Main Module Reference

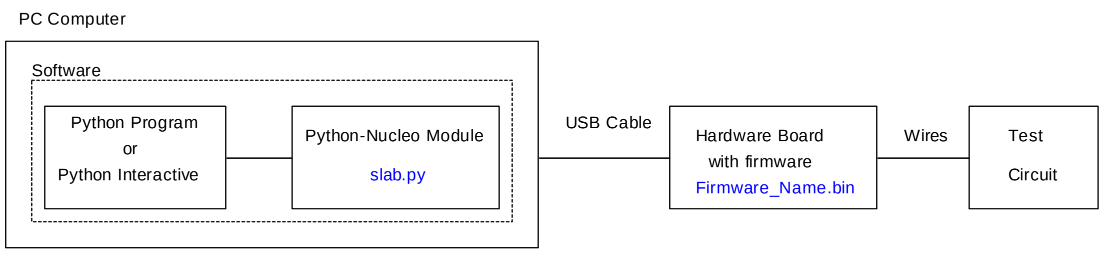
This document is the basic reference for the Python interface to the SLab system. It describes the main module associated to the slab.py file.

Index

Introduction

This document describes the Small Lab Python System, referred from now on a shorter name "Slab System". This system is designed so that it can assist in the measurement of several kinds of test circuits in order to assist on electronics learning and design on a low budget.

The Slab System, at least, requires the four elements shown in the following figure:



First, we need to have two software elements: A Python 2.7 interpreter and a slab.py Python module. The module depends on Python modules of the SciPy package like NumPy and Matplotlib. You can always include those modules on a standard Python distribution but the easiest way is to choose a Python distribution that includes the required modules. In particular the slab.py module has been confirmed to work with the Anaconda Python 2.7 distribution that can be obtained from the reference below:

https://www.continuum.io/downloads

The slab.py module can be obtained from the same place where you obtained this document.

Together with the software elements, we need also some hardware elements. First we will need a hardware board that implements the electrical measurements requested by the Python code on the Test circuit we want to measure.

Hardware Board

The hardware board is an electronic circuit that includes a microcontroller and several ADC (Analog to Digital Converter) and several DAC (Digital to Analog Converter) usually included as part of the microcontroller. The board can also provide several digital I/O lines.

The communication between the PC hand the Hardware Board is done trough a serial COM link typically implemented on a USB connection so you will also need an USB cable. The operation uses a client-server model. The board is a server that waits for the PC client to make a request. When the board receives a request, it performs the requested measurements and returns the result to the PC. After that, it waits for another command.

The system includes CRC checks on the serial channel so that, in case of communication errors, you get exception errors instead of corrupted data. Communication, however, is not fully protected with timeouts. That means that a communication error can leave the Python module or the Hardware Board indefinitely waiting for data on the serial channel. In that case you shall reset the board and restart the python code.

Care has been taken to minimize the work the board must perform shifting as much work as possible to the PC. That is reasonable as the PC is faster and contains much more memory than the board.

Communication protocols between the board and the PC have been standardized so that, following the standard, you can develop your own board to perform the measurements. Any board that includes at least 2 DACs and 4 ADCs and a serial connection can be used to interface the SLab Python scripts if it is loaded with the proper firmware. Upon connection the Python module interrogates the board about its features son no centralized control of the board is needed.

One of the options available for the hardware board is to use a STM32 Nucleo Board. Those boards, are provided by STMicroelectronics as a way to demonstrate their range or ARM Cortex 32 bit of microcontrollers. They all include a serial over USB link and are easy to program using a simple drag and drop operation. All are programmable using the online MBED compiler, so it is easy to develop a firmware for them. Not all Nucleo boards feature DACs, so only a subset of them are adequate to work on the SLab

Currently, MBED firmwares have been developed for two Nucleo boards: the STM32-Nucleo-L152RE and the STM32-Nucleo-F303RE.

In order to communicate with the Slab Python module, the board needs to host a proper firmware. For the supported boards, the firmware, in a file with **.bin** extension, can be obtained from the same place that hosts this document and the slab.py file.

Both the slab.py module and the board firmware feature a version code composed of two numbers separated by a point like "1.0". The number before the point is the major version number and the number after the point are the minor version number.

Major number versions increase when more commands are added to the firmware whereas minor number versions correct errors without adding new commands. In this reference it will be shown, for each command, the minimum version of the firmware required for it to work.

Major versions of the slab.py module and the board firmware are developed together so, for best operation, the Nucleo firmware shall have the same or greater major number than the Python module.

As it was said before, the supported board firmwares have been developed in the MBED compiler. The source code will be releseased together with the binary code. Additional information about the compiler can be found in the MBED site:

https://developer.mbed.org/

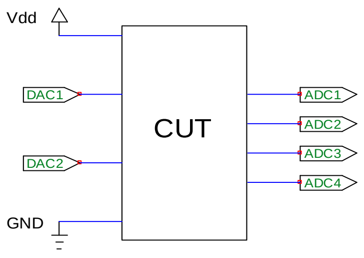
In order to load the firmware on the board just connect the board to the PC and, after the drivers are loaded, it will show up as a mass storage disk. Just drag and drop the firmware file, with ***.bin*** extension, over this disk to install the firmware on the board.

The board connects with a test circuit we want to measure. The connections between the board and the test circuit will use wires between the board and the circuit.

