transfer = psd_overall[5:240].copy().drop('Shaker',axis=1)
for col in transfer.columns:
    transfer[col] = (psd_overall[col]/psd_overall['Shaker']) ** 0.5
transfer
```

Now we can plot it to see the natural frequencies of these two beams.

[ ]:
```
fig = px.line(transfer)
fig.update_layout(
    title_text='Transfer Function',
    xaxis_title_text='Frequency (Hz)',
    yaxis_title_text='Transfer (g/g)',
    legend_title_text='',
    xaxis_type='log',
    yaxis_type='log',
)
```

Integrate to Displacement

We also have a function to integrate to velocity and displacement which is done here (see docs).

[ ]:
```
displacements = {}
for device in data.keys():
  #Get Z axis and filter
  accel = data[device]['Z (40g)'].to_frame()

  #Double Integrate to Displacement
  [df_accel, df_vel] = endaq.calc.integrate.integrals(accel, n=1, highpass_cutoff=2,␣
↪tukey_percent=0.2)
  [df_vel, df_disp] = endaq.calc.integrate.integrals(df_vel, n=1, highpass_cutoff=2,␣
↪tukey_percent=0.2)
  df_disp = df_disp*9.81*39.37 #convert to in

  #Rename the column as our test/device name
  df_disp.columns = [device]

  #Combine Times
  displacements[device] = df_disp

displacements
```

Resample at Slower Rate

Displacement is dominated by lower frequency content, so we can reduce the amount of data by resampling at 200 Hz with another function.

[ ]:
```
displacement = pd.DataFrame()
for device in displacements.keys():
  #Resample at 100 Hz
  disp_resampled = endaq.calc.utils.resample(displacements[device], sample_rate = 200)

  #Rename the index name to help with "melting"
  disp_resampled.index.name = 'index'

  #Combine Into One DataFrame
  displacement = pd.concat([displacement, disp_resampled.reset_index().melt(id_vars=
↪'index').dropna()])

displacement
```

[ ]:
```
fig = px.line(displacement,
              x='index',
              y='value',
              color='variable')
fig.update_layout(
    title_text = 'Z Axis Displacement Resampled at 200 Hz',
    yaxis_title_text = 'Displacement (in)',
    xaxis_title_text = '',
```

(continues on next page)

(continued from previous page)

```
    legend_title_text = '',
)
```

Synchronization is quite impressive if I do say so myself!! Maybe off by about 0.003 seconds?

2.4.7 Story 3: Shock Event

Now let's analyze some data that was discussed in our blog on the shock response spectrum.

[ ]:
```
doc = endaq.ide.get_doc('https://info.endaq.com/hubfs/data/Motorcycle-Car-Crash.ide')
accel = endaq.ide.to_pandas(doc.channels[8])
accel
```

Plot at Peak Time

The `around_peak()` function (see docs) takes in a dataframe and plots around the peak value across all columns.

[ ]:
```
fig = endaq.plot.plots.around_peak(
    accel,
    num=1500,
    leading_ratio=0.4
    )
fig.update_layout(
    accel_time_labels,
    title_text='Acceleration Around Peak',
)
```

Low Pass Filter

Now let's apply a bunch of low-pass filters (see docs).

[ ]:
```
accel = accel['Y (500g)'].to_frame() #keep as a dataframe
accel = (accel - accel.median()) * -1 #apply a high pass and make the shock positive␣
↪acceleration

accel.columns = ['No Low-Pass']
freqs = [1600, 800, 400, 200, 100, 50]
for freq in freqs:
  name = 'Filtered at: '+str(freq)+' Hz'
  accel[name] = endaq.calc.filters.butterworth(accel['No Low-Pass'].to_frame(), high_
↪cutoff=freq)

accel = accel['2019-07-03 17:05:08.4':'2019-07-03 17:05:08.55'] #isolate the time of␣
↪interest
accel = accel - accel.iloc[0] #force start to 0 to remove any filtering artifact
accel
```

[ ]:
```
fig = px.line(accel)
fig.update_layout(
```

(continues on next page)

(continued from previous page)

```
    accel_time_labels,
    title_text='Motorcycle Car Crash, Effect of Filtering'
)
```

Shock Response Spectrum

First we need to define which frequencies we want to calculate and plot the SRS for.

[ ]:
```
freqs = endaq.calc.utils.logfreqs(accel, init_freq=1, bins_per_octave=12)
```

Now we can calculate the shock response spectrum (see docs).

[ ]:
```
srs = endaq.calc.shock.shock_spectrum(accel, freqs=freqs, damp=0.05, mode='srs')
```

[ ]:
```
fig = px.line(srs)
fig.update_layout(
    title_text='Shock Response Spectrum (SRS)',
    xaxis_title_text="Natural Frequency (Hz)",
    yaxis_title_text="Peak Acceleration (g)",
    legend_title_text='',
    xaxis_type="log",
    yaxis_type="log",
  )
```

Pseudo Velocity Shock Spectrum

That function also allows us to calculate the PVSS.

[ ]:
```
pvss = endaq.calc.shock.shock_spectrum(accel, freqs=freqs, damp=0.05, mode='pvss')
pvss = pvss*9.81*39.37 #convert to in/s
```

[ ]:
```
fig = px.line(pvss)
fig.update_layout(
    title_text='Psuedo Velocity Shock Spectrum (PVSS)',
    xaxis_title_text="Natural Frequency (Hz)",
    yaxis_title_text="Psuedo Velocity (in/s)",
    legend_title_text='',
    xaxis_type="log",
    yaxis_type="log",
  )
```

Enveloped Half Sine

Once a PVSS is calculated we have a function to find a half-sine pulse that would envelop that PVSS (this can make reproducing the event easy with a drop test). See the docs for more.

[ ]:
```
t = np.linspace(0,0.1,num=1000)  # NOTE: if there aren't enough samples, low-frequency␣
↪artifacts will appear!

def half_sine_pulse(t, T):
    result = np.zeros_like(t)
    result[(t > 0) & (t < T)] = np.sin(np.pi*t[(t > 0) & (t < T)] / T)
    df_result = pd.DataFrame({'Time':t,
                              'Pulse':result}).set_index('Time')
    return df_result
```

[ ]:
```
for c in ['No Low-Pass', 'Filtered at: 200 Hz']:

  half_sine_params = endaq.calc.shock.enveloping_half_sine(pvss[c]/9.81/39.37, damp=0.05)
  pvss_pulse = endaq.calc.shock.shock_spectrum(half_sine_params[0] * half_sine_pulse(t,␣
↪T=half_sine_params[1]),
                                               freqs=freqs, damp=0.05, mode='pvss')*9.
↪81*39.37

  fig.add_trace(go.Scattergl(
      x=pvss_pulse.index,
      y=pvss_pulse[pvss_pulse.columns[0]].values,
      name='Half Sine of '+c+': '+str(np.round(half_sine_params[0],1))+'g, '+ str(np.
↪round(half_sine_params[1],5)) +'s',
      line_width=6,
      line_dash='dot'
  ))

fig.show()
```

Impact of Damping

[ ]:
```
pvss_damping = pd.DataFrame()
damps = [0, 0.025, 0.05, 0.10]
for damp in damps:
  pvss = endaq.calc.shock.shock_spectrum(accel['No Low-Pass'].to_frame(), freqs=freqs,␣
↪damp=damp, mode='pvss')
  name = 'Damping Ratio = '+str(damp)
  pvss_damping[name] = pvss[pvss.columns[0]]*9.81*39.37 #convert to in/s
```

[ ]:
```
fig = px.line(pvss_damping)
fig.update_layout(
    title_text='Impact of Damping on PVSS',
    xaxis_title_text="Natural Frequency (Hz)",
    yaxis_title_text="Psuedo Velocity (in/s)",
    legend_title_text='',
    xaxis_type="log",
```

(continues on next page)

(continued from previous page)

```
    yaxis_type="log",
  )
```

Integration to Velocity

[ ]:
```
[df_accel, df_vel] = endaq.calc.integrate.integrals(accel, n=1, highpass_cutoff=None,␣
↪tukey_percent=0)

df_vel = df_vel-df_vel.iloc[0] #forced the starting velocity to 0
df_vel = df_vel*9.81*39.37 #convert to in/s
```

[ ]:
```
fig = px.line(df_vel)
fig.update_layout(
    title_text="Integrated Velocity Time History",
    xaxis_title_text="",
    yaxis_title_text="Velocity (in/s)",
    legend_title_text=''
)
fig.show()
```

2.4.8 Story 4: Moving Frequency

[ ]:
```
df_vibe = pd.read_csv('https://info.endaq.com/hubfs/data/motorcycle-vibration-moving-
↪frequency.csv',index_col=0)
df_vibe = df_vibe - df_vibe.median()
df_vibe
```

Shaded Plot in Time

[ ]:
```
fig = endaq.plot.plots.rolling_min_max_envelope(
    df_vibe,
    desired_num_points=1000,
    plot_as_bars=True,
    opacity=1.0
    )
fig.update_layout(
    accel_time_labels,
    title_text='Engine Reving of Motorcycle',
)
fig.show()
```

Spectrogram

```
[ ]: freqs, bins, Pxx, fig = endaq.plot.octave_spectrogram(df_vibe[['Z (40g)']],
                                                           window=0.5,
                                                           bins_per_octave=12,
                                                           max_freq=1000,
                                                           freq_start=40)
     fig.show()
```

Plot of Frequency vs Time

```
[ ]: df_Pxx = pd.DataFrame(Pxx, index= freqs, columns = bins)
     df_Pxx = 10 ** (df_Pxx/10)
```

```
[ ]: fig = px.line(df_Pxx[df_Pxx.index<500].idxmax())
     fig.update_layout(
         title_text="Moving Peak Frequency",
         xaxis_title_text="",
         yaxis_title_text="Peak Frequency (Hz)",
         showlegend=False
     )
     fig.show()
```

PSD Animation over Time

Now we can prepare for creating an animation of the PSD over time.

