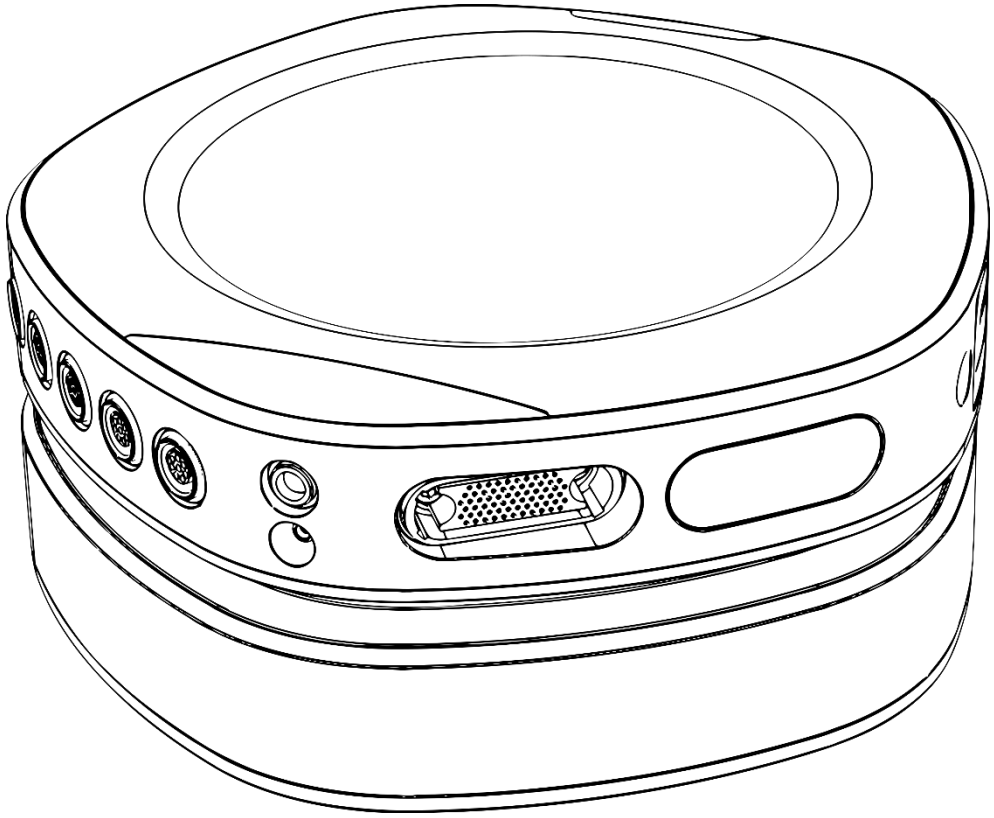
saga 32 64+



API DOCUMENTATION

TMSI PYTHON INTERFACE

REF: 93-2304-0004-EN-3-0

REV 3 **EN**

Last update: 2022-07-29

TABLE OF CONTENTS

1 INTRODUCTION

1.1 About this document

This document describes the TMSi Python Interface (*TMSiPy*), what it does, how it should work and what you can and cannot expect from TMSi regarding this interface. Please read carefully through this document.

1.2 TMSi Python Interface

The TMSi Python Interface is a library for Python written by TMSi to interface the SAGA Device Driver to Python. This interface allows you to acquire data, change device configurations and write and read .Poly5 or .XDF files via Python, as well as writing to an LSL stream outlet.

Functionality that is included in the TMSi Python Interface is limited to the signal acquisition part of the driver and loading data which was previously recorded in TMSi Polybench, the TMSi SAGA interface for MATLAB, or the TMSi Python Interface. Compared to previous versions, on-board memory recordings can now be configured, except for the scheduled RTC start.

1.3 License and Support

The TMSi Python Interface is **free of charge** and distributed under the Apache License, Version 2.0. The complete text of the license is included in LICENSE.txt, which can be found in the root folder of the TMSi Python Interface.

TMSi has tested the interface, but cannot guarantee that it works under every circumstance, let alone that it will function in the experimental setup that you have in mind.

1.4 TMSi Support

The TMSi Python Interface can be obtained through our website's Support page or directly via our GitLab page. We cannot help you write Python code, but any questions about (using) the TMSi Python Interface can be asked via support@tmsi.com.

