Python Code for the Automation of the Measurement of Cryogenic LNAs with the Variable Temperature Load Method Using the NFA N8975B

Pablo Collado Soto
Juan Daniel Gallego Puyol

CDT Technical Report 2021-4

Observatorio de Yebes
Apdo. 148 19080 Guadalajara
SPAIN
Phone: +34 949 29 03 11
Fax: +34 949 29 00 63

GOBIERNO DE ESPAÑA MINISTERIO DE TRANSPORTES, MOVILIDAD Y AGENDA URBANA INSTITUTO GEOGRÁFICO NACIONAL cnig

Unión Europea
Fondo Europeo de Desarrollo Regional
"Una manera de hacer Europa"

YEBES

Acknowledgments

Some of the measurement instruments used in the work described in this report were acquired thanks to the YDALGO project financed with Multiregional Operational Programme for Spain 2014-20 ERDF funds.

Contents

1 Abstract **2**

2 Introduction **2**

3 Instrument communication: `SCPI`, `VISA` and `GPIB` **2**
- 3.1 A bit of history . . . 2
- 3.2 `SCPI` commands . . . 3
- 3.3 Instrument Transport Protocols . . . 3
  - 3.3.1 `T&M` specific interfaces . . . 4
  - 3.3.2 General purpose interfaces . . . 4
- 3.4 Providing a common API: `VISA` . . . 4
- 3.5 Skipping `VISA` . . . 4
- 3.6 Our approach . . . 5

4 Code dependencies **5**

5 Environment configuration and initial run **6**
- 5.1 Configuring a `Windows` environment . . . 6
- 5.2 Configuring a `GNU/Linux` environment . . . 7
- 5.3 Configuring a `macOS` environment . . . 7

6 Tool structure **8**
- 6.1 Folder structure . . . 8
  - 6.1.1 ./guis . . . 8
  - 6.1.2 ./goodies . . . 9
  - 6.1.3 ./calib . . . 9
  - 6.1.4 ./presets . . . 9
  - 6.1.5 ./plots . . . 9
  - 6.1.6 ./data . . . 9
- 6.2 Configuration files . . . 9
- 6.3 General parameters: `conf.json` . . . 9
- 6.4 Instrument commands `scpi_commands.json` . . . 10

7 Using the tool **10**

8 Closing thoughts **10**

Listings

1 Installing code dependencies . . . 7
2 Installing code dependencies on to a virtual environment . . . 7
3 Running the tool . . . 8

1 Abstract

This report describes the details of the Python code developed for the automation of the method of the *Variable Temperature Load* (VTL) used for the accurate measurement of the *noise temperature* and *gain* of cryogenic amplifiers. This work is based on a pre-existent `HTBasic` program which was running on an old `Windows-XP` machine for controlling a *Keysight 8975B Noise Figure Meter*, a *Lake Shore 336 Temperature Controller* and an *Agilent 34970A multiplexer*. The new Python code maintains the compatibility with the old calibration table files (in plain ASCII text) and provides the results in a compatible format as well. In addition to the upgrade to a more modern and non-proprietary language, some convenient additional features have been added to improve the user experience. Some details on the Variable Temperature Load (VTL) method and in the instrumentation used can be found elsewhere [1], [2], [3], [4].

2 Introduction

With the increasing complexity of measuring instruments, software has proven to be indispensable to alleviate the work of lab staff. Physical interfaces (i.e. buttons and dials) have turned into touch-enabled graphical interfaces with an ever increasing number of controls. Being able to automate repetitive measuring procedures is bound not only to increase the productivity of technicians, but also to provide a sturdier resilience against human errors.

Software development is characterised by a tremendously fast evolution. It seems only yesterday that programmers had to wrestle with assembly code whilst we now have languages such as `python` that provide an abstract model we can leverage to speed software development up.

The previous paragraph sets the scene for the topic of this technical report. Up until now, lab staff at the *Yebes Astronomical Observatory* were using `HTBASIC`-based programs to interact with measuring equipment. We were tasked with updating programs written with this archaic language into `python` whilst swapping the current text-based interface for a *Graphical User Interface* (`GUI`).

Achieving this objective requires growing accustomed to the communication protocols enabling control of measuring instruments together with the design of a software architecture that allows for easy modifications later down the road. This design idea will contribute to expanding the tool's lifespan as much as possible. We'll also address the choice of dependencies and pre-existing libraries to facilitate our work whilst providing a resilient tool.

Please note the comprehensive `API` documentation is attached at the end of the report.

3 Instrument communication: `SCPI`, `VISA` and `GPIB`

3.1 A bit of history

The automation of *Test and Measurement* `T&M` equipment called for the development of an easy interconnection system between instrument controllers (i.e. a common PC) and the instruments themselves. This would allow user-written scripts and procedures to be seamlessly carried out without the need of an operator "pushing the buttons".

The pioneer of this technology was *Hewlett-Packard (HP)* with the development of the *HP Interface Bus* (HP-IB). This technology would later be standardized by the *IEEE* and become the IEEE-488 standard, better known as *General Purpose Interface Bus* (GPIB).

Just providing a communication infrastructure between controllers and instruments wasn't enough. We still needed some way of telling the instruments **what** to do.

3.2 SCPI commands

In an effort to standardize the syntax of the **commands** driving the instruments the *IEEE* developed the IEEE-488.2 standard to define the *Standard Commands for Programmable Instruments* (SCPI). SCPI is nothing more than a syntax specification that instruments commands adhere to: the commands themselves are **not** standardized. That implies that a two multimeters may be driven by different commands if they are manufactured by different companies, even though they accomplish the exact same goal. The same is true for updates to a given instrument: version x of a logic analyser might use different commands than version y.

The only exception to the aforementioned IEEE-488.2 standard is the existence of the IEEE-488.2 *Common Commands*. These are defined in the standard itself but are not measurement related. They are preceded by an asterisk (*) and let a controller query data such as the instrument's name or the state of it's registers. The most common example is the *IDN? command, which let's a controller find the instruments IDeNtification (i.e. name).

SCPI **commands** are strings composed of several headers separated by colons (:). Commands are active in the sense that they "make" the instrument do something, **queries** on the other hand retrieve information. A command can be turned into a query by appending a question mark (?) at the end. Note that not all commands have a query counterpart and vice versa (an example would be the *IDN?) command we just saw. Please note this is by **no** means a comprehensive guide of SCPI: we have just provided the bare minimum to allow for understanding the program we have written. What's more, the commands one is to employ are **totally dependent** on the instrument to control. Information on them is available on the associated manual.

3.3 Instrument Transport Protocols

When taking a brief look at the history behind the development of controller-instrument communication we mentioned GPIB. GPIB is one of the many examples of transport protocols specific to the T&M industry. SCPI commands rely on these protocols to be transmitted from the controller to the instrument, as well as for data to be transferred in the opposite direction; they are the backbone of controller to instrument communications. These protocols can be classified with regard to the *Input/Output* (I/O) hardware they are supported on.

3.3.1 T&M specific interfaces

The prime example of this class of protocols is GPIB itself. It relies on special connectors that resemble the common parallel connectors that were common for connecting to printers back in the day. Even though these did originally connect to PCs themselves, more modern approaches have appeared. We can leverage "protocol transducers" that can "change" the transport protocol supporting SCPI commands. Common examples are Ethernet to GPIB converters or USB to GPIB converters. These allow a computer to send SCPI commands through it's network stack or USB port, respectively, whilst knowing it'll arrive through a common GPIB interface to the instrument. Despite the I/O interface the computer used is a general purpose one, we are conceptually using GPIB to communicate with the instrument: these converters are *logically transparent*.

3.3.2 General purpose interfaces

An example of a transport protocol employing general purpose interfaces is the *High Speed LAN Instrument Protocol* (HiSLIP), which sits on top of a common TCP/IP network stack. Please do not be confused by the terminology: from the network stack's point of view, HiSLIP is an *application protocol* at the top of the stack. Then, HiSLIP messages are encapsulated within TCP segments which will usually travel as the payload of an Ethernet frame. One can use a network analyser such as WireShark or tcpdump to verify this is indeed true.

3.4 Providing a common API: VISA

With the existence of several protocols, how should a program automating a measure communicate with the instrument? With what we know, we would need to alter the code so as to make it compatible with **every** transport protocol we want to use. That would be an error-prone burden that we luckily don't have to endure thanks to the *Virtual Instrument Software Architecture* (VISA).

VISA provides a common *Application Programming Interface* (API) that user-made programs can leverage. This API makes all the logic dealing with the transport protocol **transparent** to the user. This allows us to only be concerned about the SCPI commands we want to send. We are provided with a common interface we will use no matter the transport protocol in use. This decoupling of the transport and logic planes allows for portability and code reuse that's quite convenient.

In order to employ VISA's facilities we need to have a VISA implementation installed on our machine. We can get a hold of one by installing programs such as the *Keysight IO Libraries* or by just downloading it from entities such as *National Instruments*. Other options like pyvisa-py, a VISA implementation in pure python, are also available. The important aspect is that, in order to leverage VISA, we need to have an implementation.

3.5 Skipping VISA

Several instruments offer the possibility of skipping all the above and just open a (usually) TCP connection to a given port. We can then send raw SCPI commands over that connection to operate the instrument. This scheme relies on the instrument implementing a server that makes those commands effective to then retrieve the results.

Whilst this approach has no dependencies whatsoever (it just needs the ability to open a `socket`) it requires the end user to manage the open sockets and handle any possible errors. They'll also need to manage timeouts, parse the outputs returned by the instrument, ensure no invalid commands are sent so as not to risk crashing the server on the other side, deal with escaping the data...

Managing all these aspects might be worthwhile for very simple scenarios, but one will usually find that they are reinventing the wheel over and over whilst not having access to facilities such as asynchronous interrupts offered by protocols like `GPIB` and `HiSLIP`. It's up to the programmer automating measures to decide what solution is best.

3.6 Our approach

We have decided to leverage `VISA`'s capabilities and install the *Keysight IO Libraries* to do so. We'll also be using the `pyvisa` module to gain access to `VISA`'s functionality from `python` scripts.

4 Code dependencies

The program we have written targets the `Python 3.7.7` interpreter and has a few dependencies that are not part of the `python` distribution itself:

1. **`pyvisa`:** This module *wraps* the `VISA` implementation we have installed on a system so that we can issue `SCPI` commands seamlessly from `python`. It offers a simple interface based on the `write()` and `read()` functions that send commands to the instruments and retrieve results, respectively. It also offers the `query()` function for convenience, which equates to calling the former two functions back to back.

2. **`matplotlib`:** We'll use this well-known library to provide all sorts of visualizations and graphs as they are needed.

3. **`reportlab`:** We'll just scratch the surface of this library to generate `PDF` reports based on the retrieved and processed data.

4. **`win32print`:** We'll use this library to communicate with the printing system in `Windows` so that we can print the generated documents and reports.

5. **`PyQt5`:** This module provides `python` *bindings* for the well-known `Qt GUI` framework so that we can build graphical interfaces for communicating with our program.

Even though they are **not needed** for the program to function, we did find the following two tools to aide in the development of the tool:

1. **`QtDesigner`:** This `GUI` app let's us design our own interfaces using a *drag-and-drop* system. These are later described through `XML` files that we can convert into real `python` *classes* with `pyuic5`.

2. **`pyuic5`:** This `CLI` (*Command Line Interface*) tool reads `XML` files generated by `QtDesigner` and then generates a `python` *class* that we can later import to use the `GUI` as part of our own program.

5 Environment configuration and initial run

Unlike some other languages, `python` does require some setting up to allow us to launch the tool we have developed. This process varies slightly across the different operating systems but the idea behind it is exactly the same:

1. **Install a `python` interpreter:** We need the `python` interpreter to be able to execute `python` scripts.