The whole purpose the Slab Python System is being able to program the operation of six pins on the hardware board: two DAC pins DAC1 and DAC2 that can force a voltage on the circuit and four ADC pins ADC1, ADC2, ADC3 and ADC4 that can read the circuit voltage. The particular name and location of those pins can change from one board to another. Some board firmwares can feature more DACs or ADCs but two DACs and four ADCs is the minimum to be compatible with the Slab module.

Buffered measurements

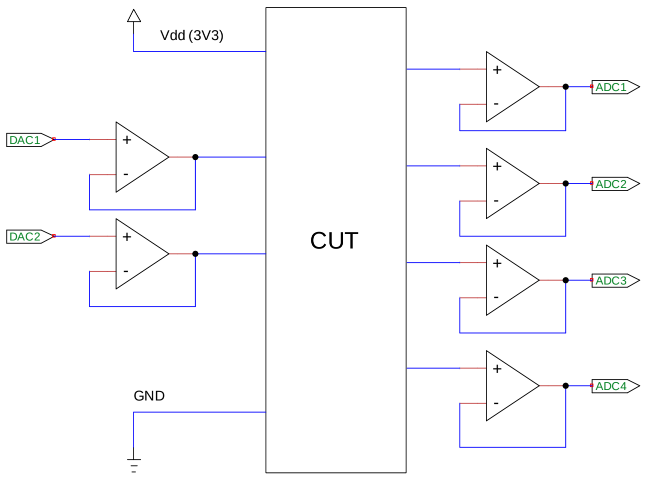
Measurements with the Slab System can be performed basically with unbuffered or buffered I/Os. Unbuffered means that you connect the DAC outputs and ADC inputs directly to the circuit under test (CUT) as shown in the following figure:



The main limitation of the unbuffered approach is that DAC outputs usually have finite output resistance and the ADC inputs consume current from the CUT while they are measuring. For low impedance nodes DAC outputs cannot provide enough current and for high impedance nodes ADC input currents modify the operation of the circuit itself. That means that the unbuffered approach is only valid when DACs are connected to high impedance nodes and ADCs are connected to low impedance nodes.

As an example, in a STM32 Nucleo F303RE board we have measured an output resistance of about 12kΩ in the DAC channel 1 at pin A2. This is in concordance with the maximum 15kΩ specified in the microcontroller datasheet. That means that any circuit you connect to this DAC needs to have a much greater input resistance. In another example we have measured a voltage divider formed by two 680kΩ resistors. Due to the input current on the ADC, the output voltage is about a 42% of the input instead of the correct 50% value. The main problem with the ADC is that it doesn't behave as a resistor. Charge is drawn at every conversion son the errors due to the circuit output resistance depends on the number and speed of the measurements.

Buffered measurements means that you connect the DAC outputs and ADC inputs to the circuit under test (CUT) using intermediate buffer follower circuits as shown in the following figure:



As CMOS operational amplifiers in follower configuration can have very high input impedance and very low output impedance, the buffered approach greatly increases the range of circuits we can measure properly.
Follower circuits, however, can add their own errors but they can be corrected usually by calibration and are less problematic that the errors that the buffers correct.
As boards like the nucleo ones operate at 3.3V, it is important to use full rail opamps whenever possible to prevent limitations on de dynamic range.
Examples of adequate opamps could be the microchip MCP6002 (2 AO in one chip) or MCP6004 (4 AO in one chip) but there are a lot of other possible candidates.
For each supported board, a buffered setup will be proposed to ease the use of the buffered solution.

It is strongly recommended to use a buffered setup if you want to obtain good measurements on the circuit under test. In the case of unbuffered measurements you need to guarantee the impedance requirements of ADCs and DACs.

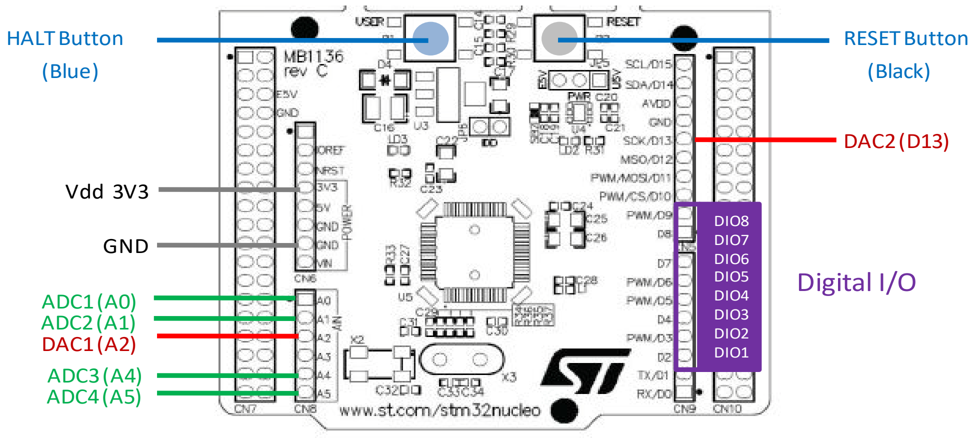
Currently Supported Boards

This section describes the boards officially supported by the SLab System. Officially supported means that a firmware for these boards are provided from the same location that the slab.py module and the documentation. As information about the board capabilities is obtained from the board itself, it is easy to develop a firmware for other boards without changing the slab.py module.

Currently, there are two boards officially supported by the SLab System.

STM32 Nucleo64 F303RE

This board features a STM32F303RE microcontroller in a 64 pin package. You can get more information about this board on the STMicroelectronics and the MBED sites.
The board includes connections to all I/O pins of the included microcontroller. Current version 1.x firmware, however, only uses eight analog pins and eight digital I/O pins.