2 DESCRIPTION OF THE TMSI PYTHON INTERFACE

2.1 Features

The code in the TMSi Python Interface is a Python library that can be used to control TMSi SAGA devices. The goal of this library is to provide an easy and intuitive way of accessing the TMSi devices using Python. You can configure your experiments, write your own code or turn to supported libraries.

The library provides the following functionality:

- Sampling over USB.
- Sampling over electrical (docked), optical and wireless interface.
- (Limited) live plotting of sampled data.
- (Limited) live filtering of sampled data.
- Sampling in impedance mode and (limited) live plotting of impedance values.
- Changing the SAGA device configuration, including saving and loading the configuration using an XML-structured format.
- Directly saving sampled data to Poly5 file format or XDF file format.
- Loading data from Poly5 or XDF file format.
- Catching device related errors. However, no typical error handling has been included.
- Writing to a LabStreamingLayer (LSL) stream outlet
- (Limited) live plotting of the sampled data in a heatmap, specifically designed for HD-EMG applications. Textile HD-EMG grids' configuration is included.
- Downloading recordings stored on SAGA's onboard memory.
- Configuring the device for a card recording (backup logging or button start).
- Offline synchronisation of files recorded with LSL.

The library does not (yet) support the following features:

- Configuring the card to start a scheduled recording.

The library has been tested on the following Python version:

- Python 3.9.12 (Windows 10 / 64bit).

2.2 Installation requirements

The TMSi Python Interface requires the following for Windows computers:

- A computer with Windows 10, 64bit.
- SAGA Device Driver 2.0.0.
- Python version 3.9 or higher
    - Python for Windows: (https://www.python.org/downloads/release/python-3912/).

Other Python versions have not been tested with this release (v3.0.0.0).

2.3 Installation

When using the Python library, make sure that the TMSi SAGA Device Driver has been successfully installed. Before using the library, copy the folder (including TMSiSDK and examples) to the working directory and start by having a look at the example code provided in the 'examples' folder (explained in chapter 3).

Additional Python libraries need to be installed using the Python package manager. Doing so is done best using virtual environments. The Python interface is provided with a PowerShell script that enables the creation of a virtual environment, and the installation of the required libraries (stored in the *requirements_Windows.txt*). In order to run this script, you need to ensure that Python is available on the system's path, which can be confirmed during Python installation or manually via Windows' Environment Variables. To run the PowerShell script, it is needed to give permission to PowerShell. Allowing the installer script to run can be done by opening a command prompt in the directory of the Python interface (type *cmd*), and typing the following line:

PowerShell.exe -ExecutionPolicy UnRestricted -File .\installer39_Windows.ps1

When the Spyder IDE is used, it's needed to change the Python interpreter to the virtual environment's Python installation. The custom environment can be configured in *Preferences → Python interpreter.* The specific file to select can be found in the following path: *.../tmsi_python_interface/.venv/Scripts/python.exe*

A final step in using Spyder is to configure the IPython Graphics Backed to 'inline'. This can be configured from Spyder's menu bar as follows: 'Tools → Preferences → IPython Console → Graphics → Graphics backend → Inline'. Plots can be made interactive again, for instance for analysis purposes, using the *ipython* library that comes with Spyder. This can be done using the two lines of code below:

```
ipython = get_ipython()
ipython.magic("matplotlib qt")
```

2.4 LabStreamingLayer integration

LabStreamingLayer (LSL) is separate program/library used to stream and record data. In order to use the TMSi Python Interface in combination with LSL, the liblsl-library has to be installed separately. More information on LSL, including 'getting started' information, can be found in the documentation of LabStreamingLayer itself (https://labstreaminglayer.readthedocs.io/info/intro.html).

When using the TMSi Python Interface in combination with LSL, the following warning can show up*: 'Couldn't create multicast responder for 224.0.0.1 (set_option: A socket operation was attempted to an unreachable host)'*. This warning originates form the liblsl-library and can unfortunately not be solved by TMSi. The warning does not influence the connection between the TMSi Python interface and LSL and can simply be ignored.

Synchronisation

Files recorded using the LSL-based LabRecorder applications contain separate streams for each device that made data available to the LSL platform. TMSi has developed code that makes it convenient to align these streams, by creating a new file that contains a single stream with all data points. TMSi has found accurate results for sample rates of 2048 Hz and lower, where the difference between synchronised data points remained within a single sample in 15-minute recordings. For recordings made with sampling frequencies of 4000 or 4096 Hz, differences of a few samples (approximately 1ms) per 15 minutes are found. Therefore, we strongly recommend to use the synchronisation tool only for recordings that are made at sample rates of 2048 Hz or lower.