```
[ ]: df_Pxx.index.name='Frequency (Hz)'
     df_Pxx.columns.name = 'Time (s)'
     df_Pxx.columns = np.round(df_Pxx.columns,2)
```

Create base figure with animation, this will render fine but we'll want to add the other lines for scaling and reference.

```
[ ]: fig = px.line(
         df_Pxx.reset_index().melt(id_vars='Frequency (Hz)'),
         x='Frequency (Hz)',
         y='value',
         animation_frame='Time (s)',
         log_x=True,
         log_y=True,
     )
```

Add in the max, min, median, and mean lines.

```
[ ]: def add_line(df_stat,name,dash,color):
       fig.add_trace(go.Scattergl(
         x=df_stat.index,
         y=df_stat.values,
         name=name,
         line_width=3,
```

(continues on next page)

(continued from previous page)

```
    line_dash=dash,
    line_color=color
))

#Add max, min, median
for stat,dash,quant in zip(['Max','Min','Median'],
                           ['dash','dash','dot'],
                           [1.0,0.0,0.5]):
  df_stat = df_Pxx.quantile(quant, axis=1)
  add_line(df_stat,stat,dash,'#6914F0')

#Add in mean
df_stat = df_Pxx.mean(axis=1)
add_line(df_stat,'Mean','dot','#2DB473')
```

[ ]:
```
fig.update_layout(
    legend_y=-0.7,
    yaxis_title_text='Acceleration (g^2/Hz)'
)
fig.show()
```

Compare to Overall PSD

[ ]:
```
psd = endaq.calc.psd.welch(df_vibe[['Z (40g)']], bin_width=0.25)
oct_psd = endaq.calc.psd.to_octave(psd,octave_bins=12,fstart=40)
```

[ ]:
```
fig = px.line(oct_psd[oct_psd.index<=1000])
fig.update_layout(
    psd_labels
)
```

2.4.9 Story 5: Sound Data

[ ]:
```
mic = endaq.ide.to_pandas(endaq.ide.get_doc('https://info.endaq.com/hubfs/data/sound/
↪gangnam-style.ide').channels[8],time_mode='seconds')
mic
```

[ ]:
```
mic = mic*-5.3075 #convert to Pa, this will be natively done with devices starting in␣
↪the next month or so
```

[ ]:
```
fig = endaq.plot.plots.rolling_min_max_envelope(
    mic,
    desired_num_points=1000,
    plot_as_bars=True,
    opacity=1.0
    )
fig.update_layout(
    title_text='Gangnam Style, Then Fan',
```

(continues on next page)

(continued from previous page)

```
    yaxis_title_text='Sound Level (Pa)',
    xaxis_title_text='',
    showlegend=False
)
fig.show()
```

Save .WAV File

[ ]:
```
import scipy.io.wavfile

mic_normalized = mic.copy()
mic_normalized /= np.max(np.abs(mic_normalized),axis=0)

scipy.io.wavfile.write(
    'sound.wav',
    int(np.round(1/endaq.calc.utils.sample_spacing(mic))),
    mic_normalized.values.astype(np.float32),
)
```

Play in Notebook

[ ]:
```
import IPython
IPython.display.display(IPython.display.Audio('sound.wav'))
```

Convert to dB

[ ]:
```
n = int(mic.shape[0]/100)
rolling_pa = mic.rolling(n).std()[::n]
rolling_dB = rolling_pa.apply(endaq.calc.utils.to_dB, reference=endaq.calc.utils.dB_refs[
↪"SPL"], raw=True)
```

[ ]:
```
fig = px.line(rolling_dB)
fig.update_layout(
    title_text='Gangnam Style, Then Fan Sound Level',
    yaxis_title_text='Sound Level (dB)',
    xaxis_title_text='',
    showlegend=False
)
fig.show()
```

dB vs Frequency

```
[ ]: df_pascal_psd = endaq.calc.psd.welch(mic, bin_width=1)
df_pascal_octave = endaq.calc.psd.to_octave(df_pascal_psd*df_pascal_psd.index[1],agg="sum
↪",octave_bins=3, fstart=10)

df_audio_psd_dB = df_pascal_octave.apply(endaq.calc.utils.to_dB,
                                         reference=endaq.calc.utils.dB_refs["SPL"]**2,
                                         squared=True,
                                         raw=True)
```

```
[ ]: fig = px.line(df_audio_psd_dB)
fig.update_layout(
    title_text='Sound Level vs Frequency',
    xaxis_title_text='Frequency (Hz)',
    xaxis_type='log',
    yaxis_title_text='Sound Level (dB)',
    showlegend=False
)
```

Spectrogram

```
[ ]: freqs, bins, Pxx, fig = endaq.plot.octave_spectrogram(mic, window=0.5, bins_per_
↪octave=12, freq_start=40, max_freq=5000)
fig.show()
```

2.4.10 enDAQ Cloud as an Alternative

Our enDAQ cloud (cloud.endaq.com) offers an environment to generate interactive dashboards for free without the need to write Python code.

But what is especially unique is that our cloud also allows paying tiers (starting at $100/month) to customize these dashboards with code to accelerate the analysis cycle and allow deploying customizing dashboard generation to colleagues and customers without ever needing to install anything.

Here is an example that has it's own unique URL for sharing with colleagues (requires log-in).

For more on generating these custom reports see our Help Article.

2.4.11 What's Coming Next?

More webinars and more functionality!

1. User Requested Examples
2. Release of endaq.batch for Batch Processing
3. Updating enDAQ Cloud to Provide Access to New Python Library

2.5 enDAQ Custom Analysis

2.5.1 Introduction

This notebook and accompanying webinar was developed and released by the enDAQ team. This is the fifth "chapter" of our series on *Python for Mechanical Engineers*: 1. Get Started with Python * Blog: Get Started with Python: Why and How Mechanical Engineers Should Make the Switch 2. Introduction to Numpy & Pandas for Data Analysis 3. Introduction to Plotly for Plotting Data 4. Introduction of the enDAQ Library - There are lots of examples in this! 5. **More Examples! (Today's Webinar)** - Frequency Analysis (FFTs and PSDs) - Simple Shock Response Spectrums - Peak Analysis - Preview `endaq.batch`

To sign up for future webinars and watch previous ones, visit our webinars page.

2.5.2 Installation

Available on PyPi via: `> pip install endaq`

For the most recent features that are still under development, you can also use pip to install endaq directly from GitHub:
`> pip install git+https://github.com/MideTechnology/endaq-python.git@development`

[ ]:
```
!pip install -q endaq
!pip install -q kaleido #this is for rendering images with plotly
exit() #needed in Colab because they pre-load some libraries, wouldn't be neccessary if␣
↪running locally
```

```
     || 71 kB 3.0 MB/s
     || 11.3 MB 13.0 MB/s
     || 62 kB 749 kB/s
     || 38.1 MB 1.3 MB/s
     || 93 kB 1.0 MB/s
     || 83 kB 1.0 MB/s
     || 25.3 MB 51.8 MB/s
ERROR: pip's dependency resolver does not currently take into account all the packages␣
↪that are installed. This behaviour is the source of the following dependency conflicts.
google-colab 1.0.0 requires pandas~=1.1.0; python_version >= "3.0", but you have pandas␣
↪1.3.4 which is incompatible.
google-colab 1.0.0 requires requests~=2.23.0, but you have requests 2.26.0 which is␣
↪incompatible.
datascience 0.10.6 requires folium==0.2.1, but you have folium 0.8.3 which is␣
↪incompatible.
albumentations 0.1.12 requires imgaug<0.2.7,>=0.2.5, but you have imgaug 0.2.9 which is␣
↪incompatible.
     || 79.9 MB 1.2 MB/s
```

[ ]:
```
import endaq

endaq.plot.utilities.set_theme(theme='endaq')

import plotly.express as px
import plotly.graph_objects as go
import plotly.io as pio; pio.renderers.default = "iframe"
import pandas as pd
import numpy as np
import scipy
```

2.5.3 PSDs & FFTs

Simple Sine Wave

[ ]:
```
time = np.linspace(0,2,200,endpoint=False)

sine_waves = pd.DataFrame(index=time)
sine_waves['0.8g @ 2 Hz'] = 0.8*np.sin(2*np.pi*2 * time)
sine_waves['1g @ 3 Hz'] = np.sin(2*np.pi*3 * time)
```

(continues on next page)

(continued from previous page)