2. **Install the required dependencies:** We need to acquire the dependencies we listed in the previous section. Due to it's cross-platform nature we decided to use the `pip` package manager.

We'll now detail how to set up the stage for both `Windows` (our target operating system) and `GNU/Linux` systems:

5.1 Configuring a `Windows` environment

Whilst it's true that we can install the `python` interpreter directly through an executable acquired at python.org we decided to use the WinPython distribution to ease the deployment of the finished product to target workstations. This distribution provides an interpreter of its own bundled with all the needed libraries as well as the ones we install ourselves. We just need to run a command prompt from the distribution (i.e. folder) itself so that the `path` the shell looks for the interpreter and modules is altered. Instead of checking the default system directories characteristic of a regular python installation, our shell will search the distribution's folders to try and resolve any imports we might issue. This approach is a "clumsy" version of the *virtual environments* we can create through the `python3 -m venv <venv_name>` command. They are just altering the system's path temporarily to allow us to store libraries in different directories. Then, as long as we copy the entire *WinPython* distribution directory to an arbitrary computer, we'll be able to run out tool without any problems.

Once the *WinPython* distribution is up and running we need to acquire the required dependencies. We can do so through the `pip` package manager. `Pip` looks for modules in the *Python Package Index* and then makes them available to our local installation. If run from *WinPython*'s shell or from a *virtual environment*, the installed modules will **not** be visible to the rest of the system. This helps us craft a reproducible environment that won't conflict with any system-wide installation.

Installing all the dependencies is a matter of executing:

```
python3 -m pip install --upgrade pyvisa matplotlib \
    reportlab win32print PyQt5
```

Listing 1: Installing code dependencies

5.2 Configuring a `GNU/Linux` environment

Most major linux-based distributions such as *Debian* and *Ubuntu* come bundled with a `python3` interpreter. That's why we need only install the dependencies. Should the interpreter not be present we can run `apt install python3` on *Debian-based* systems to download and install the interpreter.

Once the interpreter is present we can install packages either through the standard `apt` package manager or through `pip`, just like before. Due to it's system-agnostic nature, we have decided to leverage `pip`, which we can do with the same command as the one shown on listing 1. If not executed from within a virtual environment, the previous command will install the modules to the **system**: they'll be available to every script executed on the machine. In order to install then on to a confined virtual environment you can execute:

```
# Create the nfa_tool virtual environment
python3 -m venv nfa_tool

# Activate the environment
source nfa_tool/bin/activate

# Install the dependencies on the virtual environment
python3 -m pip install --upgrade pyvisa matplotlib \
    reportlab win32print PyQt5

# (Optional) Deactivate the virtual environment
deactivate
```

Listing 2: Installing code dependencies on to a virtual environment

Leveraging `apt` for this task implies the installation will be system-wide whilst risking not running the most recent version of the packages. As the package names much vary from distribution to distribution we'll only state that these packages *usually* follow the `python3-<package_name>` syntax. Then, installing `pyvisa` could be done through `apt install python3-pyvisa`. We repeat this is **not** and infallible method, nor a comprehensive package list, and we do recommend using `pip` for this task.

5.3 Configuring a `macOS` environment

Due to `macOS` being based on `UNIX`, the installation procedure is practically identical to the one we are to employ on a `GNU/Linux` system. The `python3` does also come bundled with the most recent versions, but we can decide to leverage the Homebrew package manager to install it with `brew install python3`. After that step the process is identical.

Once the dependencies are met we can run the tool by issuing the following command from a shell that's aware of where dependencies are (i.e. the *WinPython* shell):

```
# From a shell with access to the install dependencies and with the
    # current directory being the one where all the source code is located
python3 main.py
```

Listing 3: Running the tool

With the content of listing 3 we should now be greeted with the main menu of the tool we have developed. If that's not the case we encourage you to make sure the needed dependencies are installed and that the shell you are running said command from is aware of the location of those dependencies within the system.

6 Tool structure

The tool can be broken down into two different packages. One of them handles all the `GUI` related stuff and implements the logic driving the program's operation. This is what we have deemed the `GUI package`. There is a second one gathering all the instrument handling logic together with functionality ranging from the printing of documents to the loading of data from text files with an old format. All the scripts providing these facilities have been bundled in the `goodies package`.

We have prepared a comprehensive API description of our tool in which we go through each and every `function`, `method` and `class`. We believe anyone looking for information will be satisfied with its contents. Please note that references to the *DUT* within the API documentation refer to the *Device Under Test* (i.e. the amplifier we are characterizing).

A key aspect to take into account is the folder structure used for the organization of both the source and data files together with the purpose of configuration files. We'll devote the next subsections to these two topics.

6.1 Folder structure

We'll assume `.` refers to the directory containing the `main.py` file. As seen on the previous section, this file is the program's entry-point, that is, the file we are to run to start the program itself. We should start off by stating that this `.` directory contains the so called `GUI package`: all the source files implementing parts of the `GUI` live in this directory. Together with them we find the `conf.json` and `scpi_commands.json` files that contain key configuration parameters. We'll look further into them in the following subsection.

6.1.1 ./guis

This directory contains the `gui` files generated by `pyuic5`. As stated before, `pyuic5` takes the `XML` files generated by `QtDesigner` as input to then convert them to real `python` classes. These files are **not to be modified**, as `pyuic5` will overwrite them when run. These **have not been included** in the API documentation due to them not having any logic whatsoever. The main reason behind using `QtDesigner` was to clearly separate the logic from the `GUI` itself in the first place.

6.1.2 ./goodies

This directory directory which contains all the files integrating the `goodies package`. These have all been documented on the API documentation.

6.1.3 ./calib

This directory houses several text files containing data to be loaded into the program. These values are necessary when processing the data measured with the instrument. The `NDB_X.txt`, `ATT_X.txt`, `ATT_X_DISTRIB.txt` and `ATT_X_TEMPE.txt` files loaded in the `goodies package` are all found here.

6.1.4 ./presets

This directory contains a collection of `JSON` files, each holding different combinations of presets the user can load into the program. These presets control everything from the frequency band to measure to the limits to employ when plotting graphs.

6.1.5 ./plots

This directory contains the saved plots in `PNG` format.

6.1.6 ./data

This directory contains the calibrations and measurements performed with the tool. We find the `calibrations` and `measurements` subdirectories and each of those is further subdivided into `PDFs`, `jsons` and `txts` directories. They all contain the same information in different formats as specified by the directory they reside in. It's worth noting that the special `./data/counters.json` file contains the current counters for the calibration and measurement files (each contains an index indentifying them) as well as the last comment that was associated to a measure so that it can later be retrieved.

6.2 Configuration files

The `conf.json` and `scpi_commands.json` files contain certain parameters that need to be configured by the user. These are regular text files, so editing them shouldn't pose a problem. Let's take a look at what each of them defines:

6.3 General parameters: `conf.json`

This file contains very general parameters ranging from the path to look for state files loaded in the instrument to the instrument `VISA` addresses. Given the readability of the `JSON` syntax each of the parameters should be fairly easy to modify.

6.4 Instrument commands `scpi_commands.json`

This file contains each of the `SCPI` commands each instrument can invoke as well as a list of standard `SCPI` commands that'll be made available to every instrument. These commands are totally instrument dependent (except for the standard ones) and should be configured based on the contents of the instruments manual or programmer's reference.

7 Using the tool

Due to the tool having a graphical user interface be believe it's operation to be rather simple. Most of the subtleties underlying its implementation haven been commented on in the `API documentation` and the tool is really verbose when running. By *verbose* we mean it outputs a lot of messages to inform the user of everything that's going or that could concern them. Thus, we believe we are not doing the program any favours if we provide a rather extensive explanation about aspects that are better learnt by using the tool directly.

The only "non-graphical" aspect users should be aware of is the configuration of the `conf.json` and `scpi_commands.json` files we mentioned in the previous section. These are `JSON` formatted files that can be edited with **any** text editor. One should make sure their contents are correct before running the tool or the program might fail to open connections to instruments, among other possible errors. If this aspect is accounted for, the tool should smoothly perform its task.

8 Closing thoughts

We sincerely hope lab personnel can make the most out of this tool. We have tied to be as thorough as possible during the documentation process so that anybody that's tasked with maintaining or enhancing this program has an easier time. Given the way the tool was written an the documentation backing it up we believe this could be the basis for the update of other programs employed in the *amplifiers* laboratory so that there are less obstacles preventing the update of several of the laboratory's computers.

References

[1] I. Malo, J. Gallego, R. Amils, R. García, M. Diez, I. López, and A. Barcia, "Heated Cryogenic Load in Q band (33-50 GHz) Waveguide for Precision Noise Measurements," *CDT Technical Report 2016-3*, 2016. Report available here.

[2] I. Malo, J. Gallego, R. Amils, R. García, M. Diez, I. López, and A. Barcia, "Improved Design of a Q band (33 50 GHz) Cryogenic Heated Load in Waveguide for Precision Noise Measurements with Reduced Heat Capacity," *CDT Technical Report 2016-6*, 2016. Report available here.

[3] J. Gallego, R. Amils, C. D. González, I. López, and I. Malo, "Using Keysight PNA-X and NFA Noise Receivers for Noise Temperature Measurements with Cryogenic Variable Temperature Load," *CDT Technical Report 2019-17*, 2019. Report available here.

[4] C. D. González, R. Amils, J. Gallego, I. López, and I. Malo, "Comparison of Noise Measurement Results with Three Different Cryogenic Variable Temperature Loads," *CDT Technical Report 2020-3*, 2020. Report available here.

Keysight N8975B Measurement Tool API Documentation

Release 1.0.0

Pablo Collado Soto

Apr 06, 2021

CONTENTS:

1 Gui package **1**

1.1 gui.calibration_menu module 1
1.2 gui.document_viewer module 3
1.3 gui.instrument_opening module 4
1.4 gui.main module 5
1.5 gui.measure_worker module 7
1.6 gui.measurement_menu module 8
1.7 gui.plotting_window module 11
1.8 gui.presets_menu module 13

2 Goodies package **15**

2.1 goodies.data_loading module 15
2.2 goodies.data_plotting module 17
2.3 goodies.data_printing module 17
2.4 goodies.data_processing module 18
2.5 goodies.data_storing module 20
2.6 goodies.instrument_handlers module 23
2.7 goodies.parsers module 28

Index **31**

CHAPTER

ONE

GUI PACKAGE

1.1 gui.calibration_menu module

class gui.calibration_menu.**MainWindow**(*parent*, **args*, ***kwargs*)

Bases: `PyQt5.QtWidgets.QMainWindow`, `guis.calibration_gui.Ui_calibrationMenu`

Implements the calibration menu on the GUI interface.

This class presents the users with a menu they can use to calibrate a given DUT. The process will generate data that's later made available to the main menu window's class so that it can be used for subsequent measurements.

last_cal_lables

A dictionary mapping references to text labels in the GUI to keys so that their contents can be easily updated. They display the parameters of the last calibration.

Type `dict`

last_cal_strs

A list containig the strings associated to the labels reference by the values of the *last_cal_labels* dictionary.

Type `list`

parent

Reference to the class implementing the main menu so as to access needed attributes through it.

Type *MainWindow*

cal_conf

A dictionary containing the parameters characterizing the currently loaded calibration.

Type `dict`

tmp_cal_data

A dictionary containing the values for the HOT and COLD powers we have just obtained from a measure.

Type `dict`

delay

Time it took to gather the current calibration results from the instrument. If it's *-1* (o nonsensical value) we know an error has occured.

Type `float`

meas_finished

A flag indicating whether the measure has finished or not. As measures are executed concurrently we need some means of synchronization between them.

Type `bool`

`last_cal`

A dictionary containing the data for the most recent calibration that's stored on a file.