The following table shows the analog pins we will use to interface the test circuit.

Pin	SLab Name	Kind	Functionality
GND	GND	Supply	Common reference for all voltages
3V3	Vdd		3,3V Constant supply voltage
A0	ADC1	ADC	Analog input: Can read analog voltage of connected node Read voltages must be bounded to the GND to Vdd range between 0.0V and 3.3V
A1	ADC2		
A4	ADC3		
A5	ADC4		
A2	DAC1	DAC	Analog output: Can force analog voltage of connected nodes D13 range is limited to 0.0V to 1.5V. A2 can go from 0.0V to 3.3V
D13	DAC2		

Aside from the analog pins we have eight digital I/O pins from DIO1 to DIO8 that correspond with pins D2 to D9 on the board.

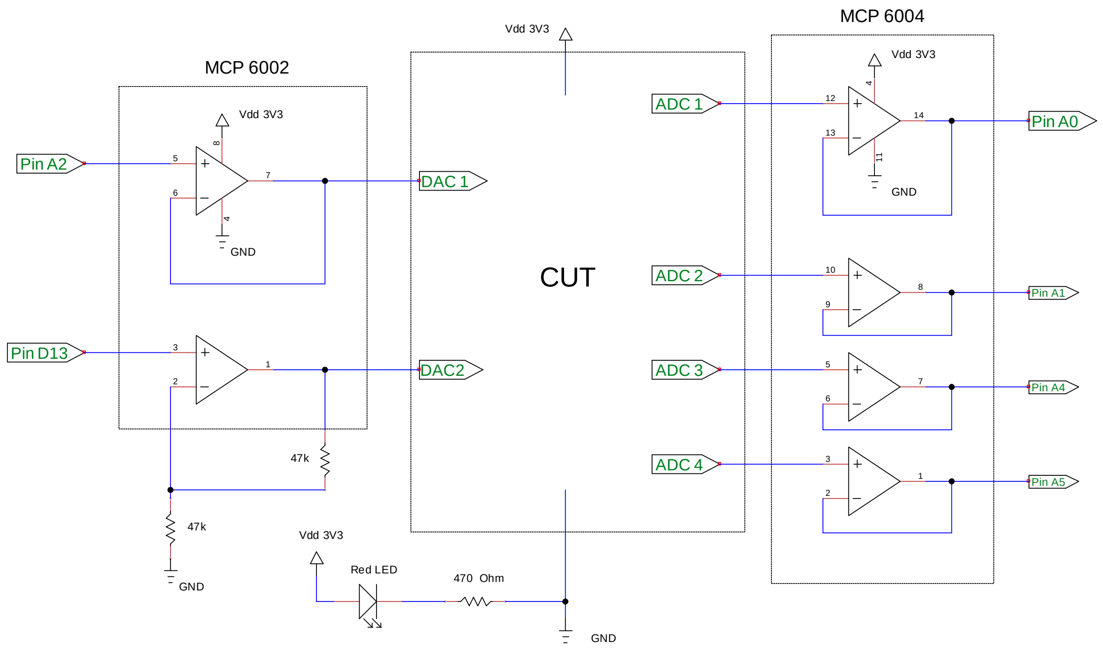
The board also includes a black **reset** button and a blue user button that is used for the **halt** signal.
The black **reset** button interrupts all communication and is nearly equivalent to unplug and plug again the board from the computer. That usually means that you need to reconnect to the board and repeat its configuration. As the current board firmware has no timeouts implemented, this button could be the only way to recover a misbehaved board.
The blue **halt** button generates a halt signal for the board firmware. Wave measurements usually take some time because several samples are read in sequence. If you want to stop the board operation you can use the **halt** button to end the current board action.

Firmware file for this board has the name SLAB-Nucleo64-F303RE-X.Y.bin where X and Y are the major and minor numbers of the version code.

The ADC readings on this board are not fully in sync with the DAC generated values so it is advisable to calibrate the board to obtain good readings.

Recommended setup for the board

In order to prevent errors due to the current injected in the ADCs and the output resistance of the DACs, a buffered setup both for inputs and outputs is recommended. The following figure shows a setup that uses one MCP6004 quad opamp for the four ADCs and one MCP6002 dual opamp for the two DACs.



The schematic also includes a red LED so that you can see it light when the board is powered.

Observe that in the second DAC channel connected to pin D13 a gain 2 non inverting amplifier configuration is used. As we remember, pin D13 is connected to the on board user LED. We need to keep voltages at this pin below half Vdd to prevent the LED to affect the DAC response. Alternatively you can disconnect the LED on the board by removing the proper solder bridge and use a normal follower circuit. This is explained in depth in the *"SLab F303 Shield A Reference"* document.

If you perform the proper calibration on the circuit, the obtained calibration tables will take care of the amplification at DAC 2 so you will not need to take this channel gain into account when performing measurements.

Bandwidth for buffer in DAC 2 is 500kHz and for all other buffers is 1MHz. As current firmware for this board is limited to a minimum 25µs sample time, the bandwidth is enough for the Slab system capabilities.