```
sine_waves['0.6g @ 5 Hz'] = 0.6*np.sin(2*np.pi*5 * time)
sine_waves['0.5g @ 4 & 6 Hz'] = 0.5*np.sin(2*np.pi*4 * time) + 0.5*np.sin(2*np.pi*6 *␣
↪time)
sine_waves['0.3g @ 7 Hz'] = 0.3*np.sin(2*np.pi*7 * time)

sine_waves
```

```
      0.8g @ 2 Hz  1g @ 3 Hz  0.6g @ 5 Hz  0.5g @ 4 & 6 Hz  0.3g @ 7 Hz
0.00     0.000000   0.000000     0.000000         0.000000     0.000000
0.01     0.100267   0.187381     0.185410         0.308407     0.127734
0.02     0.198952   0.368125     0.352671         0.583150     0.231154
0.03     0.294500   0.535827     0.485410         0.794687     0.290575
0.04     0.385403   0.684547     0.570634         0.921177     0.294686
...           ...        ...          ...              ...          ...
1.95    -0.470228  -0.809017    -0.600000        -0.951057    -0.242705
1.96    -0.385403  -0.684547    -0.570634        -0.921177    -0.294686
1.97    -0.294500  -0.535827    -0.485410        -0.794687    -0.290575
1.98    -0.198952  -0.368125    -0.352671        -0.583150    -0.231154
1.99    -0.100267  -0.187381    -0.185410        -0.308407    -0.127734

[200 rows x 5 columns]
```

[ ]:
```
fig = px.line(sine_waves)
fig.update_layout(
    title_text='Comparison of Fabricated Sine Waves',
    yaxis_title_text='Acceleration (g)',
    xaxis_title_text='Time (s)',
    legend_title_text=''
)
```

Now to compute the PSD on this using `endaq.calc.psd.welch()`, see docs

[ ]:
```
psd = endaq.calc.psd.welch(sine_waves, bin_width=0.5)
```

[ ]:
```
fig = px.line(psd)
fig.update_layout(
    title_text='Comparison of the PSD of Two Fabricated Sine Waves',
    yaxis_title_text='Acceleration (g^2/Hz)',
    xaxis_title_text='Frequency (Hz)',
    xaxis_type='log',
    legend_title_text=''
)
```

PSDs are the more appropriate way to analyze vibration data for a number of reasons (see blog) but we typically see people want to see the FFT because it's units are easier to intuitively understand.

In my experience the best way to do this is to compute a PSD using our function with a few modifications: - Use scaling as `parseval` which means it is `g^2` instead of `g^2/Hz` - Use a boxcar window so in effect there is no windowing - Define a very fine bin width - Set the overlap between FFTs to 0 (not necessary if the bin width is small enough) - Scale from g^2 as RMS to g-peak via `**2 * (2**0.5)`

[ ]:
```
fft = endaq.calc.psd.welch(
    sine_waves,
    scaling='parseval',
    window='boxcar',
    noverlap=0,
    bin_width=0.1,
)
fft = fft**0.5 * (2**0.5) #scale from g^2 as RMS to g-peak

fig = px.line(fft)
fig.update_layout(
    title_text='Comparison of the FFT (from PSD) of Two Fabricated Sine Waves',
    yaxis_title_text='Acceleration (g)',
    xaxis_title_text='Frequency (Hz)',
    xaxis_type='log',
    legend_title_text='',
    template='endaq_light'
)
```

```
/usr/local/lib/python3.7/dist-packages/scipy/signal/spectral.py:1966: UserWarning:

nperseg = 1000 is greater than input length  = 200, using nperseg = 200
```

Real Sine Wave

Let's load a IDE file with `get_doc` docs.

[ ]:
```
doc = endaq.ide.get_doc('https://info.endaq.com/hubfs/100Hz_shake_cal.ide')
endaq.ide.get_channel_table(doc)
```

```
<pandas.io.formats.style.Styler at 0x7f84f6b2e950>
```

Now get the actual data out of one channel with `to_pandas()` docs.

Then generate a plot with `rolling_min_max_envelope()` (docs) that will instead of plotting all data points make a shaded plot which will look identical to plotting all points but be more responsive and faster (not entirely necessary on this dataset, but becomes so with larger ones).

[ ]:
```
accel = endaq.ide.to_pandas(doc.channels[80], time_mode='seconds')
accel = accel-accel.median() #remove DC offset
endaq.plot.rolling_min_max_envelope(
    accel,
    plot_as_bars=True,
    desired_num_points=1000,
    opacity=0.7
)
```

Here we'll plot the raw data around the peak event with `around_peak()` docs.

[ ]:
```
endaq.plot.around_peak(accel, num=500)
```

[ ]:
```
fft = endaq.calc.psd.welch(
    accel[4.5:9.5], #just do it in time of first dwell
    scaling='parseval',
    window='boxcar',
    noverlap=0,
    bin_width=0.5,
)
fft = fft**0.5 * (2**0.5) #scale from g^2 as RMS to g-peak

fig = px.line(fft[:500])
fig.update_layout(
    title_text='FFT (from PSD) of Real World 10.3g Sine Wave',
    yaxis_title_text='Acceleration (g)',
    xaxis_title_text='Frequency (Hz)',
    legend_title_text='',
)
```

Comparing FFT Options

FFT Option Overview

There are a lot of different ways to compute a FFT in Python. I've been using Welch's method so far but I want to compare that to some other more "direct" methods. So let's compare them! First let's conveniently wrap our manipulation of Welch's method PSD into a FFT inside a function.

[ ]:
```
def welch_fft(df, bin_width=0.5):
  fft = endaq.calc.psd.welch(df, bin_width=bin_width, scaling='parseval', window='boxcar
↪', noverlap=0)
  return fft**0.5 * (2**0.5) #scale from g^2 as RMS to g-peak
```

Now let's wrap something around Numpy's FFT functions for a real discrete Fourier transform.

Notice here we actually have phase information! So we'll return that too.

[ ]:
```
def numpy_fft(df):
  """
  Using Numpy's rfft functions compute a discrete Fourier Transform
  """
  freq = np.fft.rfftfreq(len(df), d=endaq.calc.utils.sample_spacing(df))
  df_fft = pd.DataFrame(
    np.fft.rfft(df.to_numpy(), axis=0),
    index=pd.Series(freq, name="frequency (Hz)"),
    columns=df.columns
  )

  df_mag = df_fft.apply(np.abs, raw=True) / len(df_fft)
  df_phase = df_fft.apply(np.angle, raw=True)

  return df_mag, df_phase
```

Now we will use the FFTW algorithm which is available in the pyFFTW library under a GPL license (which makes it potentially difficult for us to use because we use the more premissive MIT license).

First let's download it.

[ ]:
```
!pip install -q pyfftw
```

```
     || 2.6 MB 5.3 MB/s
```

Now let's use it in a function which allows for a drop-in replacement to the Numpy code. This algorithm is generally regarded as the fatest for computing discrete Fourier transforms - so we'll put it to the test!

[ ]:
```
import pyfftw

def fftw_fft(df):
  """
  Using the FFTW algorithm, compute a discrete Fourier Transform
  """
  freq = pyfftw.interfaces.numpy_fft.rfftfreq(len(df), d=endaq.calc.utils.sample_
↪spacing(df))
  df_fft = pd.DataFrame(
    pyfftw.interfaces.numpy_fft.rfft(df.to_numpy(), axis=0),
    index=pd.Series(freq, name="frequency (Hz)"),
    columns=df.columns
  )

  df_mag = df_fft.apply(np.abs, raw=True) / len(df_fft)
  df_phase = df_fft.apply(np.angle, raw=True)

  return df_mag, df_phase
```

Now let's see a FFT result with this library. Note though this will be relatively large to plot…

[ ]:
```
fft, phase = fftw_fft(accel[4.5:9.5])

fig = px.line(fft[80:120])
fig.update_layout(
    title_text='FFT using FFTW of Real World 10.3g Sine Wave',
    yaxis_title_text='Acceleration (g)',
    xaxis_title_text='Frequency (Hz)',
    legend_title_text='',
)
```

Remember one of the benefits of the DFT is to get phase. Although not particularly useful for this dataset let's plot it, notice the shift for the Z axis at the drive frequency.

[ ]:
```
fig = px.line(phase[80:120])
fig.update_layout(
    title_text='Phase using FFTW of Real World 10.3g Sine Wave',
    yaxis_title_text='Phase Angle (radians)',
    xaxis_title_text='Frequency (Hz)',
    legend_title_text='',
)
```

FFT Option Comparison

Now let's do the fun part to compare the three approaches! First let's make a sine wave with a bit over 1M points.

[ ]:
```
time = np.linspace(0,200,2**20,endpoint=False)

sine_waves = pd.DataFrame(index=time)
sine_waves['10g @ 100 Hz'] = 10*np.sin(2*np.pi*100 * time)
sine_waves['8g @ 99 Hz'] = 8*np.sin(2*np.pi*99 * time)
sine_waves['6g @ 100.25 Hz'] = 6*np.sin(2*np.pi*100.25 * time)
```

[ ]:
```
fig = px.line(sine_waves[:0.01])
fig.update_layout(
    title_text='Comparison of Fabricated Sine Waves',
    yaxis_title_text='Acceleration (g)',
    xaxis_title_text='Time (s)',
    legend_title_text=''
)
```

[ ]:
```
from time import process_time
import timeit
pyfftw.forget_wisdom() #FFTW will be faster on subsequent runs

lengths = [2**14, 16411, 2**17, 131101, 2**20, 1000003]

ffts = pd.DataFrame()
times = pd.DataFrame()
def melt_fft(fft,name,L):
  fft =  fft.reset_index().melt(id_vars='frequency (Hz)')
  fft['Algo'] = name
  fft['Length'] = L
  return fft

def time_fft(func):
    return np.array(timeit.repeat(func, number=1, repeat=10)).mean()

def time_pyfftw(x, threads=1):
    pyfftw.forget_wisdom()
    rfft = pyfftw.builders.rfft(x, threads=threads, auto_align_input=True)
    rfft()
    return np.array(timeit.repeat(lambda: rfft(), number=1, repeat=10)).mean()

for L in lengths:
  x = sine_waves.iloc[:L, 0].to_numpy()

  t1 = process_time()
  wfft_bin = welch_fft(sine_waves.iloc[:L], bin_width=0.5)

  t2 = process_time()
```

(continues on next page)

(continued from previous page)