Type `dict`

`measure_error`

Flag signalling whether an error occured during the data acquisition procedure. It lets us controll the wait loops associated to the concurrent chunk of the measurement procedure.

Type `bool`

`calibrationProcess``()`

Carries out the entire calibration procedure.

This method is the backbone of the entire calibration procedure. It will issue the necessary measures on the instruments to then process the data and provide the raw and processed datasets to the main menu window's class so that it can be employed for further measures.

This method also performs all the "low-level" transformations needed for the calibration to succeed, such as the computation of a frequency list "on the fly" based on the confiured presets defining the frequency band to analyse.

The measure is carried out concurrently in an independent thread. This let's us continually refresh the GUI so that the user doesn't feel "lost" at any point.

After finishing the calibation procedure, it will automatically close the calibration menu to return the user to the main menu.

`finishedCalibration``(`*powers*, *delay*, *c_type*, *errs*`)`

Makes data collected by a *MeasureWorker* available to class.

Measures are carried out concurrently. One thread will manage the UI so that it doesn't freeze and the other will command the measure in the instrument and gather the results. Said worker invokes this method upon completion.

We then have to gather the data passed by the worker and make it available to the rest of the class through attributes.

Parameters

- **`powers`** (`list`) – A list containing two lists, one holds the HOT power values and the other contains the COLD power values.
- **`delay`** (`float`) – Time it took to gather the current calibration results from the instrument. If it's *-1* (o nonsensical value) we know an error has occured.
- **`c_type`** (`str`) – A string describing the kind of power (HOT or COLD) we are receiving. This parameter plays almost no role in the case where we use a noise diode as a noise source (i.e. the case for all calibrations).
- **`errs`** (`list`) – A list of two element lists each containing the error code (int) and error text (str) describing the error. These are obtained form the instrument itself through the invokation of the *SYST:ERR?* SCPI command.

`load_last_calibration``()`

Loads the data from the last calibration contained in a file.

This method loads the data for the most recent calibration that's contained in a JSON-formatted file into an intermediate dictionary.

This data might be loaded into the program afterwards by clicking on a GUI button for this very purpose.

The method handles the case where the requested file cannot be opened and updates the GUI accordingly to show that fact to the user.

newCalibration()

Makes the new calibration menu visible to the user.

That's pretty much all there is to see here :P

recoverCalibration()

Loads the most recent calibration into the program.

This mehtod just makes the *last_cal* attribute together with the current *cal_conf* one to the main window's class so that it can be used for subsequent measures.

After loading the last calibration data into the program and displaying an informative message to the user, it automatically closes the window.

`gui.calibration_menu.`**beep**(*n*)

Produces a controllable number of beeps.

This method beeps as many times as indicated by its input argument. It leverages the list comprehension syntax to carry out its purpose on a single line.

Parameters **n** (`int`) – The number of beeps to emit.

1.2 gui.document_viewer module

class `gui.document_viewer.`**MainWindow**(*parent*, **args*, ***kwargs*)

Bases: `PyQt5.QtWidgets.QMainWindow`, `guis.document_viewer_gui.Ui_documentViewer`

Implements the document viewer on the GUI interface.

The document viewer lets users browse calibrations and measurements saved on files both past and present. It displays a text report with its data and offers buttons to generate graphs, superimpose two graphs to allow for a better comparison and also offers the possibility of generating PDF reports that can be printed. Simpler text reports can be printed as well.

parent

Reference to the class impleemnting the main menu so as to access needed attributes through it.

Type *MainWindow*

fname

Path to the currently opened file.

Type pathlib.Path

fType

A string indicating whether the current file contains measurement or calibration data.

Type `str`

_enButtons(*state=None*)

Changes GUI button's state.

Parameters **state** (`bool`, optional) – The state to set the buttons on. If it's not specified, the button's state will be toggled, that is, they will be enabled if they were disabled and vice versa.

closeEvent(*event*)

Sets the state of GUI elements before closing.

Before clsoing the window it disables the buttons and resets several labels.

This method will automatically be called by PyQt when the X button on a window is pressed.

Parameters **`event`** – The close event sent by PyQt. We just need to accept it to let the window close normally.

`generateGraphFile``()`
Generates a PDF report containing a graph of the currently loaded file.

The file will be saved on disk.

`loadCalibration``()`
Load a calibration file.

It gets the filename from the window provided by the getOpenFileName() PyQt method. It handles any exceptions that may occur when opening the file so as not to crash the entire program.

It'll also display the file's data as a texttual report.

`loadMeasurement``()`
Load a measurement file.

It gets the filename from the window provided by the getOpenFileName() PyQt method. It handles any exceptions that may occur when opening the file so as not to crash the entire program.

It'll also display the file's data as a texttual report.

`load_last_measure``()`
Loads the last measure's data.

If the file cannot be found our there is an error opening it we'll silenty return and no data will be displayed.

`printFile``()`
Generates and prints a PDF report of the currently loaded file.

The file will be saved on disk.

`printGraphFile``()`
Generates and prints a PDF report containing a graph of the currently loaded file.

The file will be saved on disk.

`showGraph``()`
Displays a graph of the current file.

`superimposeGraph``()`
Shows a graph superimposing the current file with one provided by the user.

It gets the file to superimpose from the window provided by the getOpenFileName() PyQt method. It handles any exceptions that may occur when opening the file so as not to crash the entire program.

1.3 gui.instrument_opening module

`class` `gui.instrument_opening.`**`MainWindow`**`(`*parent*, **args*, ***kwargs*`)`
Bases: `PyQt5.QtWidgets.QMainWindow`, `guis.instrument_status_gui.Ui_instrumentStatus`

Implements the instrument opening logic.

This class tries to open each of the configured instruments. Upon failure it asks the user whether to continue with the execution or halt the program altogether.

`parent`
Reference to the class impleemnting the main menu so as to access needed attributes through it.

Type *MainWindow*

`open_instruments``()`

Tries to open the configured instruments.

Upon failure, it asks the user whether to halt execution or continue. After opening all the configured instruments the associated window will close automatically.

If a new instrument needs to be added, one of the blocks below needs to be copied and tweaked.

1.4 gui.main module

class `gui.main.`**`MainWindow`**`(`**args*, ***kwargs*`)`

Bases: `PyQt5.QtWidgets.QMainWindow`, `guis.main_gui.Ui_MainWindow`

Implements the main menu on the GUI interface.

This class also serves as a "backbone" containing all the loaded configurations as well as the gathered measurements together with the processed data derived from them.

`child_windows`

References to the different classes implementing the rest of windows integrating the GUI.

Type `dict`

`app`

Reference to the running app allowing for the forced processing of events.

Type QtWidgets.QApplication

`loaded_data`

Collection of lists containing data loaded based on user defined presets as well as the presets themselves.

Type `dict`

`calibration_data`

Collection of lists contaning the gathered calibration data together with the settings used to perform it.

Type `dict`

`measurement_data`

Collection of lists contaning the gathered measurement data together with the settings used to perform it.

Type `dict`

`nfa`

Reference to the noise figure analyzer (NFA) through which to command it.

Type *instrument_handlers.nfa*

`lake_shore`

Reference to the Lake Shore 336 thermometer through which to command it

Type *instrument_handlers.lake_shore_336*

`mpx`

Reference to the HP Multiplexer through which to command it.

Type *instrument_handlers.mpx*

`gen_conf`

General configuration parameters loaded from the *conf.json* file.

Type `dict`

`controller`
Wrapper for *pyvisa.ResourceManager* providing references to instruments where needed.

Type *instrument_handlers.instrument_manager*

`calibration_menu``()`
Loads the last calibration data and then shows the calibration menu.

If no presets are currently loaded, an error is shown and the user can't proceed to calibration.

`document_viewer``()`
Shows the document viewer with the last measure already loaded.

`errorMessage`(*msg*)
Displays an error message to the user.

Due to calling exec_(), this message will block the main window until dismissed.

Parameters **`msg`** (`str`) – Message to display.

`infoMessage`(*msg*)
Displays an informative message to the user.

Due to calling exec_(), this message will block the main window until dismissed.

Parameters **`msg`** (`str`) – Message to display.

`initial_configuration``()`
Carry out initial configurations.

Sets the current date and loads from in the *conf.json* file. It then tries to open the instruments one by one and prompts whether to load a default configuration on the NFA.

`measurement_menu``()`
Checks whether certain criteria are met and then shows the measurement menu.

First, it makes sure that a calibration is currently loaded. It then checks if that calibration data has been recovered, in which case an informative message is shown.

It then checks wheteher key parameters differ between the current calibration data and the current presets. If they do and error message is shown and the measure is aborted.

If the configured frequency band is NOT the same as the one used in the calibration, it checks whether all the points to measure have indeed been calibrated. This can be the case when we have a very precise calibration and a measurement bandwidth with less resolution. If this requirement is NOT met, the measure is aborted. Otherwise, the current calibration data is DECIMATED permanently.

If all the above succeds, the measurement menu is presented to the user.

`plotCalibration``()`
Plots the last calibration and shows it to the user.

`plotMeasurement``()`
Plots the last measurement and shows it to the user.

`presets_menu``()`
Fills in the current presets and shows the presets configuration menu.

`question`(*title*, *msg*, *default='No'*, *calling_child=None*)
Presents the user with a Yes/No question.

The question popup will block the main window until answered.

Parameters

- **`title`** (`str`) – Title for the question window.

- **`msg`** (`str`) – Question for the user.
- **`default`** (`str`) – Default answer that's selected on the GUI.
- **`calling_child`** (`MainWindow`) – Reference to the calling window so that the question message is centered on it.

Returns The answer provided by the user.

Return type QtWidgets.QMessageBox.(Yes/No)

`setCalibrationData`(*cal_data*)
Updates the contents of the *calibration_data* attribute.

These can be obtained from a new calibration or a recovered one.

Parameters **`cal_data`** (*`dict`*) – Calibration data and parameters used when carrying it out.

`setMeasureData`(*meas_data*)
Updates the contents of the *measurement_data* attribute after a new measure.

Parameters **`meas_data`** (*`dict`*) – Measurement data and parameters used when carrying it out.

`setPresets`(*presets*)
Updates the contents of the *loaded_data* attribute with new parameters.

This method is called from the presets_menu.MainWindow() class to update the current user presets to use for calibrations and measurements. If there is an error loading data files from disk, an error is displayed.

Parameters **`presets`** (*`dict`*) – Configured parameters by the users.

1.5 gui.measure_worker module

`class` `gui.measure_worker.`**`MeasureWorker`**(*nfa*, *measure_type*, *n_source*)
Bases: `PyQt5.QtCore.QObject`

Implements a worker that gathers data concurrently.

In order to avoid halting the main GUI loop when we are waiting for the instrument to provide measure results we need to concurrently gathere that data in a second thread. This class implements the worker that'll utilize that thread and then signal the calling class when the instrument has provided the data.

`finished`
A signal instance letting us connect it to a slot on the calling class so that values can be passed between the worker and that calling class.

Type PyQt5.QtCore.pyqtSignal

`nfa`
Reference to the Keysight NFA instrument so that we can command it.

Type *nfa*

`m_type`
A string describing the type of power we are retrieving, be it a HOT or a COLD power.

Type `str`

`n_source`
The type of measure we are carrying out, be it a noise diode or temperature controlled load one.

Type `str`

finished

takeMeasure()

Worker's task.

This method controls the task the worker is to perform. It will then emmit the *finished* signal to make the gathered data available to the caller.

Note the *m_type* attribute is silently ignored when taking a measure with a diode as a noise source.

1.6 gui.measurement_menu module

class `gui.measurement_menu.`**MainWindow**(*parent*, **args*, ***kwargs*)

Bases: `PyQt5.QtWidgets.QMainWindow`, `guis.measurement_gui.Ui_measureMenu`

Implements the measurement menu on the GUI interface.