2.5 Usage

Below, a code snippet is given that shows basic usage of the TMSi Python Interface.

```
1    from TMSiSDK import tmsi_device
2    from TMSiSDK.device import DeviceInterfaceType, ChannelType
3
4    tmsi_device.initialize()
5    discoveryList = tmsi_device.discover(tmsi_device.DeviceType.saga,
         DeviceInterfaceType.docked, DeviceInterfaceType.usb)
6
7    dev = discoveryList[0]
8    dev.open()
9    dev.start_measurement()
10
11   dev.stop_measurement()
12   dev.close()
```

The first two lines import two different subclasses of the TMSiSDK which are required to run the code snippet. Line 4 of the code snippet initialises the SDK, after which a device discovery, with specific Interface Types, can be done. This is shown on line 5 of the code snippet. After successful discovery of the available device, a device object can be created (line 7), after which the connection to the device can be opened (line 8). Next, the user may want to change the configuration, initialise a `file_writer` object or `plotter` object. The use of these functionalities is not shown in the code snippet, but can be found in the different examples that are described in Chapter 3. The code snippet shows how to start and stop a measurement on lines 9 and 11. Finally, it is important that the connection to the device is closed at the end of each code execution, which is shown on line 12 of the code snippet.

2.6 Content

The library contains the following directories and classes, which has changed compared to previous versions of the TMSi Python Interface. More information on the changes in the code can be seen in chapter 3, where code snippets are provided to show the differences in syntax.

TMSiSDK/

Folder containing all device-related SDK files.

TMSiFileFormats/

Folder containing file writers and file readers.

TMSiPlotters/

Folder containing the different plotter objects and a GUI framework in which the plotting windows can be embedded.

TMSiProcessing/

Folder containing additional processing capacities. Currently, this contains a class that creates a framework for real-time filtering of incoming data.

TMSiSDK/device

This class is the main device interface and explicitly defines how the classes, properties and methods of TMSi devices are specified in the TMSi Python Interface.

TMSiSDK/error

This class contains TMSi Error codes that can be retrieved in the application.

TMSiSDK/sample_data

This class defines the properties of received samples.

TMSiSDK/sample_data_server

This class puts data, retrieved from a specific device, into a queue. Different consumer types, such as file writers or plotters, are registered to the created queues. The module keeps track of which consumers are registered to which device.

TMSiSDK/settings

This class specifies the 'locally' used settings of the SDK module.

TMSiSDK/tmsi_device

This class is a starting module to initialise the TMSi Python Interface and to instantiate device objects.

TMSiSDK/_resources

Folder containing EEG channel locations of our headcaps and conversion files for displaying the Textile HD-EMG grids correctly. Files are used for plotting purposes and electrode locations in .XDF files.

TMSiSDK/configs

Folder containing different xml-configuration files.

TMSiSDK/devices/

Folder containing device-specific implementations of the SDK. Currently, only the SAGA device is supported.

TMSiSDK/devices/saga/

This folder contains all SAGA-specific device interface implementations, such as the TMSi_Device_API. Furthermore, this folder includes the saga_device and saga_types class, which contain device-specific classes and methods based on the general TMSiSDK/device class. Finally, this folder contains the XML-based reading and writing of SAGA's configuration.

TMSiFileFormats/file_writer

This class explicitly states how the classes, properties and methods of file writers for specific data formats need to be defined.

TMSiFileFormats/file_formats/

This folder contains all different types of specific file writers. This version of the TMSi Python Interface contains a class which writes data to TMSi Polybench native .Poly5 files, .XDF files or an LSL-stream outlet.

TMSiFileFormats/file_readers/

This folder contains all different types of specific file readers. This version of the TMSi Python Interface contains a class which reads data from TMSi Polybench native .Poly5 files as well as a reader for .XDF files. This makes it possible to review/analyse recordings in the Python environment. The XDF file format enables loading and processing data in the analysis library “MNE-Python” (https://mne.tools/stable/).

TMSiPlotters/gui/

This folder contains the GUI with which the user can interact when plotting data. All different plotter objects are embedded in this GUI. Opening and closing of the GUI controls starting and stopping of measurements.

TMSiPlotters/plotters/

This folder contains the three types of plotters that are available to the user.