```
  wfft = welch_fft(sine_waves.iloc[:L], bin_width=1/(L*endaq.calc.utils.sample_
↪spacing(sine_waves)))

  t3 = process_time()
  nfft, phase = numpy_fft(sine_waves.iloc[:L])

  t4 = process_time()
  fftw, phase = fftw_fft(sine_waves.iloc[:L])

  t5 = process_time()

  times_t = pd.DataFrame(
      {'Welch w/ 0.5 Hz Bin':t2-t1,
       'Welch': t3-t2,
       'Numpy':t4-t3,
       'FFTW':t5-t4,
       'Numpy - No OH': time_fft(lambda: np.fft.rfft(x)),
       'FFTW - No OH': time_pyfftw(x, threads=1)
      },
      index=pd.Series([L],name='Length')
  )
  times = pd.concat([times,times_t.reset_index().melt(id_vars='Length')])

  wfft_bin = melt_fft(wfft_bin.copy(),'Welch w/ 0.5 Hz Bin',L)
  wfft = melt_fft(wfft.copy(),'Welch',L)
  nfft = melt_fft(nfft.copy(),'Numpy',L)
  fftw = melt_fft(fftw.copy(),'FFTW',L)

  ffts = pd.concat([ffts,wfft_bin,wfft,nfft,fftw])
```

[ ]:
```
times['str_len'] = times['Length'].astype('str')
fig = px.bar(
    times,
    x='str_len',
    y='value',
    color='variable',
    log_y=True,
    barmode='group',
    labels={'value':'Computation Time (s)','str_len':'Array Length','variable':''}
)
fig.show()
```

[ ]:
```
fig = px.scatter(
    times,
    x='str_len',
    y='value',
    color='variable',
    log_y=True,
    log_x=True,
    labels={'value':'Computation Time (s)','str_len':'Array Length','variable':''}
)
```

(continues on next page)

(continued from previous page)

```
fig.show()
```

So what does this mean!? FFTW is the fastest as expected, but only if we first structure the data in a more efficient way. But typically you will not have the data structured in this "optimal" way for FFTW which means the time it takes to restucture it is real.

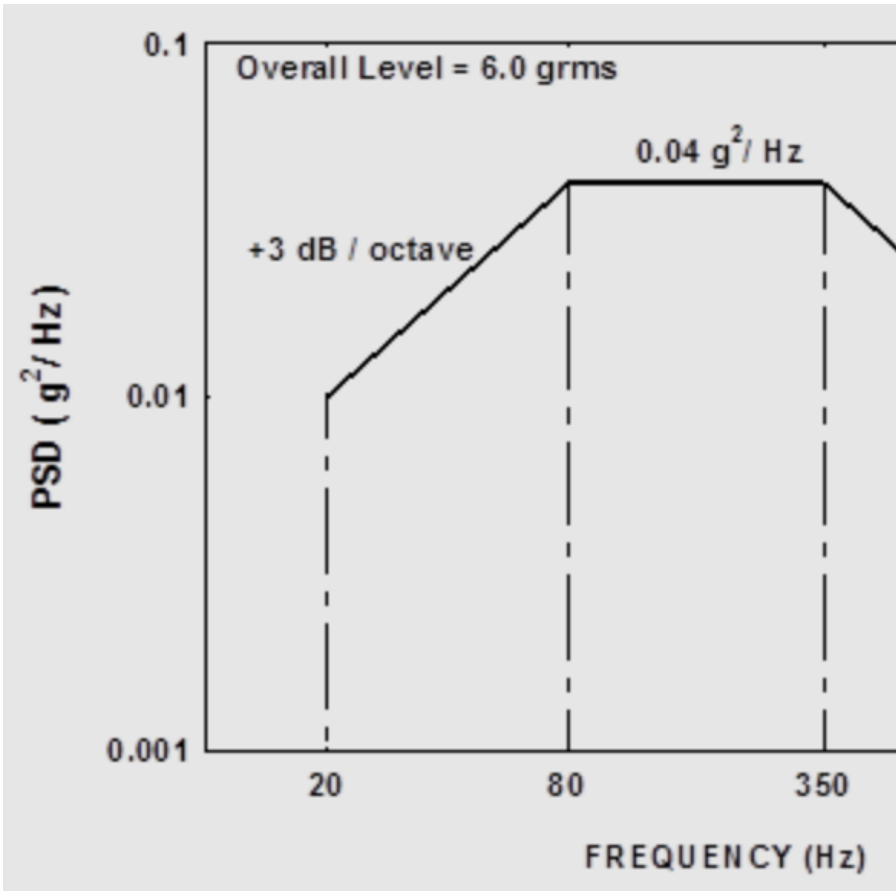
Long story short for *this audience*, using Welch's method is fastest!

[ ]:
```
fig = px.line(
    ffts[(ffts['frequency (Hz)']>95) & (ffts['frequency (Hz)']<105)],
    x='frequency (Hz)',
    y='value',
    color='Algo',
    facet_col='Length',
    facet_row='variable',
    title='FFTs: Acceleration (g) vs Frequency (Hz)',
    labels={'value':'','frequency (Hz)':''}

)
fig.for_each_annotation(lambda a: a.update(text=a.text.split("=")[-1]))
fig.update_layout(width=1200,height=1000)
fig.show()
```

[ ]:
```
fig.write_html('FFT-compare.html',include_plotlyjs='cdn')
```

When comparing the actual FFT results we notice that Welch's with a fixed frequency bin width gives us identical results regardless of the time duration we use - **and these are the "right" or at least expected result**.

Random Vibration (NAVMAT PSD)



Let's look at the NAVMAT PSD which is as follows:

[ ]:
```
navmat = pd.read_csv('https://info.endaq.com/hubfs/navmat-p-9492.csv',delimiter='\t',
↪header=None,index_col=0,names=['Random'])
navmat.index.name='Time (s)'

navmat['Add 2g @ 100 Hz'] = navmat.Random + 2*np.sin(2*np.pi*100 * navmat.index)
navmat['Add 2g @ 200 Hz'] = navmat.Random + 2*np.sin(2*np.pi*200 * navmat.index)
navmat['Add 2g @ 400 Hz'] = navmat.Random + 2*np.sin(2*np.pi*400 * navmat.index)
navmat
```

```
                Random  Add 2g @ 100 Hz  Add 2g @ 200 Hz  Add 2g @ 400 Hz
Time (s)
0.000000  -8.698490e-07    -8.698490e-07    -8.698490e-07    -8.698490e-07
0.000038   8.459350e-07     4.793334e-02     9.583830e-02     1.914556e-01
0.000076   1.871620e-06     9.583932e-02     1.914566e-01     3.811528e-01
0.000114   3.444110e-06     1.436908e-01     2.866355e-01     5.673498e-01
0.000153   6.051080e-06     1.914608e-01     3.811570e-01     7.483369e-01
...                 ...              ...              ...              ...
9.999847   3.435810e-02    -1.571110e-01    -3.468213e-01    -7.140265e-01
9.999886   3.562460e-02    -1.080109e-01    -2.509046e-01    -5.315224e-01
9.999924   1.458970e-02    -8.125498e-02    -1.768794e-01    -3.665897e-01
9.999962   1.604150e-02    -3.183180e-02    -7.967766e-02    -1.751775e-01
10.000000 -2.424250e-04    -2.424250e-04    -2.424250e-04    -2.424250e-04
```

(continues on next page)

(continued from previous page)

```
[262144 rows x 4 columns]
```

[ ]:
```
endaq.plot.rolling_min_max_envelope(navmat, plot_as_bars=True, desired_num_points=500,␣
↪opacity=0.8)
```

[ ]:
```
endaq.plot.around_peak(navmat, num=1000)
```

FFTs

[ ]:
```
fft = welch_fft(navmat, bin_width=1)

fig = px.line(fft[20:2000])
fig.update_layout(
    title_text='NAVMAT P-9492 FFT (1 Hz Bin Width)',
    yaxis_title_text='Acceleration (g)',
    xaxis_title_text='Frequency (Hz)',
    xaxis_type='log',
    yaxis_type='log',
    legend_title_text=''
)
```

Now let's demonstrate the trouble with FFTs by varying the length of time we'll use and the bin width/resolution in the FFT.

[ ]:
```
bins = ['1/T', 1, 10]
times = [1.0, 10]

ffts = pd.DataFrame()
for bin in bins:
  for time in times:
    bin_t = bin
    if bin=='1/T':
      bin_t = 1/time
    fft = welch_fft(navmat[:time], bin_width=bin_t)
    fft = fft[20:2000].reset_index().melt(id_vars='frequency (Hz)')
    fft['Time'] = time
    fft['Bin Width'] = bin

    ffts = pd.concat([ffts,fft])

ffts
```

```
     frequency (Hz)         variable     value  Time Bin Width
0         20.000229           Random  0.288549   1.0       1/T
1         21.000240           Random  0.169284   1.0       1/T
2         22.000252           Random  0.099324   1.0       1/T
3         23.000263           Random  0.101153   1.0       1/T
4         24.000275           Random  0.100844   1.0       1/T
..              ...              ...       ...   ...       ...
```

(continues on next page)

(continued from previous page)