This class presents the users with a menu they can use to measure a given DUT. The process will generate data that's later made available to the main menu window's class so that it can be used when plotting and generating reports.

parent

Reference to the class impleemnting the main menu so as to access needed attributes through it.

Type *MainWindow*

tmp

A dictionary containing the data gathered with each subsequent call to the *measurement()* method.

Type `dict`

measure_step

An integer controlling the point of reentry on the *measurement()* method for each subsequent call.

Type `int`

delay

Time it took to gather the current calibration results from the instrument. If it's *-1* (o nonsensical value) we know an error has occured.

Type `float`

wait

A flag controlling whether to wait for the temperatures to adapt to newly configured setpoints when performing a hot and cold load measure.

Type `bool`

inDialog

An input dialog in charge of getting a user's comment for a measure.

Type QtWidgets.QInputDialog

commentIndex

Counter letting several methods alter the comment that's currently shown on the comment selection widget implemented by attribute *inDialog*.

Type `int`

meas_finished

A flag indicating whether the measure has finished or not. As measures are executed concurrently we need some means of synchronization between them.

Type `bool`

`measure_type`
A string describing the type of measure we are carrying out, be it a noise diode measure or a heat controlled one.

Type `str`

`manual_th`
A flag controlling whether to use a user provided hot can cold temperature for later data processing. This temperature is provided as a number on an input dialog.

Type `bool`

`_commentHandler``()`
Handles all the subtleties related to acquiring a comment for a measure.

This method initializes several attributes with sane defaults and displays the last known value on the comment input dialog implemented by attribute *inDialog*.

It'll also try to store the comment input by the user.

Returns The comment input by the user.

Return type str

`_store_n_retrieve_last_comment`(*mode='retrieve'*, *index=- 1*, *comment=None*)
Either retrieves a past comment for a measure or stores the current one.

This method let's the caller recover any of the previously saved comments or store the current one. The method also checks that the comment to store is not the same as the last one of the list so as to save space.

This method will also create a new list for storing comments should one not be present on the auxiliary *counters.json* file. It'll also create a blank file if *counters.json* itself is not present.

Parameters

- **`mode`** (`str`) – Controls whether we are to retireve or store a comment.
- **`index`** (`int`) – The index to look for in the saved comments list.
- **`comment`** (`str`, optional) – The comment to store if we are to store a comment. This parameter is not needed if we are to retrieve a comment.

Returns

It returns a string containing the last comment if we are retrieving it or an empty string if an error (such as the file with containing the saved commnet not being found) is encountered. If we are storing the last comment, nothing will be returned.

Return type str/None

Raises **`IndexError`** – If the index determining the comment to recover is out of bounds we'll let the caller handle it. That exception is used for bounding the *commentIndex* attribute that controls history navigation.

`changeSetpointsNDelays``()`
Allows changing the temperature setpoints and associated delays.

When using a heat controlled load for measures we need to effectively change the load's physical temperature. The Lake Shore 336 model allows us to configure a setpoint that will in tur modify the load's temperature.

As temperature is a continuous magnitued we need to wait some time for this change to stabilize. The program is prepared to wait for a configurable amount of time for that to happen.

This method spawns a set of input dialogs letting the user change both the setpoints and the delays associated to them.

`eventFilter`(*source_object*, *event*)

Lets this MainWindow instance handle keystrokes through a filter.

We have enabled a "history search" function that lets the user recover any previous comments associated with a measure. The controls are similar to those of a regular shell (i.e. Key_Up cycles the history backward and Key_Down goes forward). The problem is that the keystrokes would normally be handled by the *QtWidgets.QInputDialog* instance that's executing at the top of the "loop pile" (note we invoked its event loop with the *self.inDialog.exec_()* statement). We then need to "get in between" and register those keystrokes before they're dismissed by the input dialog. We can do so by means of an *eventFilter* we installed on the input dialog with line *self.inDialog.installEventFilter(self)* in the constructor.

The previous paragraph can be sumarized by saying that any events sent to the input dialog (even draw events) are intercepted by this method, which will be called for each of them. We'll then look for *Key_Up* and *Key_Down* events to act accordingly.

Parameters

- **`source_object`** (`QtCore.QObject`) – The object whose events we are intercepting.
- **`event`** (`QtCore.QEvent`) – The event we are to process.

Returns

A flag indicating whether to filter (i.e. stop) the event (True) or let other's process it.

Return type bool

`finishedMeasure`(*powers*, *delay*, *m_type*, *errs*)

Makes data collected by a *MeasureWorker* available to class.

Measures are carried out concurrently. One thread will manage the UI so that it doesn't freeze and the other will command the measure in the instrument and gather the results. Said worker invokes this method upon completion.

We then have to gather the data passed by the worker and make it available to the rest of the class through attributes.

Parameters

- **`powers`** (`list`) – A list containing two lists, one holds the HOT power values and the other contains the COLD power values.
- **`delay`** (`float`) – Time it took to gather the current calibration results from the instrument. If it's *-1* (o nonsensical value) we know an error has occured.
- **`m_type`** (`str`) – A string describing the kind of power (HOT or COLD) we are receiving. This parameter let's us know where to save the received power when using a heat controlled load as a noise source.
- **`errs`** (`list`) – A list of two element lists each containing the error code (int) and error text (str) describing the error. These are obtained form the instrument itself through the invokation of the *SYST:ERR?* SCPI command.

`hotNColdMeasure`()

Checks condtions for a mesure are met and displays the measure control menu.

As a temperature controller is crucial for a temperature controlled load measure the method checks it's present and then modifies the GUI's appearance to prepare for a new measure.

`measurement`()

Carries out the entire measurement procedure.

This method is the backbone of the entire measurement procedure. It will issue the necessary measures on the instruments to then process the data and provide the raw and processed datasets to the main menu window's class so that it can be employed when plotting or generating reports.

This method uses a re-entrant pattern. Once the user clicks a button the measurement procedure will be bootstrapped and a step will be carried out. We'll then wait for the user to hit that same button again to proceed to the next step and so on. That's why the *measurement_step* attribute is crucial: it let's us know where to re-enter.

Note the above is only true for the case where we use a temperature controlled load as a noise source. In the case where we use a noise didode the method will re-enter automatically so that there is no further need for user interaction. What's more, measures with a noise diode take advantage of the fact that we can gather both the HOT and COLD power values in a single frequency sweep due to the fast switching capabilites inherent to diodes.

This method also performs all the "low-level" transformations needed for the measurement to succeed, such as the computation of a frequency list "on the fly" based on the confiured presets defining the frequency band to analyse.

The measure is carried out concurrently in an independent thread. This let's us continually refresh the GUI so that the user doesn't feel "lost" at any point.

After finishing the measurement procedure, it will automatically close the measurement menu to return the user to the main menu.

noiseDiodeMeasure()

Adapts the GUI for a new noise diode measure and bootstraps it.

After modifying the GUI's appearance, the *measurement()* method will be called, hence starting the entire measurement procedure.

ready_new_measure()

Resets the measurement GUI and attributes for a new measure.

skip_wait()

Changes the state of the *wait* flag to skip the setpoint-related delay.

gui.measurement_menu.**beep**(*n*)

Produces a controllable number of beeps.

This method beeps as many times as indicated by its input argument. It leverages the list comprehension syntax to carry out its purpose on a single line.

Parameters **n** (`int`) – The number of beeps to emit.

1.7 gui.plotting_window module

1.7.1 Embedding in Qt

Simple Qt application embedding Matplotlib canvases. This program will work equally well using Qt4 and Qt5. Either version of Qt can be selected (for example) by setting the `MPLBACKEND` environment variable to "Qt4Agg" or "Qt5Agg", or by first importing the desired version of PyQt.

This is entirely based on the source file found at: https://matplotlib.org/stable/gallery/user_interfaces/embedding_in_qt_sgskip.html

class gui.plotting_window.**ApplicationWindow**(*parent*)

Bases: `PyQt5.QtWidgets.QMainWindow`

Implements the plotting window for the entire GUI.

This class implements a window that will be used as a canvas for every graph we are to draw in the program. We are leveraging *matplotlib*'s object-oriented model instead of the imperative one that resembles the likes of *Matlab* so accomplish this.

The idea is to provide reusable axes objects and clear them before every plot so that things don't get messy with superimposed graphs. They key here is that we can decide not to clean the axes and easily superimpose graphs as well!

`_main`
Main widget holding our plot canvas.

Type QWidget

`layout`
Main widget's layout letting us place a canvas within it.

Type QVBoxLayout

`parent`
Reference to the class impleemnting the main menu so as to access needed attributes through it.

Type *main.MainWindow*

`static_canvas`
The canvas we'll draw graphs on.

Type FigureCanvas

`_temp_ax`
Axes belonging to the *static_canvas* figure where we'll draw noise temperature graphs.

Type Axes

`_gain_ax`
Axes belonging to the *static_canvas* figure where we'll draw gain graphs.

Type Axes

`closeEvent`(*event*)
Clears the axes to prepare them for a subsequent plot.

This method will automatically be called by PyQt when the X button on a window is pressed. Thus, we'll just clear the axes before the window itself is closed so that the next plot finds a perfectly clean canvas.

Parameters **`event`** – The close event sent by PyQt. We just need to accept it to let the window close normally.

`plotFile`(*presets*, *data*, *mode*, *superimpose=False*)
Draws a graph on the instance's axes.

This method can be called from anywhere and will draw whateevr is passed as parameters on this instance's axes.

Parameters

- **`presets`** (`dict`) – Dictionary containing graphic options such as the frequency for the band markers and graphic limits.
- **`data`** (`dict`) – Dictionary containing the data to plot both on the X and Y axes. That is, it contains the appropriate magnitudes together with the frequencies for those values.
- **`mode`** (`str`) – A string representing the type of data to plot, be it a calibration or a measurement.

- **superimpose** (bool, optional) – A flag controlling whether to superimpose the graph described by the *data* parameter on an existing one. This is used when comparing curves.

1.8 gui.presets_menu module

class gui.presets_menu.**MainWindow**(*parent*, **args*, ***kwargs*)

Bases: PyQt5.QtWidgets.QMainWindow, guis.presets_gui.Ui_presetMenu

Implements the preset selection menu on the GUI interface.

This class presents the users with a menu where they can configure the presets to use for subsequent calibrations and measurements. It lets them retrieve a given configuration from disk and save a new one to a file as well.

data_fields

A dictionary mapping the references to the input boxes where data will be input to different keys so that the values can be easily retrieved when needed.

Type dict

presets

A dictionary containing the configured presets. These are only updated when the user explicitly saves them or he/she loads a new configuration from a saved file.

Type dict

parent

Reference to the class impleemnting the main menu so as to access needed attributes through it.

Type *MainWindow*

skipMsg

A flag controlling whether an informative message is shown when closing this menu.

Type bool

closeEvent(*event*)

Closes the window whilst reminding the user to save the presets.

The method displays a message to the users to remeind them that they need to have loaded the presets for them to become effective in the rest of the program. It let's the user avoid the closing of the window to continue modifying or to load the presets.

The *closeEvent()* method is "automagically" called by PyQt when closing (i.e. clicking the X button) a window.

Parameters **event** – The close event sent by PyQt. We just need to accept it to let the window close normally.

closeNotSaving()

Closes the window without informing the user.

Through the *skipMsg* attribute we manage to inhibit the display of a message telling the user the need to have explicitly loaded the presets for them to be employed for the rest of operations.

fill_preset_data()

Fills in the different input boxes with the current presets.

When this method is called it'll try to fill in the currently loaded presets into the different input boxes. The *data_fields* attribute helps make this method much shorter.