TMSiPlotters/plotters/impedance_plotter

This class creates a viewer for the measured impedance values in either a grid or head layout. Next to the plot, the specific values for each channel is displayed. When file storage is enabled, the measured impedance values are saved to a .txt-file in the measurements folder.

TMSiPlotters/plotters/plotter

This class creates a viewer that displays the sampled time-series data online. For reasons of performance, the visualised data is downsampled to 500 Hz. The plotter allows for autoscaling or changing the displayed range of the different channels, observing different time ranges (from 1 to 10 seconds) and making a selection on which channels to display during the recording.

TMSiPlotters/plotters/plotter_hd_emg

This class provides a viewer that displays the high-pass filtered RMS values in a grid layout. To improve the resolution, linear interpolation is applied to neighbouring data

points. The plotter is both available for 32-channel as well as 64-channel configurations. For Textile HD-EMG grids, only the 32-channel variant is supported. The user can set the orientation of the grid, as well as the upper limit for the colorbar, prior to starting the application. Scaling or setting the range of the colorbar is also available.

TMSiProcessing/filters/

This folder contains the different types of pre-configured filters that can be used in combination with a plotter object. Currently, only a semi-real-time filter is implemented that can be used to retrieve and filter data, which can be passed to the plotter. Different (Butterworth) filters can be generated for the three analogue channel types (BIP, UNI and AUX). The user can configure a high-pass, low-pass and band-pass filter for these channel groups.

TMSiProcessing/synchronisation/

This folder contains a class that makes it possible to synchronise different streams in an .xdf file, based off the timestamps that the LSL framework registered during acquisition.

3 MIGRATION INFORMATION (V2 TO V3)

The content described in chapter 2.6 shows the differences in the architecture between v2.0.0.0 and v3.0.0.0 of the TMSi Python Interface.

Architecture change

Comparison of the old and new examples shows that the majority of changes only impacts the import statements of the code. After importing specific modules, most classes can be used in the same way as in previous versions. An example of how such an import statement is made, is shown in the code snippet below.

TMSi Python interface < v3.0.0.0	TMSi Python interface = v.3.0.0.0

`from TMSiSDK import device`
`from TMSiSDK import filters`
`from TMSiSDK.file_writer import`
`  FileWriter, FileFormat`
`from TMSiSDK.file_readers import`
`  Poly5Reader`
`...`
`file_writer =`
`  FileWriter(FileFormat.poly5,`
`  'path_to_file.poly5')`
`filter_appl =`
`  filters.RealTimeFilter(dev)`
	`from TMSiSDK import device`
	`from TMSiProcessing import filters`
	`from TMSiFileFormats.file_writer`
	`  import FileWriter, FileFormat`
	`from TMSiFileFormats.file_readers`
	`  import Poly5Reader`
	`...`
	`file_writer =`
	`  FileWriter(FileFormat.poly5,`
	`  'path_to_file.poly5')`
	`filter_appl =`
	`  filters.RealTimeFilter(dev)`

Change to device discovery

An issue was found when switching between Data Recorder interfaces and using the device via the changed interface. Therefore, the creation of a device object has been split in two steps: device discovery and device opening. An advantage of this change is that multi-device recordings can be set up more conveniently, as multiple handles to specific device objects can be received with the `discover()` function.

TMSi Python interface < v3.0.0.0	TMSi Python interface = v.3.0.0.0

`dev = tmsi_device.create(`
`  tmsi_device.DeviceType.saga,`
`  DeviceInterfaceType.docked,`
`  DeviceInterfaceType.usb)`
 	
`dev.open()`
	`discoveryList = tmsi_device.discover(`
	`  tmsi_device.DeviceType.saga,`
	`  DeviceInterfaceType.docked,`
	`  DeviceInterfaceType.usb)`
	
	`if (len(discoveryList) > 0):`
	`  dev = discoveryList[0]`
	`  dev.open()`

Change to TMSiPlotters

With the change in the overall architecture, the way the plotter objects are handled has also been reconsidered. In previous versions, the plotters contained various User Interface elements. This made it inconvenient to embed the viewers in new applications. To make this more straight-forward, the plotter classes have been splitted into a GUI window and the actual viewers. The GUI provides a framework in which the viewers can be embedded. Depending per type of viewer, one or multiple UI buttons are included to interact with the viewers. The type of embedded viewer is an input parameter that can be provided to the `PlottingGUI`. More information can be seen in the code snippet below.