```
787     1950.319916  Add 2g @ 400 Hz  0.388329  10.0        10
788     1960.321557  Add 2g @ 400 Hz  0.369708  10.0        10
789     1970.323197  Add 2g @ 400 Hz  0.335365  10.0        10
790     1980.324838  Add 2g @ 400 Hz  0.378497  10.0        10
791     1990.326478  Add 2g @ 400 Hz  0.346982  10.0        10

[104548 rows x 5 columns]
```

[ ]:
```
fig = px.line(
    ffts,
    x='frequency (Hz)',
    y='value',
    color='variable',
    log_y=True,
    log_x=True,
    facet_row='Time',
    facet_col='Bin Width',
    title='FFTs: Acceleration (g) vs Frequency (Hz)',
    labels={'value':'',
            'frequency (Hz)':'',
            'variable':''}

)
fig.update_layout(width=800,height=600)
fig.show(renderer='svg')
```

Our peaks stay consistent around the 2g we'd expect… but what is happening with the other frequency content!?

PSDs

[ ]:
```
psd = endaq.calc.psd.welch(navmat,bin_width=1)

fig = px.line(psd[20:2000])
fig.update_layout(
    title_text='NAVMAT P-9492 PSD (1 Hz Bin Width)',
    yaxis_title_text='Acceleration (g^2/Hz)',
    xaxis_title_text='Frequency (Hz)',
    xaxis_type='log',
    yaxis_type='log',
    legend_title_text=''
)
```

Now let's do the same comparison of PSDs

[ ]:
```
psds = pd.DataFrame()
for bin in bins:
  for time in times:
    bin_t = bin
    if bin=='1/T':
      bin_t = 1/time
```

(continues on next page)

(continued from previous page)

```
    psd = endaq.calc.psd.welch(
        navmat[:time], bin_width=bin_t)
    psd = psd[20:2000].reset_index().melt(id_vars='frequency (Hz)')
    psd['Time'] = time
    psd['Bin Width'] = bin

    psds = pd.concat([psds,psd])
```

[ ]:
```
fig = px.line(
    psds,
    x='frequency (Hz)',
    y='value',
    color='variable',
    log_y=True,
    log_x=True,
    facet_row='Time',
    facet_col='Bin Width',
    title='PSDs: Acceleration (g^2/Hz) vs Frequency (Hz)',
    labels={'value':'',
            'frequency (Hz)':'',
            'variable':''}

)
fig.update_layout(width=800,height=600)
fig.show(renderer='svg')
```

The peak at those sine tones decrease **yet** the bin width was wider and that sine tone was at one single frequency… so it is less dense. But then the other random/broadband levels are consistent regardless of our length of time or frequency resolution.

PSD Frequency Resolution and Octave Spacing

[ ]:
```
psd_coarse = endaq.calc.psd.welch(navmat,bin_width=10)

fig = px.line(psd_coarse[20:2000])
fig.update_layout(
    title_text='NAVMAT P-9492 PSD (10 Hz Bin Width)',
    yaxis_title_text='Acceleration (g^2/Hz)',
    xaxis_title_text='Frequency (Hz)',
    xaxis_type='log',
    yaxis_type='log',
    legend_title_text=''
)
```

Convert to an octave spaced PSD with `to_octave()` docs.

[ ]:
```
psd = endaq.calc.psd.welch(navmat,bin_width=1)
oct_psd = endaq.calc.psd.to_octave(psd,fstart=20,octave_bins=3)

fig = px.line(oct_psd[:2000])
```

(continues on next page)

(continued from previous page)

```
fig.update_layout(
    title_text='NAVMAT P-9492 PSD 1/3 Octave',
    yaxis_title_text='Acceleration (g^2/Hz)',
    xaxis_title_text='Frequency (Hz)',
    xaxis_type='log',
    yaxis_type='log',
    legend_title_text='',
    template='endaq_light'
)
```

```
/usr/local/lib/python3.7/dist-packages/endaq/calc/psd.py:161: RuntimeWarning:

empty frequency bins in re-binned PSD; original PSD's frequency spacing is too coarse
```

Cumulative Sum from PSD

[ ]:
```
cum_rms = endaq.calc.psd.welch(navmat,bin_width=1, scaling='parseval').cumsum()**0.5

fig = px.line(cum_rms[10:2000])
fig.update_layout(
    title_text='NAVMAT P-9492 PSD with Added Sine Tone',
    yaxis_title_text='Cumulative Acceleration RMS (g)',
    xaxis_title_text='Frequency (Hz)',
    xaxis_type='log',
    legend_title_text=''
)
```

Random Vibration Examples

Bearing

The first example was the topic of our blog on top 12 vibration metrics.

[ ]:
```
bearing = pd.read_csv('https://info.endaq.com/hubfs/Plots/bearing_data.csv', index_col=0)
```

[ ]:
```
endaq.plot.rolling_min_max_envelope(bearing, plot_as_bars=True, desired_num_points=500,␣
↪opacity=0.8)
```

[ ]:
```
psd = endaq.calc.psd.welch(bearing,bin_width=10)

fig = px.line(psd)
fig.update_layout(
    title_text='Bearing Vibration',
    yaxis_title_text='Acceleration (g^2/Hz)',
    xaxis_title_text='Frequency (Hz)',
    xaxis_type='log',
```

(continues on next page)

(continued from previous page)

```
    yaxis_type='log',
    legend_title_text='',
    template='endaq_light'
)
```

Car Engine

During a morning commute (many years ago) I mounted an enDAQ sensor to the car's engine.

[ ]:
```
engine = endaq.ide.to_pandas(endaq.ide.get_doc('https://info.endaq.com/hubfs/data/
↪Commute.ide').channels[8])
engine = endaq.calc.filters.butterworth(engine,low_cutoff=1)
```

[ ]:
```
endaq.plot.rolling_min_max_envelope(engine, plot_as_bars=True, desired_num_points=500,␣
↪opacity=0.8)
```

[ ]:
```
psd = endaq.calc.psd.welch(engine,bin_width=1)

fig = px.line(psd)
fig.update_layout(
    title_text='VIbration of an Engine',
    yaxis_title_text='Acceleration (g^2/Hz)',
    xaxis_title_text='Frequency (Hz)',
    xaxis_type='log',
    yaxis_type='log',
    legend_title_text='',
    template='endaq_light'
)
```

Here we'll use the `octave_spectrogram()` (see docs) to generate a spectrogram with log spaced frequency bins.

[ ]:
```
data, fig = endaq.plot.octave_spectrogram(engine[['Z']], window=2, bins_per_octave=24,␣
↪freq_start= 20, max_freq=100)
fig.show()
```

```
/usr/local/lib/python3.7/dist-packages/endaq/calc/psd.py:161: RuntimeWarning:

empty frequency bins in re-binned PSD; original PSD's frequency spacing is too coarse
```

[ ]:
```
data = 10 ** (data/10)

fig = px.line(data[data.index<500].idxmax())
fig.update_layout(
    title_text="Moving Peak Frequency",
    xaxis_title_text="",
    yaxis_title_text="Peak Frequency (Hz)",
    showlegend=False
```

(continues on next page)

(continued from previous page)

```
)
fig.show()
```

2.5.4 Quick Poll - What FFT Support Should We Add?

Remember this is an open source library you can view, comment and "react" to feature requests and bug reports. Here is an "issue" created to document this need to add some FFT support.

2.5.5 Simple Shock Response Spectrums

We'll look at two datasets in our blog on pseudo velocity.

Punching Bag

```
[ ]: punch = endaq.ide.to_pandas(endaq.ide.get_doc('https://info.endaq.com/hubfs/data/
     ↪Punching-Bag.ide').channels[8], time_mode='seconds')
     punch = punch - punch.median()
```

```
[ ]: endaq.plot.rolling_min_max_envelope(punch, plot_as_bars=True, desired_num_points=500,␣
     ↪opacity=0.8)
```

```
[ ]: fig = endaq.plot.around_peak(punch, num=1000, leading_ratio=0.2)
     fig.update_layout(
         xaxis_title_text='',
         yaxis_title_text='Acceleration (g)',
         legend_title_text=''
     )
```

First determine the frequency bins to calculate it at.

```
[ ]: freqs = endaq.calc.utils.logfreqs(punch[29:30], bins_per_octave=12)
```

Now perform the shock response spectrum calculation for those frequencies.

```
[ ]: srs_punch = endaq.calc.shock.shock_spectrum(punch[29:30], freqs=freqs, damp=0.05, mode=
     ↪'srs')
```

Now plot!

```
[ ]: fig = px.line(srs_punch)
     fig.update_layout(
         title_text='Shock Response Spectrum (SRS) of Punching Bag',
         xaxis_title_text="Natural Frequency (Hz)",
         yaxis_title_text="Peak Acceleration (g)",
         legend_title_text='',
         xaxis_type="log",
         yaxis_type="log",
       )
```

Repeat for the PVSS.

