SPOWTT On-board Tool Documentation

This is the documentation for the SPOWTT On-board Tool.

The tool is mostly written in Python, with some minimal HTML+CSS+JS for the visualization of an SVG graph.

Installation and configuration

Running the tool

The source code folder contains the data-processing module '*dataprocessing.py*', the REST-like server '*server.py*', and the SVG graph builder '*svg_maker.py*'.

The scripts are invoked in the following way

> *python -O server.py python -O dataprocessing.py [/path/to/data/dir/] [Vertical acceleration field name]*

The python command flag '-O' is reccomended when not launching for debug.

For more information regarding spawning these modules, refer to the documentation of each individual module: `server.run_server()` and `dataprocessing.start_observer()`

The following customization endpoints are available:

- **Data-processing**
  - **Command line parameter *Monitored directory***

    This is the directory that will be Monitored for new datafiles.

  - **Command line parameter *Vertical acceleration field name***

The name of the field containing the timeseries for the z-axis acceelrations in the MDF datafile.

- `dataprocessing.ACTION_ON_FILES_FOUND_UPON_STARTUP`
- `dataprocessing.MSI_CALC__FORECAST_TIME_INTERVAL`
- `dataprocessing.ACTION_ON_FILES_FOUND_UPON_STARTUP`
- `dataprocessing.SERVER_ENDPOINT_URL`
- `dataprocessing.LOG_DIR`
- `dataprocessing.LOGFILENAME`

- **Server**
  - `svg_maker.graph()`
    - Font (size, weight, color) for: title, axis label, tickmarks, bottom value, side value
    - Tickmarks: linear / logarithmic
    - Plot size
    - Gradient colors

Tool Description

Data-processing module

The purpose of the dataprocessing module is to monitor a given directory for datafiles produced by the data aqusition system, consume them for calculations as they are produced, log and send the result to the visualization process.

The following steps follow the disovery of a new datafile:

- **Wait datafile to be done being written to**

  This issue and the employed solution are discussed below

- **Read the datafile**

  Perform some sanity checks on the file Append the signalset to life-long buffer: `MDF_Monitor.data_buffer`

- **Recalculate values**

  Compute the new MSI and MSI-forecast (1 hour) based on the new data

- **Save and display values**

The values are logged to a file, at path `dataprocessing.LOG_DIR` and are sent to the server to be displayed on the plot.

The data aquisition suite is currently set to write measurements to files in chunks of 30 seconds; the file format of these is the MicroLab's *MDF* format. This format uses two separate files for the header and the datafile, as there may be multiple datafiles associated with one header file. The header is a binary file with extension '*.mdf*', and carries a representation of the structure of an entry (a row) in the datafile, with extension '*.dta*', as well as metadata about each time trace.

The files are also created at the start of the timetrace they will contain. This means that, to ensure that we are not trying to read a file that is being written to, we need to wait for the next file to appear in the directory. File events will then point to the file containing the time trace immediately following the one in the file that just appeared.

When a new file event arrives, its data are read and if valid it is appended to a `pymarin.TimeSignalSet`.

The set now also contains the more recent data, out of which we need the timebase and the vertical acceleration components to calculate the sickness index.

The sickness index is calculated using the following function found in PyMARIN. It takes the FFT (windowed if *dom* param is positive) of the acceleration data It should be noted that this function requires vertical acceleration without the gravitational component.

pymarin.analysis.timedom.compute_msi_iso(*time*, *zacc*, *T0=3600*, *km=0.3333333333333333*, *dom=-1*)

This function computes the ISO-MSI based on a time trace of the vertical acceleration.

It is based on the pymarin.analysis.freqdom.calc_msi_iso() funciton.

Requires the analyse_toolbox.

Parameters

- **time** (`numpy.ndarray`) – array of size N [s]
- **zacc** (`numpy.array`) – 1D array of size N [m/s2]
- **T0** (*float*) – loat to specify time duration of MSI in [s]
- **km** (*float*) – vomiting ratio factor.
- **dom** (*float*) – if dom is positive, then a wosa spectrum is computed based on dom if dom is negative, then a fft spectrum is computed. This is the prefered method.

Returns

mmsi

Return type

float

dataprocessing.MSI_CALC__FORECAST_TIME_INTERVAL= *3600*

The amount of time to forecast the MSI for

dataprocessing.ACTION_ON_FILES_FOUND_UPON_STARTUP= *<function _delete_old_files>*

This global variable lets you choose what to do with files found in directory at script launch. It can be set to one of the following three functions: `_process_old_files()`, `_delete_old_files()`, `_ignore_old_files()`.

dataprocessing.SERVER_ENDPOINT_URL= *'http://127.0.0.1'*

This point to the server.py instance. In this case it is simply 'localhost'

dataprocessing.LOG_DIR= *WindowsPath('D:/Data/Logs')*

The directory where values are logged as they are produced.

dataprocessing.LOGFILENAME= *'MSI_20200312.log'*

The log format is one file per day, with file name 'MSI_YYYYMMDD.log'

***class* dataprocessing.MDF_Monitor(*vertical_acceleration_field_name: str, callback: Callable[[float, float], None]*)**

This class is used to generate events upon creation of MDF datafiles to feed the data to the MSI calculation.

Parameters

- **vertical_acceleration_field_name** (`str`) – The name of the data field containing the vertical accelerations
- **callback** (`(float,float) -> None()`) – The callable invoked when new values for MSI/MSIf are calculated

dataprocessing.start_observer(*path: str, accfield: str, callback: Callable[[float, float], None] = <function _send_MSI>*) → watchdog.observers.read_directory_changes.WindowsApiObserver

Kick-off the folder watcher process

Parameters

- **path** (`str`) – Path to the directory to monitor for data files

- **callback** (`Callable[[float, float], None]`) – The callable to be invoked when new results are available. Useful for unit tests.