TMSi Python interface < v3.0.0.0	TMSi Python interface = v.3.0.0.0

`from TMSiSDK import plotters`	`from TMSiPlotters.gui import`
	`  PlottingGUI`
	`from TMSiPlotters.plotters import`
	`  PlotterFormat`

# Signal plotter 	
plot_window=plotters.RealTimePlot(
figurename = 'A RealTimePlot',
device = dev,
channel_selection = [0,1,2])
 	
# Impedance plotter 	
plot_window=plotters.ImpedancePlot(
figurename = 'An Impedance Plot',
device = dev,
layout = 'head')
	# Signal plotter
	plot_window = PlottingGUI(
	plotter_format = PlotterFormat.signal_viewer,
	figurename = 'A RealTimePlot',
	device = dev,
	channel_selection = [0, 1, 2],
	filter_app = filter_appl)
	
	# Impedance plotter
	plot_window = PlottingGUI(
	plotter_format = PlotterFormat.impedance_viewer,
	figurename = 'An Impedance Plot',
	device = dev,
	layout = 'head')

Furthermore, support for the Textile HD-EMG grid is included. This requires the user to perform two steps: 1. Loading the right device configuration (.xml-file), specifically made for the Textile HD-EMG grid. 2. Provide input to the `PlottingGUI` on what type of grid is used. When no information is specified, it is assumed that no Textile HD-EMG grid is used. See the code snippet below for an example on how to configure the different plotters for a recording with a Textile HD-EMG grid. Reordering of the channels is only applied in visualization, not in saving the data. When loading the recorded data, the configuration file can be used (in TMSiSDK/_resources) to reorder the data.

TMSi Python interface < v3.0.0.0	**TMSi Python interface = v.3.0.0.0**

N.A.	cfg = get_config(
	"saga32_config_textile_grid_large”)]
	
	# Impedance plotter
	window = PlottingGUI(plotter_format =
	PlotterFormat.impedance_viewer,
	device = dev,
	layout = 'grid',

	grid_type = '32ch textile grid large')
	
	# Signal plotter
	plot_window = PlottingGUI(
	plotter_format = PlotterFormat.signal_viewer,
	device = dev,
	grid_type = '32ch textile grid large')
	
	# Heatmap plotter
	plot_window = PlottingGUI(
	plotter_format = PlotterFormat.heatmap,
	device = dev,
	tail_orientation = ‘down’,
	grid_type = '32ch textile grid large')

4 EXAMPLES SAGA

The easiest way to start using the interface is to follow the examples. With the examples you will be guided through the main features of the TMSi Python Interface. Connect a TMSi SAGA device to the PC as you would normally do and run the different examples. All examples use the USB and electrical (docked) interface as default interface types.

Example_bipolar_and_auxiliary_measurement.py

This example shows the functionality to display the output of an AUX sensor and the output of a simultaneously sampled BIP channel on the screen. To do so, the channel configuration is updated and a call is made to update the sensor channels. The data is saved to a .Poly5 file.

Example_changing_channel_list.py

This example shows the manipulation of the active channel list and demonstrates how the ChannelName property can be used.

Example_changing_reference_method.py

This example shows how to change the reference method. The configurable options are common reference (ReferenceMethod.common) and average reference (ReferenceMethod.average). When using common reference mode, automatic switching from common to average reference, in case the common reference electrode is out of range, can also be disabled or enabled. This can be configured using ReferenceSwitch.fixed and ReferenceSwitch.auto.

Example_changing_sample_rate.py

This example shows how to change the Base Sample Rate of SAGA, as well as how the sample rate of individual channels can be changed.

Example_config_settings.py

This example shows a brief overview of the different configuration settings of the SAGA device and how to change these settings. More elaborate explanations are given in the examples of the separate configuration items.

Example_EEG_workflow.py

This example shows the functionality of the impedance plotter and the data stream plotter. The example is structured as if an EEG measurement is performed, so the impedance plotter is displayed in head layout. The channel names are set to the name convention of the TMSi EEG cap using a pre-configured EEG configuration. The data is saved to either the Poly5 or XDF file format, depending on user input.

Example_EMG_workflow.py

This example shows the functionality of both the impedance plotter and the HD-EMG heatmap plotter. It is similar to the EEG example, but is modified for a HD-EMG workflow. Note: the example is updated to show the use with a Textile HD-EMG grid.