The use of the MCP600x opamps in the buffers impose the following limits:

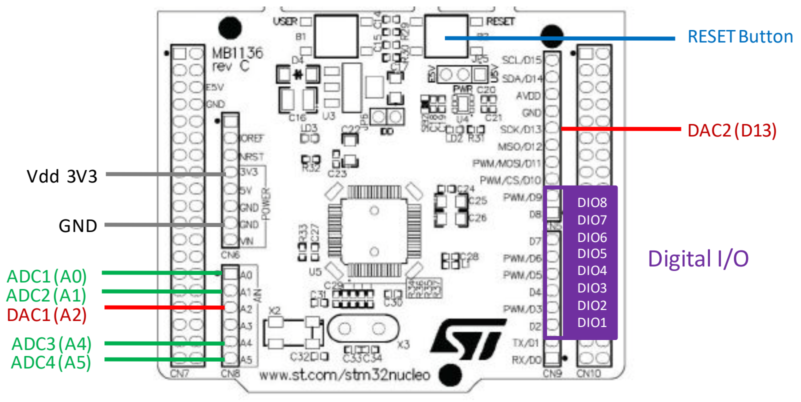
- Output swing of the opamps fall short 25 mV of the supply rails. So the system won't be able to guarantee we can force or read voltages at less than 25 mV from each supply.

- Output current of the opamps is limited to about 15mA. So you should check that you don't require more than this current from any of the DAC channels. It is recommended to set the limit to 10 mA to have some current margin.

Most of the SLab development is done using this board, so this is the recommended one.

STM32 Nucleo64 L152RE

This board features a STM32L152RE microcontroller in a 64 pin package. You can get more information about this board on the STMicroelectronics and the MBED sites.
The board includes connections to all I/O pins of the included microcontroller. Current version 1.x firmware, however, only uses eight analog pins and eight digital I/O pins.



The following table shows the analog pins we will use to interface the test circuit.

Pin	SLab Name	Kind	Functionality
GND	GND	Supply	Common reference for all voltages
3V3	Vdd		3,3V Constant supply voltage
A0	ADC1	ADC	Analog input: Can read analog voltage of connected node Read voltages must be bounded to the GND to Vdd range between 0.0V and 3.3V
A1	ADC2		
A4	ADC3		
A5	ADC4		
A2	DAC1	DAC	Analog output: Can force analog voltage of connected nodes D13 range is limited to 0.0V to 1.5V. A2 can go from 0.0V to 3.3V
D13	DAC2		

Aside from the analog pins we have eight digital I/O pins from DIO1 to DIO8 that correspond with pins D2 to D9 on the board.

The board also includes a black **reset** button and a blue user button that is used for the **halt** signal.
The black **reset** button interrupts all communication and is nearly equivalent to unplug and plug again the board from the computer. That usually means that you need to reconnect to the board and repeat its configuration. As the current board firmware has no timeouts implemented, this button could be the only way to recover a misbehaved board.
The blue **halt** button generates a halt signal for the board firmware. Wave measurements usually take some time because several samples are read in sequence. If you want to stop the board operation you can use the **halt** button to end the current board action.

Firmware file for this board has the name SLAB-Nucleo64-L152RE-X.Y.bin where X and Y are the major and minor numbers of the version code.
Current firmware for this board is limited to a minimum 50µs sample time. This is worse than the F303RE board, but, as the L152RE works at lower power levels, it is expected to obtain lower noise on the measurements. But this has not been tested.

Recommended setup for the board

This board has the same output configuration as the previous F303RE board, so the same circuit with a MCP6004 and a MCP6002 can be used. As the maximum sample frequency is lower on the L152RE board than in the F303RE board, the bandwidth of the amplifiers is enough.

This board is not fully tested during the SLab development, so expect more bugs than using the F303RE board.

First Steps

Once you have the board with the proper firmware loaded, you can now check that the connection is operational. Open Python in interactive mode and write the command:

```
>>> import slab
```

The ">>>" simbol in green text means that we need to enter this text in the Python interpreter. You don't need to write the ">>>" characters. If you write the command inside a Python script you also don't need to write the ">>>" characters.
Python should respond something like that to the previous command:

```
SLAB Module
Version 1.0 (22/2/2017)
Running interactively
```

If it gives any error, check that the slab.py file is available in the Python module search path. Test responded by the slab.py module will be written in green without ">>>" leading characters.

Now, we want to connect with the board.

The slab.py module interacts with the board firmware using a serial COM port at 38400 baud speed. For the connection to work, the drivers of the board shall be installed so it is detected by the system.

In the case of the nucleo boards, they connect with the PC using a composite ST-Link/V2-1 USB interface that includes a mass storage unit for firmware programming, a debug port and a serial com link. The mass storage interface is needed to download the firmware on the board and the serial interface is needed for the Slab module communication. The debug interface is not used in the Slab system. Refer to the nucleo documentation for details about installing the board drivers.

In order to obtain a successful connection you usually need to know the name of the port. In windows it is usually "COMn" where n is a number and you can usually request its number on Devices and Printers on the windows menu or in the device administration control panel. In other operating systems it can be different. You can refer to the Python pyserial documentation to obtain more details.

Once you have the port name of the hardware board, input the following command on the Python console, changing 'COM52' with whatever name your serial port has:

```
>>> slab.connect('COM52')
```

Python should respond with something like:

```
Getting board data
Connected to Nucleo64-F303RE SLab v1.0
No ADC calibration data found
No DAC calibration data found
```

If the port name is not a proper port it will respond something like the following two lines following with some more exception information:

```
** SLab exception
** Cannot open connection. Check port
```

If the port name exists on the system but it is not the board with the proper firmware loaded it will respond something like:

```
** Slab exception
** Board not responding. Check port and Firmware
```

Or like:

```
        ** Slab exception
        ** Bad magic from board. Check Firmware
```

In the first case there is not response from the com port after one second and in the second case the response from the board is wrong. The Slab board firmware contains a 4 byte magic code that guarantees that you only get a proper connection if you select a port that links to a board with the proper Slab firmware.