`generatePresets``()`

Updates the current presets with the current data.

This method parses the content of each and every input to try and updated the current presets. It notifies parsing error to its caller through the returned value.

The *data_fields* attribute makes this method that much shorter.

Returns

Either a dictionary containing the current presets or *None* if a parsing error was encountered.

Return type dict/None

`getPresets``()`

Returns a copy of the presets.

We return a copy so that modifications in the parent don't affect our own attribute. As dictionaries are passed by reference this could become an issue otherwise.

Returns A dictionary containing the current presets.

Return type `dict`

`loadPresets``()`

Loads presets from a file provided by the user.

It gets the filename from the window provided by the getOpenFileName() PyQt method. It handles any exceptions that may occur when opening the file so as not to crash the entire program.

It then fills in the different input boxes with the values contained in the file.

`passPresets``()`

Passes a copy of the presets to the Main Menu Window.

The method tries to update the currently loaded presets. If it succeeds, it passes a copy to the class implementing the Main Menu Window which centralizes the references to the important data structures.

`savePresets``()`

Saves the currently configured presets to a file.

It gets the filename from the window provided by the getSaveFileName() PyQt method. It handles any exceptions that may occur when saving the file.

It will only save the current presets if there are no errors when parsing them.

CHAPTER

TWO

GOODIES PACKAGE

2.1 goodies.data_loading module

`goodies.data_loading.`**`load_data_file`**`(`*path*, *parser*, *delim=None*`)`

Loads data from a file into a dictionary.

The dictionary is dinamically created based on the keys a given parser has configured. The file is then read and the dictionary filled as needed.

Note dictionaries are passed by reference. That's why this way of calling the *read_file()* is working as intended.

Parameters

- **`path`** (`str`) – A string containing the path to the file we are to load.
- **`parser`** (`function`) – The parser capable of interpreting the contents of the filed identified by *path*.
- **`delim`** (`str`, optional) – A delimiter to use for splitting each of the lines in the file. The function breaks lines on whitespaces if not specified.

Returns

A dictionary containing the data extracted from the file specified in the *path* parameter.

Return type dict

`goodies.data_loading.`**`load_last_calibration`**`(`*dump_json=False*`)`

Loads the last calibration from an internal use file.

This function is NOT CURRENTLY BEING USED. It's been left should something similar be needed by someone. As awe are storing data in a JSON format there is no need to parse a non-standard format like the one the original program was using for this purpose.

When the calibration is being carried out data will be stored to an intermediate file called *last_cal.txt* which is NOT intended for humans to read. This function loads it so that it can later be used.

Parameters **`dump_json`** (`bool`, optional) – A flag controlling whether to dump the loaded data to a JSON file.

Returns A dictioanry containing the loaded data.

Return type dict

`goodies.data_loading.`**`load_measure`**`(`*measure_num=0*`)`

Loads a measurement from a text file.

This function is NOT CURRENTLY BEING USED. It's been left should something similar be needed in the future to retrive measures saved with the old format instead of as a JSON file.

Parameters **measure_num** (int) – Measurement number to load.

Returns Loaded measurement data.

Return type dict

goodies.data_loading.**load_ndb_calib**(*t_num=0*)
Loads the NDB_x.txt file specified by the *t_num* parameter.

Parameters **t_num** (int) – The *X* especifying with *NDB_X.txt* file to load.

Returns A dictionary containing the loaded data from the file.

Return type dict

goodies.data_loading.**load_ndb_n_att_files**(*t_num=0*, *log_mode=False*, *dump_json=False*, *dbg=False*)
Loads data files from disk and stores them in several dictionaries.

This function will load files depending on the value of the *t_num* parameter. It will then return them so that they can be used by the rest of the programs.

Parameters

- **t_num** (int) – The nuber identifying the files to load from disk.
- **log_mode** (bool, optional) – Not currently used.
- **dump_json** (bool, optional) – Controls whether to export the loaded data to a JSON file for further processing.
- **dbg** (bool, optional) – Controls whether to print a message with the loaded data for *print()* based debugging. (We know, not the best practice…)

Returns

ndb_data (dict): Loaded data from the NDB_X.txt file.

att_data (dict): Loaded data from the ATT_X.txt file.

att_distrib_data (dict): Loaded data from the ATT_X_distrib.txt file.

att_tempe_data (dict): Loaded data from the ATT_X_tempe.txt file.

MAX_DATE (str): A reference to the module's MAX_DATE attribute.

Return type tuple

goodies.data_loading.**read_file**(*path*, *parser*, *data*, *delimiter*)
Reads a file line by line and extracts the appropriate data.

This method opends the file pointed to by the *path* parameter and reads it line by line. If that line is not a comment it'll be passed to the parser indicated by the *parser* parameter and data will be extracted and stored on the dictionery referenced by the *data* parameter.

Parameters

- **path** (*pathlib.Path*) – Path to the file we are to scan.
- **parser** (function) – Funtion implementing the logic capable of extracting the data for the current file.
- **data** (dict) – The dictionary we are to fill in with the extracted data.
- **delimiter** (str/None) – The character to use as a delimiter for breaking up a line into several fragments containing data. If it's *None*, lines will be broken up at whitespaces.

2.2 goodies.data_plotting module

goodies.data_plotting.**plot_data**(*fig*, *ax_temp*, *ax_gain*, *presets*, *data*, *mode='measurement'*, *save=True*, *create_fig=False*, *legends=False*, *superimposed=False*, *block=True*)

Plots data to a given pair of axes.

This function will plot whatever data is provided in the *data* parameter to the *matplotib axes* specified in the *ax_temp* and *ax_gain* parameters. It will also store the graph as a PNG file by default.

Parameters

- **fig** (*Figure*) – Figure object we are to draw on if not instructed to create a new figure.
- **ax_temp** (*Axes*) – Matplotlib axes we are to draw noise temperature curves on.
- **ax_gain** (*Axes*) –
- **presets** (dict) – A collection of presets controlling how to draw graphs.
- **data** (dict) – A dictionary containing the lists of frequencies and values to plot.
- **mode** (str) – A string determining the kind of data we are plotting (either a measurement or a calibration).
- **save** (bool, optional) – A flag controlling whetcher to store the graph as a PNG file.
- **create_fig** (bool, optional) – A flag controlling whether to draw on the figure passed on the *fig* parameter or to create our own.
- **legends** (bool, optional) – A flag controlling whether to draw a legend.
- **superimposed** (bool, optional) – A flag informing us of whetcher the current graph as specified by the *data* parameter is to be superimposed on an existing one.
- **block** (bool, optional) – A flag determining whether the graph we are to show should block the caller. This is NOT CURRENTLY USED due to the integration of the canvas into the PyQt GUI.

Returns

Either a binary buffer we can use to embed the graph into a PDF file we are to generate or None if we are just showing the graph.

Return type io.BytesIO/None

2.3 goodies.data_printing module

goodies.data_printing.**escape_characters**(*text*)

Escape charcters that could cause problems when generating a PDF.

This method is based on the one found in https://wiki.python.org/moin/EscapingHtml and we have just adapted it to suit our needs.

Parameters **text** (str) – The text to scan and escape.

Returns A string with the scaped characters.

Return type str

goodies.data_printing.**generate_pdf**(*path*, *add_graph=False*)

Generates a PDF report based on the data contained in the specified file.

Parameters

- **`path`** (*`pathlib.Path`*) – The path of the file whose data we are to include in the report.
- **`add_graph`** (`bool`, optional) – A flag controlling whether to include a graph in the PDF report.

Returns The path of the genrated PDF report.

Return type pathlib.Path

`goodies.data_printing.`**`print_document`**(*path*)

Prints the file pointed to by the path.

Parameters **`path`** (*`pathlib.Path`*) – Path of the file to print, be it a PDF or a plain text file.

2.4 goodies.data_processing module

`goodies.data_processing.`**`compute_averages`**(*freqs*, *gains*, *temps*, *n_points*, *mark_1*, *mark_2*)

Computes several metrics in a frequency band.

The frequency band is delimited by the *mark_1* and *mark_2* parameters.

Parameters

- **`freqs`** (`list`) – Measured frequencies in GHz.
- **`gains`** (`list`) – Gains in dBs.
- **`temps`** (`list`) – Noise temperatures in K.
- **`n_points`** (`int`) – Numbes of measured points.
- **`mark_1`** (`float`) – Lower limit of the frequency band of interest in MHz.
- **`mark_2`** (`float`) – Upper limit of the frequency band of interest in MHz.

Returns

t_mean (`float`): Average noise temperature in the band in K.

g_mean (`float`): Average gain in the band in dBs.

t_delta (`float`): Difference between the maximum and minimum noise temperatures in the band in K.

g_delta (`float`): Difference between the maximum and minimum gains in the band in dBs.

t_min (`float`): Minimum noise temperature in the band in K.

g_min (`float`): Minimum gain in the band in dBs.

f_min (`float`): Frequency associated to the minimum noise temperature in GHz.

Return type tuple

`goodies.data_processing.`**`linear_interpolation`**(*data*, *d_name*, *x*, *x_f*)

Linearly interpolates values to fill in "holes" in the original data.

This function linearly interpolates data at indices *x_f* and *x_o* = *x_f* - 1 in the data parameter. It just computes the slope between the *x_f* and x_o values, multiplies it with the difference between the current *x* < *x_f* and *x_o* < *x* and then applies the value at the origin of the interval as an offset.

Parameters

- **`data`** (`dict`) – Dictionary containing the *X* and *Y* values to base the interpolation on.

- **d_name** (`str`) – Key associated the *Y* values.
- **x** (`float`) – Frequency to interpolate.
- **x_f** (`int`) – Index associated to the end of the frequency interval we are interpolating on.

Returns The interpolated value.

Return type float

`goodies.data_processing.`**`load_th_tc_interpolated_calib_tables`**(*presets*, *ndb*)

Initializes the TH, TC and subsequent auxiliary data tables for calibrations.

Loads tables whose data is needed for processing calibrations.

Parameters

- **presets** (`dict`) – Presets containing configured data affecting the generation of intermediate tables.
- **ndb** (`dict`) – Data loaded from *NDB_X.txt* files.

Returns

T_c_cal (`list`): The *T_c_cal* table.

T_h_cal (`list`): The *T_h_cal* table.

ndbp (`list`): The *ndpb* table.

Return type tuple

`goodies.data_processing.`**`load_th_tc_interpolated_meas_tables`**(*prsts*, *ndb*, *att*, *att_distrib*, *att_tempe*)

Initializes the TH, TC and subsequent auxiliary data tables for measures.

Uses data contained in the parametrs, which is obtained from the values in loaded files, to generate tables needed for measure processing.

Parameters

- **prsts** (`dict`) – Presets containing configured data affecting the generation of intermediate tables.
- **ndb** (`dict`) – Data loaded from *NDB_X.txt* files.
- **att** (`dict`) – Data loaded from *ATT_X.txt* files.
- **att_distrib** (`dict`) – Data loaded from *ATT_X_DISTRIB.txt* files.
- **att_tempe** (`dict`) – Data loaded from *ATT_X_TEMPE.txt* files.

Returns

T_c (`list`): The *T_c* table.

T_h (`list`): The *T_h* table.

ldb (`list`): The *ldb* table.

ladb (`list`): The *ladb* table.

ndbp (`list`): The *ndpb* table.

ndbp_cor (`list`): The corrected *ndbp* table.

Return type tuple

goodies.data_processing.**treat_gathered_measures**(*mode*, *n_points*, *pop_freq*, *phot*, *pcold*, *th*, *tc*, *t_cal*, *g_cal*, *enr*, *last_cal_file*)

Processes gathered data so as to extract significant conclusions later on.