```
[ ]: pvss_punch = endaq.calc.shock.shock_spectrum(punch[29:30], freqs=freqs, damp=0.05, mode=
     ↪'pvss')
     pvss_punch = pvss_punch*9.81*39.37 #convert to in/s
```

```
[ ]: fig = px.line(pvss_punch)
     fig.update_layout(
         title_text='Psuedo Velocity Shock Spectrum (PVSS) of Punching Bag',
         xaxis_title_text="Natural Frequency (Hz)",
         yaxis_title_text="Psuedo Velocity (in/s)",
         legend_title_text='',
         xaxis_type="log",
         yaxis_type="log",
       )
```

MIL-S-901D Barge Shock

```
[ ]: barge = pd.read_csv('https://info.endaq.com/hubfs/data/mil-s-901d-barge.csv', names=[
     ↪'Time (s)','Accel (g)']).set_index('Time (s)')
     barge = barge - barge.median()
```

```
[ ]: endaq.plot.rolling_min_max_envelope(barge, plot_as_bars=True, desired_num_points=500,␣
     ↪opacity=0.8)
```

```
[ ]: fig = endaq.plot.around_peak(barge, num=1000, leading_ratio=0.2)
     fig.update_layout(
         xaxis_title_text='',
         yaxis_title_text='Acceleration (g)',
         showlegend=False
     )
```

First determine the frequency bins to calculate it at. Here though we will specify an initial frequency of one lower than the duration.

```
[ ]: freqs = endaq.calc.utils.logfreqs(barge, init_freq=1, bins_per_octave=12)
```

```
/usr/local/lib/python3.7/dist-packages/endaq/calc/utils.py:54: RuntimeWarning:

the data's duration is too short to accurately represent an initial frequency of 1.000 Hz
```

Now perform the shock response spectrum calculation for those frequencies.

```
[ ]: srs_barge = endaq.calc.shock.shock_spectrum(barge, freqs=freqs, damp=0.05, mode='srs')
```

Now plot!

[ ]:
```
fig = px.line(srs_barge)
fig.update_layout(
    title_text='Shock Response Spectrum (SRS) of MIL-S-901 Barge',
    xaxis_title_text="Natural Frequency (Hz)",
    yaxis_title_text="Peak Acceleration (g)",
    legend_title_text='',
    xaxis_type="log",
    yaxis_type="log",
  )
```

Repeat for the PVSS.

[ ]:
```
pvss_barge = endaq.calc.shock.shock_spectrum(barge, freqs=freqs, damp=0.05, mode='pvss')
pvss_barge = pvss_barge*9.81*39.37 #convert to in/s
```

[ ]:
```
fig = px.line(pvss_barge)
fig.update_layout(
    title_text='Psuedo Velocity Shock Spectrum (PVSS) of MIL-S-901 Barge',
    xaxis_title_text="Natural Frequency (Hz)",
    yaxis_title_text="Psuedo Velocity (in/s)",
    legend_title_text='',
    xaxis_type="log",
    yaxis_type="log",
  )
```

Compare

[ ]:
```
srs_punch = srs_punch[['Z (100g)']].reset_index().melt(id_vars='frequency (Hz)')
srs_punch['variable'] = 'Punching Bag (80g Peak)'

srs_barge = srs_barge.reset_index().melt(id_vars='frequency (Hz)')
srs_barge['variable'] = 'MIL-S-901 (600g Peak)'

srs_combined = pd.concat([srs_punch, srs_barge])
```

[ ]:
```
fig = px.line(
    srs_combined,
    x='frequency (Hz)',
    y='value',
    color='variable'
)
fig.update_layout(
    title_text='Shock Response Spectrum (SRS) Comparison',
    xaxis_title_text="Natural Frequency (Hz)",
    yaxis_title_text="Peak Acceleration (g)",
    legend_title_text='',
    xaxis_type="log",
    yaxis_type="log",
  )
```

[ ]:
```
pvss_punch = pvss_punch[['Z (100g)']].reset_index().melt(id_vars='frequency (Hz)')
pvss_punch['variable'] = 'Punching Bag (80g Peak)'

pvss_barge = pvss_barge.reset_index().melt(id_vars='frequency (Hz)')
pvss_barge['variable'] = 'MIL-S-901 (600g Peak)'

pvss_combined = pd.concat([pvss_punch, pvss_barge])
```

[ ]:
```
fig = px.line(
    pvss_combined,
    x='frequency (Hz)',
    y='value',
    color='variable'
)
fig.update_layout(
    title_text='Psuedo Velocity Shock Spectrum (PVSS) Comparison',
    xaxis_title_text="Natural Frequency (Hz)",
    yaxis_title_text="Psuedo Velocity (in/s)",
    legend_title_text='',
    xaxis_type="log",
    yaxis_type="log",
  )
```

2.5.6 Peak Finding

[ ]:
```
bumps = endaq.ide.to_pandas(endaq.ide.get_doc('https://info.endaq.com/hubfs/data/Robotic-
↪Bumps.ide').channels[80], time_mode='seconds')
bumps = bumps - bumps.median()
```

[ ]:
```
fig = endaq.plot.rolling_min_max_envelope(bumps, plot_as_bars=True, desired_num_
↪points=500, opacity=0.8)
fig.show()
```

Now we'll use SciPy's find_peaks function to isolate the major events.

[ ]:
```
peak_times, _ = scipy.signal.find_peaks(
    x = bumps['Y (40g)'].abs(),
    distance = 5 / endaq.calc.utils.sample_spacing(bumps), #spaced 5 seconds apart
    height = 8
)
```

Let's add them to the previous plot!

[ ]:
```
fig.add_trace(go.Scatter(
    x=bumps.iloc[peak_times].index,
    y=bumps.iloc[peak_times]['Y (40g)'],
    name='Peaks',
    mode='markers',
```

(continues on next page)

(continued from previous page)

```
    marker_size=10,
    marker_color='#D72D2D'
))
fig.show()
```

Now let's plot the peak event with the identified peak.

[ ]:
```
fig = endaq.plot.around_peak(bumps, num=1000, leading_ratio=0.2)
fig.update_layout(
    xaxis_title_text='',
    yaxis_title_text='Acceleration (g)',
    legend_title_text=''
)
fig.add_trace(go.Scatter(
    x=[bumps.iloc[peak_times[2]].name],
    y=[bumps.iloc[peak_times[2]]['Y (40g)']],
    name='Peak',
    mode='markers',
    marker_size=10,
    marker_color='#D72D2D'
))
fig.show()
```

PVSS of Full Time History

[ ]:
```
freqs = endaq.calc.utils.logfreqs(bumps, init_freq=1, bins_per_octave=12)
pvss_bumps = endaq.calc.shock.shock_spectrum(bumps, freqs=freqs, damp=0.05, mode='pvss')
pvss_bumps = pvss_bumps*9.81*39.37 #convert to in/s
```

[ ]:
```
fig = px.line(pvss_bumps)
fig.update_layout(
    title_text='Psuedo Velocity Shock Spectrum (PVSS) of Robotic "Bumps"',
    xaxis_title_text="Natural Frequency (Hz)",
    yaxis_title_text="Psuedo Velocity (in/s)",
    legend_title_text='',
    xaxis_type="log",
    yaxis_type="log",
  )
```

You may be surprised to see the X axis actually had higher peak velocities than the Y axis because of where the frequency content lies!

Loop Through All Peaks

```
[ ]: pvss_all = pd.DataFrame()

for peak in peak_times:
  freqs = endaq.calc.utils.logfreqs(bumps.iloc[peak-1000:peak+4000], init_freq=1, bins_
↪per_octave=12)
  pvss = endaq.calc.shock.shock_spectrum(bumps.iloc[peak-1000:peak+4000], freqs=freqs,␣
↪damp=0.05, mode='pvss')*9.81*39.37 #convert to in/s
  pvss = pvss.reset_index().melt(id_vars='frequency (Hz)')
  pvss['Peak'] = np.abs(bumps.iloc[peak]['Y (40g)'])
  pvss['Time'] = bumps.index[peak]

  pvss_all = pd.concat([pvss_all,pvss])
```

```
[ ]: pvss_all['Peak'] = np.round(pvss_all['Peak'],1)
fig = px.line(
    pvss_all,
    x='frequency (Hz)',
    y='value',
    facet_col='variable',
    color='Peak',
    hover_data=['value','Time'],
    log_x=True,
    log_y=True,
    title='Comparison of PVSS for Each Event',
    labels={'value':'Peak Pseudo Velocity (in/s)','frequency (Hz)':''}
)
fig.for_each_annotation(lambda a: a.update(text=a.text.split("=")[-1]))
fig.show()
```

Notice how the peak at 23.5g has a pseudo velocity about 1/2 that compared to the peak at virtually all the other events, including the one at 12.8g.

2.5.7 Preview of Batch

This is currently available, see docs, but we are working on a few bug fixes and improved functionality. This module allows you to batch process many *IDE* (only works for our sensors for now).

In a seperate document I first executed the following code to gather all the .IDE files I wanted to analyze (I hide the actual folder name). ~~~python import glob directory = r“C:-Notebooks…”+“\”

files = glob.glob(directory+“*.ide*”) #get all the files in that directory with a .ide extension ~~~

Then with this list of files, I performed the batch operation with the following. ~[STRIKEOUT:python calc_output = (endaq.batch.GetDataBuilder(accel_highpass_cutoff=1) .add_psd(freq_bin_width=1) .add_metrics() .aggregate_data(files)) file_data = calc_output.dataframes]~

Then I saved the output dataframes of interest to pickles that I will load next! ~[STRIKEOUT:python file_data[‘psd’].to_pickle(‘batch_psd.pkl’, protocol=4) file_data[‘metrics’].to_pickle(‘batch_metrics.pkl’, protocol=4)]~

Note that I obscured the actual filenames (that would have contained the path) with these lines prior to saving. ~[STRIKEOUT:python file_data[‘metrics’].filename = file_data[‘metrics’].filename.str.split(‘\’).str[-

1].str.split(‘.IDE’).str[0] file_data[‘psd’].filename = file_data[‘psd’].filename.str.split(‘\’).str[-1].str.split(‘.IDE’).str[0]]~

Metrics

[ ]:
```
metrics = pd.read_pickle('https://info.endaq.com/hubfs/data/batch_metrics.pkl')
metrics
```