As you can see, the Python SLab module responds with an exception every time something goes wrong. This is typical way to handle errors in Python.

If you are not able not obtain the proper name of the COM port used by the board, you can request the Slab module to autodetect the port. In order to do that, just request the connection without providing any port name:

```
        >>>slab.connect()
```

That should usually work on Windows and Linux. If the board can be detected you will see something like:

```
        Board detected at port COM52
        Getting board data
        Connected to Nucleo64-F303RE SLab v1.0
        No ADC calibration data found
        No DAC calibration data found
```

You can take note of the COM port nave for successive connections but the Python SLab module will also remember this port for you. The last successfully COM port opened is stored in a “Last_COM.dat” file, so next time the board tries to autodetect the port, it will first try to use the last valid COM port like in the next example:

```
        >>>slab.connect()

        Trying last valid port COM52
        Board detected
        Getting board data
        Connected to Nucleo64-F303RE SLab v1.0
        No ADC calibration data found
        No DAC calibration data found
```

Using auto detect is always slower than providing the proper port name but thanks to the “Last_COM.dat” file the full autodetect algorithm will only need to be carried out once. Moreover, the “Last_COM.dat” file is also stored for manually selected ports, so, except for the first time, you can always use the ***connect*** command without arguments.

The auto detect algorithm works by requesting a magic code from all connected devices. As this is an active search, it can affect the normal operation of the devices in the system. If you detect any problem, refrain from using auto detect and manually select the port at least until the "Last_COM.dat" file is generated.

If a board cannot be detected automatically you will get an exception:

```
** SLab exception
** COM Autodetect Fail
```

If the connection is established, you can disconnect from the board afterwards with the command:

```
>>> slab.disconnect()
```

Python will respond with something like:

```
Disconnected from the Board
```

As you have seen, and usual in python, we have used the slab module name for all the commands. All commands will have the structure:

```
var = slab.command_name(Arguments)
```

Where *command_name* is the name of the command and *Arguments* is an optional argument list for the command. Some commands return values. In those cases you can assign the result to a variable *var* or, in some cases, a list of variables. For commands that do not return any value, or for the cases that the command returns data you don't need, you can omit the "*var* =" part of the call. As this is standard in Python refer to the Python literature for more information about this language.

It is not recommended to remove the slab reference by using:

```
>>> from slab import *
>>> connect()
>>> disconnect()
```

It is bad Python practice and can break the operation of the program in unexpected ways. If you want to reduce the number of characters you are writing you'd better use an alias name for the module:

```
>>> import slab as sl
>>> sl.connect()
>>> sl.disconnect()
```

At any time you can ask SLab for help using the ***help*** command:

```
>>> slab.help()
```

If you know a topic, like a command name for instance, you can directly ask for it. In the case of the ***connect*** command you can write:

```
        >>> slab.help("connect")
```

That ends the introduction and now, all available commands you can type in the Python interpreter will be described grouped by functionality. The commands can be sent from an interactive console or as part of a Python script.

For each command, the minimum major version number where it is available is indicated together with its description and, usually, one usage example.

Most command will include an operation example. Examples associated to an example number, **nn** are provided as an ***example_nn.py*** python file ready to be executed.
In all examples it is supposed that you have previously imported the slab module and connected to the board using, for instance:

```
        >>> import slab
        >>> slab.connect()
```

Before leaving Python, both in interactive mode or using a script, it is recommended to disconnect from the board issuing:

```
        >>> slab.disconnect()
```

The different slab commands change the Hardware Board state. You can restore the board to the original state after power-up by executing a **hard reset** or a **soft reset**.

A **hard reset** is performed if you push the reset button on the Hardware Board. After a hard reset you need to issue a **connect** command to reconnect with the board.
A soft reset is performed by issuing the ***softReset*** command. This command sets the board state to the power-up state without needing a hard reset.
In general a soft reset is enough unless the board is not responsive.

The Slab system has a natural tendency to feature creeping. In order to make the dimension of the project easy to manage, it has been divided in several modules.
The main module, associated to the file slab.py, contains all the code that directly accesses the Hardware Board and the most immediate low complexity commands.
The functionality of the main module is extended by the use of additional modules. This document describes this main module, associated to the file slab_dc.py.

You can find in the same folder of this reference the reference for other slab sub modules. The current list of SLab modules is:

Module	**Description**
slab.py	Main SLab module that directly interacts with the Nardware Board
slab_dc.py	Module with commands to draw DC curves
slab_ac.py	Module that deals with the frequency response of circuits
slab_meas.py	Module that provides complex measurements on transient measurements
slab_fft.py	Module that deals with measurements related to the fast fourier transform
slab_ez.py	Module that eases the operations of the SLab system

The help system is unified for all the modules, so the ***help*** command provides help information for all the modules.

SLab commands introduction

The SLab system is a client-server solution. The hardware board acts as a server waiting for command requests from Python in interactive or script mode. Those commands that the hardware board can understand are named **base commands**. Each base command is assigned to a letter for easy identification. For instance, the soft reset base command is identified by the letter "E".

We don't directly issue **base commands** to the hardware board. We use the Python **SLab commands** for that. The SLab commands build upon the base commands. Any SLab command that does not directly issue a base command either does not communicate with the board or it indirectly issues a base command by calling another Slab command.