Parameters

- **mode** (str) – Identifies whether the data is generated by a measurement or calibration.
- **n_points** (int) – The number of points in the measure.
- **pop_freq** (list) – List of the measured frequencies in GHz.
- **phot** (list) – Measured hot powers in dBm.
- **pcold** (list) – Measured cold powers in dBm.
- **th** (list) – The *th* table.
- **tc** (list) – The *tc* table.
- **t_cal** (list) – Noise temperatures gathered during calibration.
- **g_cal** (list) – Gains gathered during calibration.
- **enr** (list) – ENR factors for measures.
- **last_cal_file** (*_io.TextIOWrapper*) – A file descriptor letting us update the *last_cal.txt* file containing the data for the last calibration so that we can update it.

Returns

f (list): Measured frequencies in Hz.

g (list): Measured gains in dBs.

t (list): Noise temperatures in K.

t_cor (list): Corrected noise temperatures in K.

Return type tuple

goodies.data_processing.**treat_presets**(*prsts*)

Checks the presets and modifies them so that they comply to imposed conditions.

As presets are input by users they might contain incorrect data or poorly chosen values we might need to manually alter. This function takes care of exactly that.

Parameters **prsts** (dict) – A dictionaty containing the raw user input presets.

Returns The compliant (presumably modified) presets.

Return type dict

2.5 goodies.data_storing module

goodies.data_storing.**generate_calibration_header**(*fd*, *path*, *att*, *preamp*, *bw_res*, *i_time*, *t_cold*, *enr_table*)

Fills the data in for the calibration header.

Parameters

- **fd** (*_io.TextIOWrapper*) – A file descriptor pointing to the file we are to write.
- **path** (str) – Absolute path of the file we are writing to.
- **att** (float) – Attenution used during the measure.

- **preamp** (bool) – Indicates whether the internal preamp was used in the measure.
- **bw_res** (float) – Bandwidth resolution used in the measure in Hz.
- **i_time** (float) – Integration time per point in seconds.
- **t_cold** (float) – Configured cold temperature in the presets in K.
- **enr_table** (int) – Confgured ENR table in the presets.

goodies.data_storing.**generate_measurement_header**(*m_type*, *fd*, *t_ampli*, *t_amb*, *path*, *att*, *preamp*, *bw_res*, *i_time*, *comment*, *bias_data*, *t_mean*, *t_min*, *t_max*, *delta_t*, *g_mean*, *g_min*, *g_max*, *delta_g*, *t_cold*, *ndb_table*)

Fills in the data for a measure header.

Parameters

- **m_type** (str) – The type of measure, be it a noise diode or heat controlled load one.
- **fd** (*_io.TextIOWrapper*) – A file descriptor pointing to the file we are to write.
- **t_ampli** (float) – DUT's temperature.
- **t_amb** (float) – Cryostat's temperature.
- **path** (str) – Absolute path of the file we are writing to.
- **att** (float) – Attenution used during the measure.
- **preamp** (bool) – Indicates whether the internal preamp was used in the measure.
- **bw_res** (float) – Bandwidth resolution used in the measure in Hz.
- **i_time** (float) – Integration time per point in seconds.
- **comment** (str) – Measure comment.
- **bias_data** (list) – List containing the bias data as floats.
- **t_mean** (float) – Average noise temperature in the band of interest in K.
- **t_min** (float) – Minimum noise temperature in the band of interest in K.
- **t_max** (float) – Maximum noise temperature in the band of interest in K.
- **delta_t** (float) – Difference between the maximum and minimum noise temperatures in the band in K.
- **g_mean** (float) – Average gain in the band of interest in dBs.
- **g_min** (float) – Minimum gain in the band of interest in dBs.
- **g_max** (float) – Maximum gain in the band of interest in dBs.
- **delta_g** (float) – Difference between the maximum and minimum gains in the band in dBs.
- **t_cold** (float) – Configured cold temperature in the presets in K.
- **ndb_table** (int) – Confgured NDB table in the presets.

goodies.data_storing.**make_dir**(*path*)

Creates a new directory.

Parameters **path** (*pathlib.Path*) – Path where we are to create the new directory.

`goodies.data_storing.`**`retrieve_n_update_counters`**(*mode*, *update=True*)

Retrives the file counters for a new file and optionally updates them.

Parameters

- **`mode`** (`str`) – Determines whether to regard the calibration or measurement counters.
- **`update`** (`bool`, optional) – A flag controlling whether to update the counters or not.

Returns The requested counter or -1 or 0 in case of error.

Return type int

`goodies.data_storing.`**`save_last_calibration`**(*nfa_conf*, *entry*, *o_file=None*, *header=False*)

Saves the last calibration as an unformatted file for error recovery.

The *last_cal.txt* file contains the unformatted last calibration data that we can use in case of program crashes. This file is NOT CURRENTLY USED by our program, as the recovered calibrations are extracted from JSON files.

Parameters

- **`nfa_conf`** (`dict`) – The instrument's configuration when the calibration was performed.
- **`entry`** (`list`) – Values (floats) to store.
- **`o_file`** (*`_io.TextIOWrapper, optional`*) – File descriptor pointing to the file to write.
- **`header`** (`bool`) – Flag controlling whether we are to write the file's header or a data entry.
- **`Returns`** –

  _io.TextIOWrapper/None: Returns the file descriptor to the *last_cal.txt* file if we are
  writing the header and None otherwise.

`goodies.data_storing.`**`store_calibration_data`**(*nfa_conf*, *presets*, *f*, *g*, *tcal*, *phot*, *pcold*, *th*, *tc*, *enr*, *recover_last_cal=False*)

Generates reports for calibrations.

It'll store both a text report along with a JSON file containing both the data displayed on the text report together with measure "metadata" that's useful for later analysis.

Parameters

- **`nfa_conf`** (`dict`) – The instrument's configuration when the calibration was performed.
- **`presets`** (`dict`) – Presets loaded when the measure was taken.
- **`meas_data`** (`dict`) – Data gathered during the measure.
- **`f`** (`list`) – Calibrated frequencies in GHz.
- **`g`** (`list`) – Gathered gains in dBs.
- **`tcal`** (`list`) – Gathered noise temperatures in K.
- **`phot`** (`list`) – Gathered hot powers in dBm.
- **`pcold`** (`list`) – Gathered cold powers in dBm.
- **`th`** (`list`) – Th table for calibration.
- **`tc`** (`list`) – Tc table for calibration.
- **`enr`** (`list`) – ENR table for calibration.
- **`recover_last_cal`** (`bool`, optional) – Flag controlling whether to write the calibration to a file containing an index in its name or to the *C_recovered.txt* file.

`goodies.data_storing.`**`store_measurement_data`**(*nfa_conf*, *presets*, *meas_data*)

Generates reports for measures.

It'll store both a text report along with a JSON file containing both the data displayed on the text report together with measure "metadata" that's useful for later analysis.

Parameters

- **`nfa_conf`** (`dict`) – The instrument's configuration when the measure was taken.
- **`presets`** (`dict`) – Presets loaded when the measure was taken.
- **`meas_data`** (`dict`) – Data gathered during the measure.

2.6 goodies.instrument_handlers module

`class` `goodies.instrument_handlers.`**`instrument`**(*instrument*, *cmds*)

Bases: `object`

Base class for all instruments.

It provides wrappers for the *write()*, *read()* and *query*()* methods so that the instruments are returned to local mode once a command is sent to them. We can do so thanks to the manual control of the REN line on the GPIB bus.

`name`

The name of the instrument.

Type `str`

`inst`

A reference to the instance letting us communicate with the instrument.

Type pyvisa.Resource

`cmds`

A dictionary containing the commands available to this instrument as defined on the *scpi_commands.json* file.

Type `dict`

`query`(*cmd*)

Queries the instrument and returns it to local mode.

This method will return the read data as a *string*, so we will be responsible for casting it to a suitable data type depending on its use.

Parameters **`cmd`** (`str`) – The SCPI command to execute on the instrument.

Returns The read data.

Return type str

`query_ascii_values`(*cmd*)

Queries the instrument and returns it to local mode.

This method will automatically cast the read data to *float* by default. PyVisa's original method accepts a myriad of arguments that can control from the type used for casting to the expression to apply to parse the received textual data.

Parameters **`cmd`** (`str`) – The SCPI command to execute on the instrument.

Returns The read data as a list of *floats*.

Return type list

release()
Releases this instrument.

write(*cmd*)
Writes a command to the instrument and returns it to local mode.

Parameters **cmd** (`str`) – The SCPI command to execute on the instrument.

Returns Number of bytes written.

Return type int

class `goodies.instrument_handlers.`**instrument_manager**(*instrument_addresses*, *commands*, *backend=''*, *timeout=600000*)
Bases: `object`

Implements a manager capable of returning references to objects.

This manager will return references to measuring instruments when requested. It'll also make the instrument's commands available to the instance and configure several parameters such as the instrument's timeout.

I also keeps track of the open instruments so that they can be clenaly closed on program exit.

rm
The resource manager letting use open instruments through the VISA backend we are using.

Type pyvisa.ResourceManager

i_addresses
VISA addresses for the available instruments as configured in the *conf.json* file.

Type `dict`

cmds
Available SCPI commands for the instruments as contained on the *scpi_commands.json* file.

Type `dict`

timeout
Timeout to configure for newly opened instruments in ms.

Type `float`

instruments
List of references to open instruments.

Type `list`

_check_connectivity(*inst*)
Checks connectivity with an instrument.

Parameters **inst** (*pyvisa.Resource*) – A reference to the instrument to check.

_open_resource(*instrument_name*)
Opens a resource.

This method makes the supported commands available to the instrument.

Parameters **instrument_name** (`str`) – String identifying the instrument to open.

_select_resource()
Selects an instrument to open.

This method is NOT CURRENTLY USED.

Returns A reference to the open instrument.

Return type pyvisa.Resource

`open_instrument`(*instrument='nfa'*)
Opens an instrument.

This method handles error related to the opening of the instrument so that the program doesn't crash unexpectedly.

Parameters **`instrument`** (`str`) – A string identifying the instrument to open.

Returns

It returns a reference to the instrument if no problems where encountered. It'll return None otherwise so that the caller knows the instrument couldn't be acquired.

Return type pyvisa.Resource/None

`release_instruments`()
Gracefully closes the connection to every open instrument.

It then shuts down the resource manager.

class `goodies.instrument_handlers.`**`lake_shore_218`**(*instrument*, *cmds*)
Bases: *`goodies.instrument_handlers.instrument`*

Class controlling the Lake Shore 218s cryogenic temperature thermometer.

`get_temperature`(*sensor=1*)
Gets a sensor's temperature.

Parameters **`sensor`** (`int`) – The index for one of the thermometer's sensors. It MUST be a value between 1 and 8.

Returns The temperature for the specified sensor in K or None in case of error.

Return type float/None

class `goodies.instrument_handlers.`**`lake_shore_336`**(*instrument*, *cmds*)
Bases: *`goodies.instrument_handlers.instrument`*

Class controlling the Lake Shore 336 cryogenic temperature controller.

`get_setpoint`()
Get's the current setpoint.

Returns Either the current temperature setpoint in K or None in case of error.

Return type float/None

`get_temperature`()
Gets the DUt's and cryostat's temperature.

Returns

load_temp (float/None): The cryostat's temperature in K or None in case of error.

amp_temp (float/None): The DUT's temperature in K or None in case of error.

Return type tuple/None

`set_setpoint`(*target_temp*)
Sets the setpoint to a given value.

Parameters **`target_temp`** (`float`) – The setpoint to configure in K.

Returns 0 in case everything went well and -1 in case of error.

Return type int

class `goodies.instrument_handlers.`**`mpx`**(*instrument*, *cmds*)

Bases: *`goodies.instrument_handlers.instrument`*

Class controlling the Agilent 34970A Data Acquisition Unit.