Server module

The purpose of the server module is to host the visualization interface.

This server hosts the static assets related to the webpage. These include:

- ***index.html***

  The main page body (skeleton)

- ***style.css***

  The main style sheet

- ***SPOWTT.js, darkmode.js***

  The main page script and the dark mode toggling script

- **Various assets related to the site icon**

  favicon.ico and many others in */img/*

The favicon looks like tihs:

The server also acts as a REST endpoint at http://localhost/rest/. This is used to hold the values of *MSI* and *MSIf*, which are the current value for the Motion Sickness Index and the predicted value in one hour. These values are updated by the dataprocessing module via a PUT request, which are then displayed on the plot.

Clients that want to view the plot can do so by requesting http://localhost/ or http://localhost/index.html. This page will only be served once, and in itself it does not contain the plot.

There is also a mechanism to ensure that the graph is updated sa soon as a new value is pushed to the server. This is a combination of long-polling and *ETag*'s.

The plot is loaded by the JavaScript (AJAX), which will request it from the server. The server will only construct and send another plot if the version being displayed is too old. This is accomplished by using the *If-None-Match* field in the HTTP request and the *ETag*field of an HTTP response. These field contain value in a special format, namely the string made of the two values (MSI, MSIF) concatenated by a comma with no space (e.g. “1.23,4.56”). As per the HTTP specification, the *ETag* string must be enclosed in double quotation marks, which means the *ETag* field string value will start and end with a ” symbol.

class server.SpowttVizServer

Bases: `dict`

async **wait_for_change**(*timeout_seconds: int = 20*) → Awaitable[bool]

run(*port: int = 80*)

Run the server listener (block until `stop()` is called)

async **stop**()

Stop the server

class server.MainPageHandler(*application: tornado.web.Application*, *request: tornado.httputil.HTTPServerRequest*, ***kwargs: Any*)

Bases: `tornado.web.RequestHandler`

get()

class server.MyStaticFileHandler(*application: tornado.web.Application*, *request: tornado.httputil.HTTPServerRequest*, ***kwargs: Any*)

Bases: `tornado.web.StaticFileHandler`

set_extra_headers(*path*)

For subclass to add extra headers to the response

class server.ReST_Endpoint(*application: tornado.web.Application*, *request: tornado.httputil.HTTPServerRequest*, ***kwargs: Any*)

Bases: `tornado.web.RequestHandler`

initialize(**, parent: server.SpowttVizServer*)

Set reference to parent object

`set_extra_headers(key)`

Add headers to discourage the browser from caching

`finish_with_code(code: int, body=None)`

Conclude the request with a supplied code (and body, if default argument *None* is overridden)

`check(key, *, exists: bool)`

Check whether the existance of the key is as the request expected

async `get(key)`

REST GET

`put(key)`

REST PUT

`post(key)`

REST POST

`patch(key)`

REST PATCH

`delete(key)`

REST DELETE

class `server.MsiGraphHandler(application: tornado.web.Application, request: tornado.httputil.HTTPServerRequest, **kwargs: Any)`

Bases: `tornado.web.RequestHandler`

The request handler for the SVG chart

`initialize(*, parent: server.SpowttVizServer)`

Set reference to parent object

`compute_etag()`

Computes the etag header to be used for this request.

By default uses a hash of the content written so far.

May be overridden to provide custom etag implementations, or may return None to disable tornado's default etag support.

`finish_with_NOT_MODIFIED()`

Conclude the request with a 304 (Not modified)

`finish_with_plot()`

Conclude the request with a 200 and plot in body

async `get()`

This method handles requests for the plot.

If the *ETag* is present and outdated, the plot is served. If the *onchange* query parameter is present, the server will delay the reponse until a value in the REST server is changed.

When the *ETag* matches, or when the *onchange* wait times out without a value having been changed, a 304 (Not modified) is issued.

Otherwise, the plot is served.

`server.run_server()`

Run the server until user iterrupt (Ctrl-C)

SVG Graph creation

`svg_maker.graph`*(main_value: float, side_value: float, *, title: str, y_label: str, arrow_label: str, width: int = 600, height: int = 1000, margin_side: int = 100, margin_top: int = 100, plot_style_css: str = '<style>\nsvg text.title {\n font-size : 46px;\n font-weight: 600;\n}\nsvg text.axislabel {\n font-size : 32px;\n font-weight: 500;\n}\nsvg text.valuelabel {\n font-size : 30px;\n font-weight: 600;\n}\nsvg text.arrowlabel {\n font-size : 22px;\n font-weight: 200;\n}\nsvg text.tickval {\n font-size : 24px;\n font-weight: 500;\n text-anchor: end;\n}\n</style>', tickmarks: List[int] = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100])*

This function composes the SVG plot based on the two values supplied. The main value is represented as a vertical bar, while the side value is represented by an arrow shape.

Parameters

- **main_value** (`float`) – The value to display in the plot
- **side_value** (`float`) – The value to display on the side of the plot
- **title** (`str`) – The title at the top of the plot

- **y_label** (`str`) – The y-axis label
- **arrow_label** (`str`) – The label underneath the arrow (side_value)
- **width** (`int`) – The width of the SVG
- **height** (`int`) – The height of the SVG
- **margin_side** (`int`) – The margin for the plot from the sides
- **margin_top** (`int`) – The margin for the plot from the top
- **fontsize_title** (`int`) – The font size for the title
- **fontsize_axislabel** (`int`) – The font size for the axis label
- **fontsize_valuelabel** (`int`) – The font size for the main value label
- **fontsize_arrowlabel** (`int`) – The font size for the side value label
- **fontsize_tickmarks** (`int`) – The font size for the tickmarks values