Slab commands will be described in this reference using always the same pattern. First line will include the name of the command and call parameters. Next a table of parameters and results is indicated. Below the table, at the right side, there could be an optional remark. The following table shows an example for command *setTransientStorage*.

setTransientStorage(samples,na)

Arguments	samples	Number of samples to measure	V1
	na	Number of analog channels to read (Defaults to 1)	
Returns		Nothing	

Base "S"

First line shows the command name **setTransientStorage** and its two parameters **samples** and **na**. The table that follows shows a description of the arguments and return values for the command.
Some commands feature parameters, like **na**, that are optional. They will be shown in blue color.
At the right of the table **V1** indicates that this command is implemented from major version 1.
Below the table, to the right, it shows a remark **Base "S"** that indicates that it issues a base command to the hardware board. In particular, letter "S" base command.

Possible remarks for a command are:

- **Base** : Indicates that it directly interacts with the hardware board by issuing the indicated letter base command. In some special cases more than one letter is indicated. That means that the Slab command issues several base commands.

- **Auxiliary** : Indicates that the command does not directly or indirectly communicate with the hardware board. That means that you don't need to connect to the board to use this command.

After the table, a detailed description of the command follows. Finally, except for very simple commands, an example is provided. If the example is numbered, that means that the Python script for the example is provided as an **"Example_nn.py"** file where **nn** is the example number. If a letter follows the number, as in "Example_22A", that means that the related Python script file contains more than one example.

Python is case sensitive so the commands shall be written with the proper upper and lower case letters. Most commands start in lower case and use upper case to separate words. That is a standard way to write identifiers known as lower camel case.

If a command cannot be completed or there is any error, an exception will be generated and a message will be shown, where *Message* will explain the cause of the exception:

```
** SLab exception
** Message
```

If the command is run inside a script, the run will be paused, requesting to hit the return key, so that you can see the message before the window closes.

Some commands can generate warnings that are not fatal. In those cases a message will be generated, where *Message* will explain the warning:

```
** WARNING
** Message
```

If the verbose level allows it, and if there have been any warning since connecting to the board, it will show up when the close command is executed.

Management Commands

Those commands manage the board as a whole or are a shortcut for functions from other Python modules.

help(*topic*)

Arguments	topic	Requested topic (Defaults to "root")	V1
Returns		Nothing	

Auxiliary

Gives help information about one topic. If no topic is provided, goes to the main help page.

setVerbose(*level*)

Arguments	level	Level number 0, 1, 2 or 3	V1
Returns		Previous verbose level	

Auxiliary

This command sets the verbose level of all other commands.
Some commands never write to screen while others can write more or less information depending on the verbose level. Valid values for this level are:

0	Nothing will be written on screen
1	Only warnings will be shown
2	Warnings and basic messages will be written
3	More detailed information will be written

The command returns the previous verbose level.

By default, verbose level is 2. In interactive mode it is recommended to use verbose levels of 2 or 3 to get information on command operations. When calling the commands from a script it could be interesting to set verbose level to 1 to eliminate all command messages.

The verbose level does not affect exception messages or explicit print requests.

Example:
Eliminate all messages generated by commands

```
>>> slab.setVerbose(0)
```

connect(*portName*)

Arguments	portName	String with a valid com port name Or nothing for autodetect	V1
Returns		Nothing	

Base "M" "F" "I" "L"

Connects with a Hardware Board at port ***portName***.
If no connection can be established generates an exception.
Gives a message after the connection is performed.
If the ***portName*** is omitted, it tries to autodetect the port and gives its value if found.
Autodetect works by exploring all possible COM ports, and sending a magic code request using the char 'M' until a proper response if obtained. That means that only the first board detected is found. Moreover, as the search for the port is active (sends **"M"** to all available ports) it can disrupt the operation of other devices connected to the PC.
Last valid COM port is stored in a **"Last_COM.dat"** file, so auto detect always first tries the last valid COM port.
If the board is already connected, disconnects from it and reconnects. This procedure is useful if you reset the board and want to restart the connection.

The connect command is special because it issues four base commands to the board. Command "M" to check the magic and verify that there is a board with SLab firmware on one COM port, "F" to get the board name, "I" to obtain board capabilities information and finally "L" to obtain the pin list for the board.

Hardware board state is not reset on connection or disconnection. If messages are enabled, the command will indicate if the board is at reset state. If it is not and you want to reset the board state, you can perform a **hard reset** or a **soft reset**.

Example:
Connection to a board at a known port

```
>>> slab.connect('COM52')
```

Example:
Connection to a board at an unknown port

```
>>> slab.connect()
```

That will respond with something like that the first time:

```
Board detected at port COM52
Getting board data
Connected to Nucleo64-F303RE SLab v1.0
No ADC calibration data found
No DAC calibration data found
```

If there had been a previous successful connection it will respond something like:

```
Trying last valid port COM52
Board detected
Getting board data
Connected to Nucleo64-F303RE SLab v1.0
No ADC calibration data found
No DAC calibration data found
```

disconnect()

Arguments	None	V1
Returns	Nothing	

Disconnects from a previously connected hardware board.
If no board was connected generates an exception.
Gives a message after the board is disconnected.

If any warning has been generated since the last connection and the verbose level is not zero, the number of warnings will be shown.

This command does not communicate with the board, only closes the serial communication. It is not marked as **Auxiliary** because you have to be connected to the board to use it.