Example_factory_defaults.py

This example shows how to initiate a factory reset of the SAGA device.

Example_filter_and_plot.py

This example shows how to couple an additional signal processing object to the plotter. The application of a bandpass filter on the first 24 UNI channels is demonstrated. The filter is only applied to the plotter for displaying purposes, the saved data does not contain any filtered data.

Example_impedance_plot.py

This example shows the functionality of the impedance plotter.

Example_load_save_configurations.py

This example shows how to load/save several different configurations from/to a file (in the “TMSiSDK/configs” directory).

Example_read_data.py

This example shows how to read data from a .Poly5 file. Next to this, the reordering strategy for the Textile HD-EMG grid is demonstrated. Finally, conversion to an MNE-Python object is demonstrated.

Example_SD_card_configuration.py

This example shows the functionality to get and set the configuration of the onboard memory of SAGA. The prefix file name is changed in the example.

Example_SD_card_download.py

This example shows how to download card recordings that are stored on SAGA's onboard memory.

Example_SD_card_get_list_of_files.py

This example shows how to retrieve all files that are currently stored on SAGA's onboard memory.

Example_stream_lsl.py

This example shows how to set-up an LSL-stream for SAGA, which can be used in combination with other LSL applications.

Example_switch_dr_interface.py

This example shows how to change the interface between SAGA's Docking Station and Data Recorder.

Example_synchronise_xdf.py

This function shows how to synchronise two data streams in an .xdf file, that has been collected with the LSL-based LabRecorder application.

Example_wifi_measurement.py

This example shows how to perform a measurement over the wireless interface, while data is backed up to the SD card. Finally, the full recording is downloaded from the device.

Example_xdf_to_MNE.py

This example shows how to read data from an .XDF-file and convert it in a data object that can be used in MNE-Python.

5 CHANGES WITH RESPECT TO PREVIOUS VERSION

The current version of the TMSi Python interface includes some new functionalities with respect to the previous version. These changes include:

Added functionality:

- Added possibility to synchronise data collected with LSL.
- Added an installation script to create a virtual environment with all required libraries.
- Added support for the Textile HD-EMG grids.
- Added Wi-Fi support, including back-up logging of data.
- Added support of card recordings, either to start as backup with a measurement, or by starting with a button press.

Improvements and changes

- Re-work of the architecture to make it robust for future adaptations to the interface.
    - Change to the way plotter objects are handled, refer to chapter 3 to see the changes.
- Improved performance of the interface.
- Solved reported bug fixes posted on GitLab.

6 FAQ

An unexpected error occurred. Is there anything I should do before I can start a new measurement?

A: When the script stops running, this does not necessarily mean that the device has stopped sampling. Restarting your Python kernel should ensure that the device stops sampling and is disconnected, after which you can reopen the connection to the device in a new Python kernel.

What parameters/(meta)data can I access when loading a Poly5 file?

A: The following parameters can be retrieved when a Poly5Reader object is created: `samples` (containing the acquired data), `channels` (list containing all information from the Channel class), `sample_rate`, `num_samples`, `num_channels`, `ch_names` (containing all channel names) and `ch_unit_names` (containing all unit names of the used channels).

Why do I have to set a divider for a different sample rate?

A: The sample rate of SAGA device can only be set by the base sample rate divided by a power of 2. For example, with a base sample rate of 4000 Hz, the possible sample rates are 4000, 2000, 1000 or 500 Hz. To get these you will have to set the divider to respectively 1, 2, 4, 8.

`Sample_rate=base_sample_rate/n` with `n=[1, 2, 4, 8]`

What do I need to do if I get "docking station response time-out" after an error?

A: First thing to do is to repower both docking station and data recorder and try again. If this does not work, restart the Python Console, otherwise restart the computer. Also make sure you have followed the steps mentioned in the User Manual of your device for correct installation.

I want to control configurations that are not shown in the examples. How do I know how to control them?

A. Not all functionality of the interface is shown in the examples. More information can be found in the inline documentation of the interface. More information about general control of the SAGA device can be found in the SAGA Device API.

7 CHANGELOG

June 10th, 2021 – Initial release of the TMSi Python Interface.

December 8th, 2021 – Second release of the TMSi Python Interface.

July 29th, 2022 – Third release of the TMSi Python Interface, see chapter 5 for an overview of changes and added functionality.