```
               filename  ...                  start time
0       DAQ11409_000061  ...  2021-10-27 17:33:19.722259
1       DAQ11409_000061  ...  2021-10-27 17:33:19.722259
2       DAQ11409_000061  ...  2021-10-27 17:33:19.722259
3       DAQ11409_000061  ...  2021-10-27 17:33:19.722259
4       DAQ11409_000061  ...  2021-10-27 17:33:19.722259
...                 ...  ...                         ...
148258  DAQ11409_005795  ...  2021-11-16 19:52:53.617858
148259  DAQ11409_005795  ...  2021-11-16 19:52:53.617858
148260  DAQ11409_005795  ...  2021-11-16 19:52:53.617858
148261  DAQ11409_005795  ...  2021-11-16 19:52:53.617858
148262  DAQ11409_005795  ...  2021-11-16 19:52:53.617858

[148263 rows x 6 columns]
```

[ ]:
```
metrics.calculation.unique()
```

```
array(['RMS Acceleration', 'RMS Velocity', 'RMS Displacement',
       'Peak Absolute Acceleration',
       'Peak Pseudo Velocity Shock Spectrum', 'RMS Angular Velocity',
       'Average Temperature'], dtype=object)
```

[ ]:
```
fig = px.scatter(
    metrics[metrics.calculation.isin(['RMS Acceleration',
                                      'RMS Displacement',
                                      'Peak Absolute Acceleration'])],
    x='start time',
    y='value',
    facet_col='calculation',
    facet_col_wrap=1,
    color='axis',
    labels={'value':'',
            'start time':'',
            'axis':''
           },
    hover_data=['filename']
)
fig.update_layout(height=600)
fig.for_each_annotation(lambda a: a.update(text=a.text.split("=")[-1]))
fig.update_yaxes(matches=None)
fig.show()
```

PSDs

[ ]:
```
psd = pd.read_pickle('https://info.endaq.com/hubfs/data/batch_psd.pkl')
psd
```

```
                filename       axis  ...  serial number                  start time
0        DAQ11409_000061    X (40g)  ...          11409  2021-10-27 17:33:19.722259
1        DAQ11409_000061    Y (40g)  ...          11409  2021-10-27 17:33:19.722259
2        DAQ11409_000061    Z (40g)  ...          11409  2021-10-27 17:33:19.722259
3        DAQ11409_000061  Resultant  ...          11409  2021-10-27 17:33:19.722259
4        DAQ11409_000061    X (40g)  ...          11409  2021-10-27 17:33:19.722259
...                  ...        ...  ...            ...                         ...
2876195  DAQ11409_005795  Resultant  ...          11409  2021-11-16 19:52:53.617858
2876196  DAQ11409_005795    X (40g)  ...          11409  2021-11-16 19:52:53.617858
2876197  DAQ11409_005795    Y (40g)  ...          11409  2021-11-16 19:52:53.617858
2876198  DAQ11409_005795    Z (40g)  ...          11409  2021-11-16 19:52:53.617858
2876199  DAQ11409_005795  Resultant  ...          11409  2021-11-16 19:52:53.617858

[2876200 rows x 6 columns]
```

Now we want to aggregate this see how things changed over time. So we'll round the frequency values and focus on one axis. For readability we'll also make a new column that will display time into the test in days.

[ ]:
```
psd['frequency (Hz)'] = np.round(psd['frequency (Hz)'] ,0)
psd['start time'] = pd.to_datetime(psd['start time'])
psd['Test Day'] = psd['start time']-psd['start time'].iloc[0]
psd['Test Day'] = np.round(psd['Test Day'].dt.total_seconds()/60/60/24,3)

psd_y = psd[psd.axis=='Y (40g)'].copy()
psd_y
```

```
                filename     axis  ...                  start time  Test Day
1        DAQ11409_000061  Y (40g)  ... 2021-10-27 17:33:19.722259     0.000
5        DAQ11409_000061  Y (40g)  ... 2021-10-27 17:33:19.722259     0.000
9        DAQ11409_000061  Y (40g)  ... 2021-10-27 17:33:19.722259     0.000
13       DAQ11409_000061  Y (40g)  ... 2021-10-27 17:33:19.722259     0.000
17       DAQ11409_000061  Y (40g)  ... 2021-10-27 17:33:19.722259     0.000
...                  ...      ...  ...                         ...       ...
2876181  DAQ11409_005795  Y (40g)  ... 2021-11-16 19:52:53.617858    20.097
2876185  DAQ11409_005795  Y (40g)  ... 2021-11-16 19:52:53.617858    20.097
2876189  DAQ11409_005795  Y (40g)  ... 2021-11-16 19:52:53.617858    20.097
2876193  DAQ11409_005795  Y (40g)  ... 2021-11-16 19:52:53.617858    20.097
2876197  DAQ11409_005795  Y (40g)  ... 2021-11-16 19:52:53.617858    20.097

[719050 rows x 7 columns]
```

2D Waterfall

Let's see how the PSD changes over time in a 2D plot with a bunch of lines (one per time). To help with the visualization we first need to get a bunch of colors to map to.

[ ]:
```
from plotly import colors
num_colors = len(psd_y[psd_y['Test Day']<1]['Test Day'].unique())
color_steps = colors.sample_colorscale(px.colors.sequential.Turbo, num_colors)
```

Now we can make the plot!

[ ]:
```
fig = px.line(
    psd_y[psd_y['Test Day']<1],
    x='frequency (Hz)',
    y='value',
    color='Test Day',
    hover_data=['filename','start time'],
    color_discrete_sequence = color_steps
)
fig.update_layout(
    xaxis_title_text='Frequency (Hz)',
    yaxis_title_text='Acceleration (g^2/Hz)',
    legend_title_text='',
    legend_y=-0.7,
    height=800,
    xaxis_type='log',
    yaxis_type='log',
    template='endaq_light'
)
```

3D Waterfall

I know everyone wants to see the 3D view…

[ ]:
```
fig = px.line_3d(
    psd_y[psd_y['Test Day']<1],
    x='frequency (Hz)',
    z='value',
    y='Test Day',
    color='Test Day',
    hover_data=['filename','start time'],
    color_discrete_sequence = color_steps,
    log_x=True,
    log_z=True,
    labels={'value':'Acceleration (g^2/Hz)','frequency (Hz)':'Frequency (Hz)'}
)
fig.update_layout(
    title_text='3D Waterfall Plot',
    showlegend = False,
    margin=dict(l=20, r=20, t=20, b=20),
    template='endaq_light'
)
```

[ ]:
```
fig.write_html('3d-waterfall.html',include_plotlyjs='cdn')
```

Animation

This let's us create animations which I'll first do on the first day's worth of data.

[ ]:
```
fig = px.line(
    psd_y[psd_y['Test Day']<1],
    x='frequency (Hz)',
    y='value',
    animation_frame='Test Day',
    hover_data=['filename','start time']
)
fig.update_layout(
    xaxis_title_text='Frequency (Hz)',
    yaxis_title_text='Acceleration (g^2/Hz)',
    legend_title_text='',
    xaxis_type='log',
    yaxis_type='log',
)
```

That's pretty cool! But we'll notice that the animation moves outside the inital bounds pretty quickly. So let's first find an easy way to calculate these metrics of max/min/median per frequency bin.

[ ]:
```
psd_pivot = psd_y.pivot(index='frequency (Hz)', columns='start time', values='value')
psd_pivot
```

```
start time      2021-10-27 17:33:19.722259  ...  2021-11-16 19:52:53.617858
frequency (Hz)                              ...
0.0                           2.419061e-08  ...                5.329716e-08
1.0                           2.557175e-07  ...                3.352921e-07
2.0                           2.522567e-06  ...                3.364408e-06
3.0                           1.247001e-06  ...                2.720061e-06
4.0                           7.596711e-08  ...                4.625317e-07
...                                    ...  ...                         ...
122.0                         4.794718e-08  ...                6.082880e-09
123.0                         1.851752e-08  ...                5.461760e-09
124.0                         1.526794e-08  ...                4.885745e-09
125.0                         1.554448e-08  ...                3.348997e-09
126.0                                  NaN  ...                         NaN

[127 rows x 5703 columns]
```

In this format computing the metric per frequency bin is pretty easy:

[ ]:
```
psd_pivot.max(axis=1)
```

```
frequency (Hz)
0.0      0.000018
1.0      0.000155
```

(continues on next page)

(continued from previous page)

```
2.0      0.000933
3.0      0.000521
4.0      0.000139
           ...
122.0    0.000059
123.0    0.000016
124.0    0.000011
125.0    0.000019
126.0    0.000005
Length: 127, dtype: float64
```

Now we can loop through and add these lines to our animation and update it.

[ ]:
```
def add_line(df_stat,name,dash,color):
  fig.add_trace(go.Scatter(
    x=df_stat.index,
    y=df_stat.values,
    name=name,
    line_width=3,
    line_dash=dash,
    line_color=color
))

#Add max, min, median
for stat,dash,quant in zip(['Max','Min','Median'],
                           [None,None,'dash'],
                           [1.0,0.0,0.5]):
  df_stat = psd_pivot.quantile(quant, axis=1)
  add_line(df_stat,stat,dash,'#6914F0')
```

[ ]:
```
fig.update_layout(
    legend_y=-0.7
)
fig.show()
```

2.5.8 That's a Wrap!

Hopefully you have enough to get started but if not remember we do offer services! And we will be working on more and more documentation and examples!

[ ]:
```
```

[ ]:
```
!pip install -q git+https://github.com/MideTechnology/endaq-python.git@development
exit()  #forces a restart, it will make Colab crash - that is okay! just go on to run␣
↪all the cells below
        #Note that this is only needed in colab and not if running locally
```