Example:
Connection and disconnection from the board using automatic port identification

```
>>> slab.connect()
>>> slab.disconnect()
```

softReset()

Arguments	None	V1
Returns	Nothing	

Performs a **soft reset** on the Hardware Board.
It sets the state of the hardware board to the power-up state reset condition.
This command is useful if you want to guarantee that the following commands don't depend on previous commands sent to the board.
A soft reset conserves the board connection so you don't need to reconnect.
If the board is not responsive to a **soft reset**, a **hard reset** can be generated using the reset button of the board. A hard reset requires to reconnect with the board.

Example:
Perform a soft reset

```
>>> slab.softReset()
```

printBoardInfo()

Arguments	None	V1
Returns	Nothing	

Prints information about the board on screen.
All the information is obtained by the SLab Python module from the board itself, so the SLab module does not need to be rewritten to support new boards.

Example 01:
Get information about the connected board

```
>>> slab.printBoardInfo()
```

The command will print something like:

```
Board : Nucleo64-F303RE SLab v1.0
  COM port : COM3
  Reference Vref voltage : 3.324 V
  Power Vdd voltage : 3.328 V
  2 DACs with 12 bits
  4 ADCs with 12 bits
  8 Digital I/O lines
  DAC Pins ['A2', 'D13']
  ADC Pins ['A0', 'A1', 'A4', 'A5']
  DIO Pins ['D2', 'D3', 'D4', 'D5', 'D6', 'D7', 'D8', 'D9']
  Buffer Size : 20000 samples
  Maximum Sample Period : 100.0 s
  Minimum Sample Period for Transient Async: 2.5e-05 s
  Maximum Sample Frequency for Frequency Response : 38000.0 Hz
```

Pin names are given in order, so, on the previous example, A2 is DAC 1, D13 is DAC 2 and ADC 1,2,3,4 correspond to pin names A0, A2, A4 and A5.
As this is an explicit print request it is not affected by verbose level.

wait(time)

Arguments	time	Float time value (in seconds)	V1
Returns		Nothing	

Auxiliary

Stops the program during the selected time. This command is useful to let the outputs of the system stabilize after a change in the inputs.
As the command doesn't use the Hardware Board, you don't need to connect to use this command.

Example:
Wait 2.5 seconds

```
>>> slab.wait(2.5)
```

pause(message)

Arguments	message	Message to show (Gives default message if not provided)	V1
Returns		Nothing	

Auxiliary

Writes a message. If no message is provided it will write:

"Script Paused. Hit RETURN to continue"

Then pauses the script until the **return key** is hit.
As the command doesn't use the Hardware Board, you don't need to connect to use this command.

Example:
Request a pause with default message

```
>>> slab.pause()
```

setPlotReturnData(value)

Arguments	value	Value to set True or False (Default is False)	V1
Returns		Nothing	

Auxiliary

Plot commands that will be described later, by default don't return the data plotted. Most of those commands feature a ***returnData*** parameter that enables them to return the plot data. An equivalent effect can be obtained with the ***plotReturnData*** internal variable that is defined by this command and set to **False** by default. By setting this variable to True you can make that all plot commands that feature a ***returnData*** parameter, return the data they use to generate the plot without needing to make True its own ***returnData*** parameter.

Example
Post process data from a transient plot

We make a transient async plot and, afterwards, we get the mean value of the data points

```
>>> slab.setTransientStorage(100,1)
>>> slab.setPlotReturnData(True)
>>> t,a1 = slab.tranAsyncPlot()
>>> slab.mean(a1)
```

The following code will be equivalent:

```
>>> slab.setTransientStorage(100,1)
>>> t,a1 = slab.tranAsyncPlot(returnData=True)
>>> slab.mean(a1)
```

File Commands

Commands in this section affect the interaction of SLab with the filesystem.

save (*filename, data*)

Arguments	filename	Name of the file to write (without extension)	V1
	data	Variable to store	
Returns		Nothing	

Auxiliary

This command saves a variable on a file using pickle. If you want to save several variables you can put all of them in a list or a dictionary.
An extension ".sav" is added to the filename.

Example:
Save two variables on a file with name "data.sav"

```
>>> a = "First variable"
>>> b = [1,2,3,4]
>>> slab.save("data",[a,b])
```

load (*filename*)

Arguments	filename	Name of the file to write (without extension)	V1
Returns		Variable contained on the file	

Auxiliary

This command loads a variable from a file using pickle.
An extension ".sav" is added to the filename.

Example:
Recover and show two variables from a file "data.sav"

```
>>> a,b = slab.load("data")
>>> a
>>> b
```

setFilePrefix (*prefix*)

Arguments	prefix	Prefix for standard files (Defaults to no prefix)	V1
Returns		Nothing	

Auxiliary

This command sets a prefix for all standard data (.dat) files SLab reads or writes, like the calibration files or the last COM file.
If no prefix is give, the prefix is removed.
Files affected: "Cal_Vdd.dat", "Cal_ADC.dat", "Cal_DAC.dat" and "Last_COM.dat".
If used, it is recommended to set the prefix before calling the ***connect*** command as connect tries to read several data files.

Example:
Connect using data files from the parent folder:

```
>>> slab.setFilePrefix("../")
>>> slab.connect()
```

setCalPrefix (*prefix*)

Arguments	prefix	Prefix for calibration files (Defaults to no prefix)	V1
Returns		Nothing	

Auxiliary

This command sets a prefix for all calibration data (.dat) files SLab reads or writes.
If no prefix is given, the prefix is removed.
Files affected: "Cal_Vdd.dat", "Cal_ADC.dat" and "Cal_DAC.dat".
By changing the prefix you can use different calibration files for different boards.
If used, it is recommended to set the prefix before calling the ***connect*** command as connect tries to read several data files.
The calibration prefix is added after the global file prefix set by the command ***setFilePrefix***.