`get_bias_data`()

Gathers the DUT's bias data.

Returns

A list of floats describing the voltages (in V) and the currents (in mA) of each of the bias points for each of DUT's amplification stages. If an error is encountered when requesting the data, None will be returned.

Return type list/None

class `goodies.instrument_handlers.`**`nfa`**(*instrument*, *cmds*)

Bases: *`goodies.instrument_handlers.instrument`*

Class controlling the Keysight NFA N8975B Noise Figure Analyzer.

`_check_errors`()

Checks the NFA's erro queue.

Returns

A list of the erros where each entry is composed by an error code and an error description or None if there were no erros in the queue.

Return type list/None

`general_reset`(*sf*, *ef*, *n*, *bwr*)

Resets the instrument to the sate defined by the parameters.

Parameters

- **`sf`** (`float`) – Lower frequency for the swept band in MHz.
- **`ef`** (`float`) – Upper frequency for the swept band in MHz.
- **`n`** (`int`) – Number of points in the swept band.
- **`bwr`** (`float`) – Bandwidth resolution in Hz.

`get_status`()

Gets the currently configured parameters in the NFA.

Returns

A dictionary containing the curren parameters configured in the NFA.

Return type dict

`load_config`(*dir*, *file*)

Loads a state file saved in the instrument.

The parameters are not directly supplied by the user. They are obtained from the *conf.json* file.

Parameters

- **`dir`** (`str`) – Path to the directory containing the state file.
- **`file`** (`str`) – Name of the state file to load.

prepare_hot_cold_measure(*sf*, *ef*, *n*, *it*, *att*, *g*, *bwr*)
Configures the NFA to perform a hot and cold load measure.

Parameters

- **sf** (float) – Lower frequency for the swept band in MHz.
- **ef** (float) – Upper frequency for the swept band in MHz.
- **n** (int) – Number of points in the swept band.
- **it** (float) – Integration time per point in seconds.
- **att** (int) – Attenuation in dBs.
- **g** (bool) – Flag indicating whether to use the internal preamplifier or not.
- **bwr** (float) – Bandwidth resolution in Hz.

set_attenuation(*att*)
Sets the attenuation.

This attenuation is provided by attenuators internal to the NFA.

Parameters **att** (int) – Attenuation in dBs.

set_bw_resolution(*bwr*)
Sets the bandwidth resolution for the sweeps.

Parameters **bwr** (float) – Bandwidth resolution in Hz.

set_integration_time(*it*)
Sets the integration time per point in a sweep.

Parameters **it** (float) – Integration time per point in seconds.

set_preamp_gain(*g*)
Configures the internal preamplifier.

Parameters **g** (bool) – Flag indicating whether to use the internal preamplifier or not.

set_start_freq(*sf*)
Sets the lower frequency for the swept band.

Parameters **sf** (float) – Lower frequency for the sweep in MHz.

set_stop_freq(*ef*)
Sets the upper frequency for the swept band.

Parameters **ef** (float) – Upper frequency for the sweep in MHz.

set_sweep_points(*n*)
Sets the number of points in a sweep.

Parameters **n** (int) – The number of points.

take_heated_load_measure(*mode*)
Takes a measure when using a heat controlled load.

This method is called when using a heat controlled load as a noise source. This method is called twice in a measure procedure as we cannot switch the noise source (the heat controlled load) from a cold to a hot state and vice versa in a short time. That forces us to first take a measure with a hot load and then take a second one with a cold one.

Parameters **mode** (str) – Indicates whether we are gathering the hot or cold power values. This let's use change the configuration of the NFA accordingly.

Returns

p_read (`list`): The gathered powers (hot or cold) for each frequency point in dBm. If a fatal error is encountered, this will be an empty list.

delay (float/list): The time it took the measure to complete. If an unrecoverable error was encountered, this delay will be *-1*. If the NFA emitted non-fatal erros, this will be a list of those erros (each entry being an error code and an error description).

Return type tuple

`take_noise_diode_measure``()`

Takes a measure when using a noise diode

This method is called when using a noise diode as a noise source. As a diode can switch quickly between states we can sweep the frequency range once and gather both the cold and hot powers "simultaneously". This allows us to call this method only once instead of twice.

Returns

read_powers (`list`): The gathered powers for each frequency point in dBm. If a fatal error is encountered, this will be an empty list. Both the cold and hot powers are present in this list.

delay (float/list): The time it took the measure to complete. If an unrecoverable error was encountered, this delay will be *-1*. If the NFA emitted non-fatal erros, this will be a list of those erros (each entry being an error code and an error description).

Return type tuple

2.7 goodies.parsers module

Parsers capable of extracting data from text files.

This module's functions implement the logic capable of extracting data from the different file formats we need to work with. They are passed to functions in the *data_loading* module as a parameter so that they are called for each line in a given file. The following sections describe he supported file formats.

Generalities: Note that all the preamble lines are preceded by an exclamation mark (!).

NDB_X.txt: Freq [MHz] NDB [dB]

ATT_X.txt: Freq [MHz] Input Line Loss [dB] Attenuator [dB]

ATT_X_distrib.txt: L300_60 L60_15 L15_15 (Note the added total must be 1)

ATT_X_tempe.txt: A B C T_2 T_1 (SS Line) A B C T_2 T_1 (Attenuator)

`goodies.parsers.`**`NDB_KEYS`**

List of strings identifying the data fileds (i.e. columns) to extract from NDB_X.txt files.

Type `list`

`goodies.parsers.`**`ATT_KEYS`**

List of strings identifying the data fileds (i.e. columns) to extract from ATT_X.txt files.

Type `list`

`goodies.parsers.`**`ATT_DISTRIB_KEYS`**

List of strings identifying the data fileds (i.e. columns) to extract from ATT_DISTRIB_X.txt files.

Type `list`

`goodies.parsers.`**`ATT_TEMPE_KEYS`**

List of strings identifying the data fileds (i.e. columns) to extract from ATT_TEMPE_X.txt files.

Type `list`

`goodies.parsers.`**`LAST_CAL_HEADER`**

List of strings identifying the data fileds (i.e. columns) to extract from the header (i.e. first line) of the *last_cal.txt* file.

Type `list`

`goodies.parsers.`**`LAST_CAL_KEYS`**

List of strings identifying the data fileds (i.e. columns) to extract from *last_cal.txt* file.

Type `list`

`goodies.parsers.`**`MEASURE_KEYS`**

List of strings identifying the data fileds (i.e. columns) to extract from M_X.txt files.

Type `list`

`goodies.parsers.`**`att`**(*entry='', data={}, l_no=0, ret_keys=False*)

Parser extracting data from ATT_X.txt files.

Parameters

- **`entry`** (`list`) – A list of strings, each containing a value to extact.
- **`data`** (`dict`) – Dictionary to store the extracted data on.
- **`l_no`** (`int`) – The current line number in the file we are parsing. Not used in this parser.
- **`ret_keys`** (`bool`) – A flag indicating whether to just return the keys for this file type (so that we can initialize a dictionary for storing the extracted data) or proceed with parsing.

`goodies.parsers.`**`att_distrib`**(*entry='', data={}, l_no=0, ret_keys=False*)

Parser extracting data from ATT_X_DISTRIB.txt files.

Parameters

- **`entry`** (`list`) – A list of strings, each containing a value to extact.
- **`data`** (`dict`) – Dictionary to store the extracted data on.
- **`l_no`** (`int`) – The current line number in the file we are parsing. Not used in this parser.
- **`ret_keys`** (`bool`) – A flag indicating whether to just return the keys for this file type (so that we can initialize a dictionary for storing the extracted data) or proceed with parsing.

`goodies.parsers.`**`att_tempe`**(*entry='', data={}, l_no=0, ret_keys=False*)

Parser extracting data from ATT_X_TEMPE.txt files.

Parameters

- **`entry`** (`list`) – A list of strings, each containing a value to extact.
- **`data`** (`dict`) – Dictionary to store the extracted data on.
- **`l_no`** (`int`) – The current line number in the file we are parsing
- **`ret_keys`** (`bool`) – A flag indicating whether to just return the keys for this file type (so that we can initialize a dictionary for storing the extracted data) or proceed with parsing.

`goodies.parsers.`**`last_calibration`**(*entry='', data={}, l_no=0, ret_keys=False*)

Parser extracting data from the *last_cal.txt* file.

Parameters

- **`entry`** (`list`) – A list of strings, each containing a value to extact.
- **`data`** (`dict`) – Dictionary to store the extracted data on.
- **`l_no`** (`int`) – The current line number in the file we are parsing
- **`ret_keys`** (`bool`) – A flag indicating whether to just return the keys for this file type (so that we can initialize a dictionary for storing the extracted data) or proceed with parsing.

`goodies.parsers.`**`measure`**(*entry='', data={}, l_no=0, ret_keys=False*)

Parser extracting data from M_X.txt files.

Parameters

- **`entry`** (`list`) – A list of strings, each containing a value to extact.
- **`data`** (`dict`) – Dictionary to store the extracted data on.
- **`l_no`** (`int`) – The current line number in the file we are parsing. Not used in this parser.
- **`ret_keys`** (`bool`) – A flag indicating whether to just return the keys for this file type (so that we can initialize a dictionary for storing the extracted data) or proceed with parsing.

`goodies.parsers.`**`ndb`**(*entry='', data={}, l_no=0, ret_keys=False*)

Parser extracting data from NDB_X.txt files.

Parameters

- **`entry`** (`list`) – A list of strings, each containing a value to extact.
- **`data`** (`dict`) – Dictionary to store the extracted data on.
- **`l_no`** (`int`) – The current line number in the file we are parsing. Not used in this parser.
- **`ret_keys`** (`bool`) – A flag indicating whether to just return the keys for this file type (so that we can initialize a dictionary for storing the extracted data) or proceed with parsing.

INDEX

Symbols

_check_connectivity() (*goodies.instrument_handlers.instrument_manager method*), 24
_check_errors() (*goodies.instrument_handlers.nfa method*), 26
_commentHandler() (*gui.measurement_menu.MainWindow method*), 9
_enButtons() (*gui.document_viewer.MainWindow method*), 3
_gain_ax (*gui.plotting_window.ApplicationWindow attribute*), 12
_main (*gui.plotting_window.ApplicationWindow attribute*), 12
_open_resource() (*goodies.instrument_handlers.instrument_manager method*), 24
_select_resource() (*goodies.instrument_handlers.instrument_manager method*), 24
_store_n_retrieve_last_comment() (*gui.measurement_menu.MainWindow method*), 9
_temp_ax (*gui.plotting_window.ApplicationWindow attribute*), 12

A

app (*gui.main.MainWindow attribute*), 5
ApplicationWindow (*class in gui.plotting_window*), 11
att() (*in module goodies.parsers*), 29
att_distrib() (*in module goodies.parsers*), 29
ATT_DISTRIB_KEYS (*in module goodies.parsers*), 28
ATT_KEYS (*in module goodies.parsers*), 28
att_tempe() (*in module goodies.parsers*), 29
ATT_TEMPE_KEYS (*in module goodies.parsers*), 28

B

beep() (*in module gui.calibration_menu*), 3
beep() (*in module gui.measurement_menu*), 11

C

cal_conf (*gui.calibration_menu.MainWindow attribute*), 1
calibration_data (*gui.main.MainWindow attribute*), 5
calibration_menu() (*gui.main.MainWindow method*), 6
calibrationProcess() (*gui.calibration_menu.MainWindow method*), 2
changeSetpointsNDelays() (*gui.measurement_menu.MainWindow method*), 9
child_windows (*gui.main.MainWindow attribute*), 5
closeEvent() (*gui.document_viewer.MainWindow method*), 3
closeEvent() (*gui.plotting_window.ApplicationWindow method*), 12
closeEvent() (*gui.presets_menu.MainWindow method*), 13
closeNotSaving() (*gui.presets_menu.MainWindow method*), 13
cmds (*goodies.instrument_handlers.instrument attribute*), 23
cmds (*goodies.instrument_handlers.instrument_manager attribute*), 24
commentIndex (*gui.measurement_menu.MainWindow attribute*), 8
compute_averages() (*in module goodies.data_processing*), 18
controller (*gui.main.MainWindow attribute*), 5