```
  Installing build dependencies ... done
  Getting requirements to build wheel ... done
    Preparing wheel metadata ... done
```

(continues on next page)

(continued from previous page)

```
     || 63 kB 1.1 MB/s
     || 93 kB 879 kB/s
     || 38.1 MB 1.3 MB/s
     || 83 kB 1.2 MB/s
  Building wheel for endaq (PEP 517) ... done
ERROR: pip's dependency resolver does not currently take into account all the packages␣
↪that are installed. This behaviour is the source of the following dependency conflicts.
google-colab 1.0.0 requires requests~=2.23.0, but you have requests 2.27.1 which is␣
↪incompatible.
datascience 0.10.6 requires folium==0.2.1, but you have folium 0.8.3 which is␣
↪incompatible.
albumentations 0.1.12 requires imgaug<0.2.7,>=0.2.5, but you have imgaug 0.2.9 which is␣
↪incompatible.
```

2.6 Quick Start Guide

2.6.1 Introduction

This example script will use the enDAQ Python library to do the following:

- Load open-source libraries
- Load data directly from an IDE file recorded from an enDAQ sensor
  - Provide meta data about the sensor which recorded the data
  - Summary the contents of the file as a table of all sensor channels
  - Summarize all data in a dashboard
- Get the accelerometer data
  - Apply a high pass filter
  - Plot the full time history
  - Plot the time history around the peak
  - Plot the time history in a specific time range
- Shock & Vibration Analysis
  - Linear PSD
  - Octave spaced PSD
  - FFT
  - Shock response spectrum

2.6.2 Load Libraries

```
[ ]: import endaq

endaq.plot.utilities.set_theme('endaq_light')

import plotly.express as px
import pandas as pd
import numpy as np
import scipy
```

2.6.3 Load Data

Using the endaq.ide.get_doc function, load in the contents of an IDE file. Note that this doesn't load all data into memory yet.

```
[ ]: doc = endaq.ide.get_doc('https://info.endaq.com/hubfs/data/Mining-Hammer/LOC__3__
     ↪DAQ41551_11_01_02.ide')
```

Display meta data

```
[ ]: print(f"Serial Number: {doc.recorderInfo['RecorderSerial']}")
     print(f"Part Number: {doc.recorderInfo['PartNumber']}")
     print(f"Recording Finished at {pd.to_datetime(doc.lastUtcTime,unit='s')}")
```

```
Serial Number: 9680
Part Number: S3-E2000D40
Recording Finished at 2021-03-25 03:32:32
```

Table of file contents

This uses the function endaq.ide.get_channel_table.

```
[ ]: endaq.ide.get_channel_table(doc)
```

```
<pandas.io.formats.style.Styler at 0x7f1570196fd0>
```

Dashboard of all data

Note this requires first loading in all data into memory. Then it will display bars of all data plotted from min to max.

```
[ ]: channel_dict = {doc.channels[ch].name: endaq.ide.to_pandas(doc.channels[ch]) for ch in␣
     ↪doc.channels}
     fig = endaq.plot.dashboards.rolling_enveloped_dashboard(channel_dict, plot_as_bars=True,␣
     ↪num_rows=1, num_cols=None)
     fig.show()
```

2.6.4 Get accelerometer data

```
[ ]: accel = endaq.ide.get_primary_sensor_data(doc=doc,measurement_type='accel',time_mode=
     ↪'seconds')
     accel
```

```
             X (2000g)  Y (2000g)  Z (2000g)
timestamp
2041.868560 -24.046861  14.828584  -9.588154
2041.868610  -7.205801  14.828584 -11.630245
2041.868660  14.496595  19.594306  20.532687
2041.868710  55.123480  13.240010  77.541059
2041.868760  83.944262   1.766976  84.518203
...                ...        ...        ...
2055.561659  50.956620  13.946043   5.387179
2055.561709  62.068247  14.828584   7.939793
2055.561759  40.713089  14.828584   5.217005
2055.561809   2.864111   6.709206   3.855611
2055.561859 -21.616192  -9.176533   3.004740

[273867 rows x 3 columns]
```

Apply High Pass Filter

Filter away the DC offset, there are more filtering options at endaq.calc.filters.

```
[ ]: accel = endaq.calc.filters.butterworth(accel,low_cutoff=1)
```

Plot Full Time History

endaq.plot.rolling_min_max_envelope() will reduce a large time history into a bar chart that plots from the min to the max so that it appears identical to the full time history plot yet is fast to generate and responsive.

```
[ ]: fig_full_accel = endaq.plot.rolling_min_max_envelope(
         accel,
         desired_num_points=1000,
         plot_as_bars=True,
         opacity=0.7
     ).update_layout(
         yaxis_title_text='Acceleration (g)',
         xaxis_title_text='Time (s)',
         title_text='Full Time History of Acceleration Data'
     )
     fig_full_accel.show()
```

Plot Time History Around Peak

endaq.plot.around_peak() finds and plots around the maximum value.

[ ]:
```
fig_peak = endaq.plot.around_peak(accel, num=1000).update_layout(
    yaxis_title_text='Acceleration (g)',
    xaxis_title_text='Time (s)',
    title_text='Time History Around Peak',
    legend_title_text=''
)
fig_peak.show()
```

Time History at Specific Time

[ ]:
```
fig_time = px.line(accel[2044.6:2044.7]).update_layout(
    yaxis_title_text='Acceleration (g)',
    xaxis_title_text='Time (s)',
    title_text='Time History at Specific Time',
    legend_title_text=''
)
fig_time.show()
```

2.6.5 Analysis

Linear PSD

Calculate the PSD using endaq.calc.psd.welch()

[ ]:
```
#Calculate PSD
psd = endaq.calc.psd.welch(accel, bin_width=2)

fig_psd = px.line(
    psd,
    log_x=True,
    log_y=True
).update_layout(
    xaxis_title='Frequency (Hz)',
    yaxis_title='Acceleration (g^2/Hz)',
    title_text='PSD with 2 Hz Bin Width',
    legend_title_text=''
)

fig_psd.show()
```

Octave PSD

Now convert to octave using endaq.calc.psd.to_octave()

[ ]:
```
#Calculate PSD with a finer bin width to convert to octave
psd = endaq.calc.psd.welch(accel, bin_width=0.1)

#Convert to Octave
oct_psd = endaq.calc.psd.to_octave(psd,octave_bins=6)

fig_psd_oct = px.line(
    oct_psd,
    log_x=True,
    log_y=True
).update_layout(
    xaxis_title='Frequency (Hz)',
    yaxis_title='Acceleration (g^2/Hz)',
    title_text='1/6 Octave PSD',
    legend_title_text=''
)

fig_psd_oct.show()
```

```
/usr/local/lib/python3.7/dist-packages/endaq/calc/psd.py:172: RuntimeWarning:

empty frequency bins in re-binned PSD; original PSD's frequency spacing is too coarse
```

FFT

See endaq.calc.fft for a full list of FFT functions.

[ ]:
```
#Calculate FFT
fft = endaq.calc.fft.rfft(accel[2044.6:2044.7])

#Plot
fig_fft = px.line(fft).update_layout(
    xaxis_title='Frequency (Hz)',
    yaxis_title='Acceleration (g)',
    title_text='FFT of Time Range 2044.6 to 2044.7 Seconds',
    legend_title_text=''
)

fig_fft.show()
```

Shock Response Spectrum

This uses endaq.calc.utils.logfreqs() to define log spaced frequencies to use, then passes those to endaq.calc.shock.shock_spectrum() along with the time history data to compute the shock spectrum.

[ ]:
```
#Calculate for reduced time series
accel_zoomed = accel[2044.6:2044.7]

#Define frequencies to calculate responses at
freqs = endaq.calc.utils.logfreqs(accel_zoomed, bins_per_octave=12, init_freq=0.5)

#Calculate pseudo velocity shock spectrum
pvss = endaq.calc.shock.shock_spectrum(accel_zoomed, freqs=freqs, damp=0.05, mode='pvss')

#Convert to in/s (currently in g/s)
pvss = pvss * 9.81 * 39.37

#Plot
fig_pvss = px.line(
    pvss,
    log_x=True,
    log_y=True
).update_layout(
    xaxis_title='Natural Frequency (Hz)',
    yaxis_title='Peak Pseudo Velocity (in/s)',
    legend_title_text='',
    title_text='Psuedo Velocity Shock Spectrum'
)

fig_pvss.show()
```

```
/usr/local/lib/python3.7/dist-packages/endaq/calc/utils.py:61: RuntimeWarning:

the data's duration is too short to accurately represent an initial frequency of 0.500 Hz
```

CHAPTER

THREE

INSTALLATION

The endaq package is available on PyPI, and can be installed via *pip*:

```
pip install endaq
```

For the most recent features that are still under development, you can also use *pip* to install endaq directly from the GitHub repository:

```
pip install git+https://github.com/MideTechnology/endaq-python.git@development
```

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

CHAPTER

FIVE

LICENSE

The *endaq-python* repository is licensed under the MIT license. The full text can be found in the LICENSE file.

PYTHON MODULE INDEX

e

INDEX

F

G

H

I

J

L

M

O

P

R

S

T

V

W