D

data_fields (*gui.presets_menu.MainWindow attribute*), 13
delay (*gui.calibration_menu.MainWindow attribute*), 1
delay (*gui.measurement_menu.MainWindow attribute*), 8
document_viewer() (*gui.main.MainWindow method*), 6

E

errorMessage() (*gui.main.MainWindow method*), 6
escape_characters() (*in module goodies.data_printing*), 17
eventFilter() (*gui.measurement_menu.MainWindow method*), 10

F

fill_preset_data() (*gui.presets_menu.MainWindow method*), 13
finished (*gui.measure_worker.MeasureWorker attribute*), 7
finishedCalibration() (*gui.calibration_menu.MainWindow method*), 2
finishedMeasure() (*gui.measurement_menu.MainWindow method*), 10
fname (*gui.document_viewer.MainWindow attribute*), 3
fType (*gui.document_viewer.MainWindow attribute*), 3

G

gen_conf (*gui.main.MainWindow attribute*), 5
general_reset() (*goodies.instrument_handlers.nfa method*), 26
generate_calibration_header() (*in module goodies.data_storing*), 20
generate_measurement_header() (*in module goodies.data_storing*), 21
generate_pdf() (*in module goodies.data_printing*), 17
generateGraphFile() (*gui.document_viewer.MainWindow method*), 4
generatePresets() (*gui.presets_menu.MainWindow method*), 13
get_bias_data() (*goodies.instrument_handlers.mpx method*), 26
get_setpoint() (*goodies.instrument_handlers.lake_shore_336 method*), 25
get_status() (*goodies.instrument_handlers.nfa method*), 26
get_temperature() (*goodies.instrument_handlers.lake_shore_218 method*), 25
get_temperature() (*goodies.instrument_handlers.lake_shore_336 method*), 25
getPresets() (*gui.presets_menu.MainWindow method*), 14
goodies.data_loading
    module, 15
goodies.data_plotting
    module, 17
goodies.data_printing
    module, 17
goodies.data_processing
    module, 18
goodies.data_storing
    module, 20
goodies.instrument_handlers
    module, 23
goodies.parsers
    module, 28
gui.calibration_menu
    module, 1
gui.document_viewer
    module, 3
gui.instrument_opening
    module, 4
gui.main
    module, 5
gui.measure_worker
    module, 7
gui.measurement_menu
    module, 8
gui.plotting_window
    module, 11
gui.presets_menu
    module, 13

H

hotNColdMeasure() (*gui.measurement_menu.MainWindow method*), 10

I

i_addresses (*goodies.instrument_handlers.instrument_manager attribute*), 24
inDialog (*gui.measurement_menu.MainWindow attribute*), 8
infoMessage() (*gui.main.MainWindow method*), 6
initial_configuration() (*gui.main.MainWindow method*), 6
inst (*goodies.instrument_handlers.instrument attribute*), 23
instrument (*class in goodies.instrument_handlers*), 23
instrument_manager (*class in goodies.instrument_handlers*), 24
instruments (*goodies.instrument_handlers.instrument_manager attribute*), 24

L

lake_shore (*gui.main.MainWindow attribute*), 5
lake_shore_218 (*class in goodies.instrument_handlers*), 25
lake_shore_336 (*class in goodies.instrument_handlers*), 25
last_cal (*gui.calibration_menu.MainWindow attribute*), 1
LAST_CAL_HEADER (*in module goodies.parsers*), 29
LAST_CAL_KEYS (*in module goodies.parsers*), 29
last_cal_lables (*gui.calibration_menu.MainWindow attribute*), 1
last_cal_strs (*gui.calibration_menu.MainWindow attribute*), 1
last_calibration() (*in module goodies.parsers*), 29
layout (*gui.plotting_window.ApplicationWindow attribute*), 12
linear_interpolation() (*in module goodies.data_processing*), 18
load_config() (*goodies.instrument_handlers.nfa method*), 26
load_data_file() (*in module goodies.data_loading*), 15
load_last_calibration() (*gui.calibration_menu.MainWindow method*), 2
load_last_calibration() (*in module goodies.data_loading*), 15
load_last_measure() (*gui.document_viewer.MainWindow method*), 4
load_measure() (*in module goodies.data_loading*), 15
load_ndb_calib() (*in module goodies.data_loading*), 16
load_ndb_n_att_files() (*in module goodies.data_loading*), 16
load_th_tc_interpolated_calib_tables() (*in module goodies.data_processing*), 19
load_th_tc_interpolated_meas_tables() (*in module goodies.data_processing*), 19
loadCalibration() (*gui.document_viewer.MainWindow method*), 4
loaded_data (*gui.main.MainWindow attribute*), 5
loadMeasurement() (*gui.document_viewer.MainWindow method*), 4
loadPresets() (*gui.presets_menu.MainWindow method*), 14

M

m_type (*gui.measure_worker.MeasureWorker attribute*), 7
MainWindow (*class in gui.calibration_menu*), 1
MainWindow (*class in gui.document_viewer*), 3
MainWindow (*class in gui.instrument_opening*), 4
MainWindow (*class in gui.main*), 5
MainWindow (*class in gui.measurement_menu*), 8
MainWindow (*class in gui.presets_menu*), 13
make_dir() (*in module goodies.data_storing*), 21
manual_th (*gui.measurement_menu.MainWindow attribute*), 9
meas_finished (*gui.calibration_menu.MainWindow attribute*), 1
meas_finished (*gui.measurement_menu.MainWindow attribute*), 8
measure() (*in module goodies.parsers*), 30
measure_error (*gui.calibration_menu.MainWindow attribute*), 2
MEASURE_KEYS (*in module goodies.parsers*), 29
measure_step (*gui.measurement_menu.MainWindow attribute*), 8
measure_type (*gui.measurement_menu.MainWindow attribute*), 9
measurement() (*gui.measurement_menu.MainWindow method*), 10
measurement_data (*gui.main.MainWindow attribute*), 5
measurement_menu() (*gui.main.MainWindow method*), 6
MeasureWorker (*class in gui.measure_worker*), 7
module
    goodies.data_loading, 15
    goodies.data_plotting, 17
    goodies.data_printing, 17
    goodies.data_processing, 18
    goodies.data_storing, 20
    goodies.instrument_handlers, 23
    goodies.parsers, 28
    gui.calibration_menu, 1
    gui.document_viewer, 3
    gui.instrument_opening, 4
    gui.main, 5
    gui.measure_worker, 7
    gui.measurement_menu, 8
    gui.plotting_window, 11
    gui.presets_menu, 13
mpx (*class in goodies.instrument_handlers*), 26
mpx (*gui.main.MainWindow attribute*), 5

N

n_source (*gui.measure_worker.MeasureWorker attribute*), 7
name (*goodies.instrument_handlers.instrument attribute*), 23
ndb() (*in module goodies.parsers*), 30
NDB_KEYS (*in module goodies.parsers*), 28
newCalibration() (*gui.calibration_menu.MainWindow method*), 2

nfa (*class in goodies.instrument_handlers*), 26
nfa (*gui.main.MainWindow attribute*), 5
nfa (*gui.measure_worker.MeasureWorker attribute*), 7
noiseDiodeMeasure() (*gui.measurement_menu.MainWindow method*), 11

O

open_instrument() (*goodies.instrument_handlers.instrument_manager method*), 25
open_instruments() (*gui.instrument_opening.MainWindow method*), 5

P

parent (*gui.calibration_menu.MainWindow attribute*), 1
parent (*gui.document_viewer.MainWindow attribute*), 3
parent (*gui.instrument_opening.MainWindow attribute*), 4
parent (*gui.measurement_menu.MainWindow attribute*), 8
parent (*gui.plotting_window.ApplicationWindow attribute*), 12
parent (*gui.presets_menu.MainWindow attribute*), 13
passPresets() (*gui.presets_menu.MainWindow method*), 14
plot_data() (*in module goodies.data_plotting*), 17
plotCalibration() (*gui.main.MainWindow method*), 6
plotFile() (*gui.plotting_window.ApplicationWindow method*), 12
plotMeasurement() (*gui.main.MainWindow method*), 6
prepare_hot_cold_measure() (*goodies.instrument_handlers.nfa method*), 26
presets (*gui.presets_menu.MainWindow attribute*), 13
presets_menu() (*gui.main.MainWindow method*), 6
print_document() (*in module goodies.data_printing*), 18
printFile() (*gui.document_viewer.MainWindow method*), 4
printGraphFile() (*gui.document_viewer.MainWindow method*), 4

Q

query() (*goodies.instrument_handlers.instrument method*), 23
query_ascii_values() (*goodies.instrument_handlers.instrument method*), 23
question() (*gui.main.MainWindow method*), 6

R

read_file() (*in module goodies.data_loading*), 16
ready_new_measure() (*gui.measurement_menu.MainWindow method*), 11
recoverCalibration() (*gui.calibration_menu.MainWindow method*), 3
release() (*goodies.instrument_handlers.instrument method*), 24
release_instruments() (*goodies.instrument_handlers.instrument_manager method*), 25
retrieve_n_update_counters() (*in module goodies.data_storing*), 21
rm (*goodies.instrument_handlers.instrument_manager attribute*), 24

S

save_last_calibration() (*in module goodies.data_storing*), 22
savePresets() (*gui.presets_menu.MainWindow method*), 14
set_attenuation() (*goodies.instrument_handlers.nfa method*), 27
set_bw_resolution() (*goodies.instrument_handlers.nfa method*), 27
set_integration_time() (*goodies.instrument_handlers.nfa method*), 27
set_preamp_gain() (*goodies.instrument_handlers.nfa method*), 27
set_setpoint() (*goodies.instrument_handlers.lake_shore_336 method*), 25
set_start_freq() (*goodies.instrument_handlers.nfa method*), 27
set_stop_freq() (*goodies.instrument_handlers.nfa method*), 27
set_sweep_points() (*goodies.instrument_handlers.nfa method*), 27
setCalibrationData() (*gui.main.MainWindow method*), 7
setMeasureData() (*gui.main.MainWindow method*), 7
setPresets() (*gui.main.MainWindow method*), 7
showGraph() (*gui.document_viewer.MainWindow method*), 4
skip_wait() (*gui.measurement_menu.MainWindow method*), 11
skipMsg (*gui.presets_menu.MainWindow attribute*), 13
static_canvas (*gui.plotting_window.ApplicationWindow attribute*), 12
store_calibration_data() (*in module goodies.data_storing*), 22

store_measurement_data() (*in module goodies.data_storing*), 23
superimposeGraph() (*gui.document_viewer.MainWindow method*), 4

T

take_heated_load_measure() (*goodies.instrument_handlers.nfa method*), 27
take_noise_diode_measure() (*goodies.instrument_handlers.nfa method*), 28
takeMeasure() (*gui.measure_worker.MeasureWorker method*), 8
timeout (*goodies.instrument_handlers.instrument_manager attribute*), 24
tmp (*gui.measurement_menu.MainWindow attribute*), 8
tmp_cal_data (*gui.calibration_menu.MainWindow attribute*), 1
treat_gathered_measures() (*in module goodies.data_processing*), 19
treat_presets() (*in module goodies.data_processing*), 20

W

wait (*gui.measurement_menu.MainWindow attribute*), 8
write() (*goodies.instrument_handlers.instrument method*), 24