Example:
Connect to a different board and use calibrations different to default ones:

```
>>> slab.setCalPrefix(“Board_B_”)
>>> slab.connect()
```

Basic Voltage DC Commands

This section includes the basic commands that can set or measure voltages in DC. All those commands operate on voltages typically from 0.0V to 3.3V for nucleo boards.

setVoltage(channel,value)

Arguments	channel	DAC to set 1, 2 or 3*	V1
	value	Voltage between 0.0 and Vdd	
Returns		Nothing	

Set one DAC output of the board to the selected voltage.
*All SLab compatible boards feature at least 2 DACs, some could feature more DACs.

In some boards DAC2 output is at D13 and this pin is connected to the user LED, it cannot force voltages over half Vdd, so voltage values for this DAC are only guaranteed below about 1.5V. The calibration of the board overcomes this potential problem.

Example:
Set voltage at DAC 2 to 1V

```
>>> slab.setVoltage(2,1.0)
```

readVoltage(ch1,ch2)

Arguments	Ch1	ADC to read as (+) 1, 2, 3, 4 or 0 (GND)	V1
	Ch2	ADC to read as (-) 1, 2, 3, 4 or 0 (GND) (Defaults to GND)	
Returns		Voltage read on the ADC	

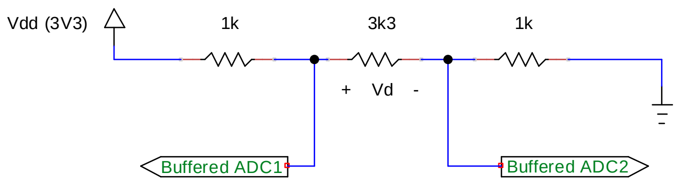
Reads the voltage of one of the four ADC inputs ADC1, ADC2, ADC3 and ADC4 expressed in volts. If two parameters are given, reads the voltage difference between the channels. Channel number 0 is considered to be GND.

Example:
Show the voltage at ADC 4

```
>>> a4 = slab.readVoltage(4)
>>> print "Voltage at ADC4 is "+str(a4)+ " Volt"
```

Example 24A:
Measure voltage in the intermediate resistor of a chain of three resistors.

We will use the following circuit:



In order to measure the voltage at the 3,3kΩ resistor we will issue:

```
>>> vd = slab.readVoltage(1,2)
>>> print "Voltage at 3k3 resistor is " + str(vd) + " V"
```

Voltage reading should be:

$$V_d = V_{dd}\frac{3{,}3k\Omega}{1k\Omega + 3{,}3k\Omega + 1k\Omega}$$

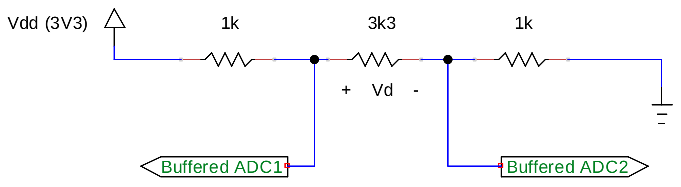
rCurrent(rvalue,ch1,chl2)

Arguments	rvalue	Resistor value (Ω)	V1
	ch1	ADC to read as (+) or 0 (GND)	
	ch2	ADC to read as (-) or 0 (GND) Defaults to 0 (GND)	
Returns		Voltage between (+) and (-)	

Reads the differential voltage between the two nodes of a resistor and computes the current from the result. The (+) and (-) nodes can be 1 to 4 for any of the ADC inputs or 0 for ground (GND). If no ch2 is provided, (-) node will be considered to be ground (GND).
If rvalue is given in Ω, result will be in A. If it is given in k Ω, result will be in mA.

Example 24B:
Measure current in the intermediate resistor of a chain of three resistors.

We will use the same circuit as in example 24A:



In order to measure the current at the 3,3kΩ resistor we will issue:

```
>>> i = slab.rCurrent(3.3,1,2)
>>> print "Current at 3k3 resistor is " + str(i) + " mA"
```

And it shall be the same as in any other resistor

```
>>> i1 = slab.rCurrent(1.0,2,0)
>>> print "Current at grounded 1k resistor is " + str(i) + " mA"
```

Current reading should be:

$$i = \frac{V_{dd}}{1k\Omega + 3{,}3k\Omega + 1k\Omega}$$

dcPrint()

Arguments	None	V1
Returns	Nothing	

Reads the voltage of the four ADC inputs ADC1, ADC2, ADC3 and ADC4 expressed in volts and show the results on screen.
As this is an explicit print request it is not affected by verbose level.

Example 02:
Show the voltage at all four ADCs

```
>>> slab.dcPrint()
```

The ***dcPrint*** command will show on screen something like:

```
ADC DC Values
  ADC1 = 0.533 V
  ADC2 = 0.131 V
  ADC3 = 0.237 V
  ADC4 = 0.384 V
```

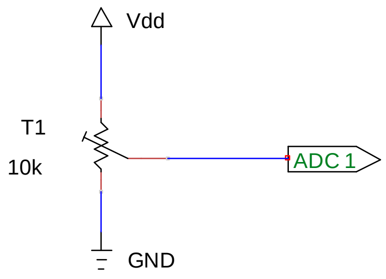
dcLive(n, wt, single)

Arguments	n	Number of last ADC to show	V1
	wt	Wait time between measurements (Defaults to 0.2s)	
	single	Only show the ADC number n	
	returnData	Return obtained data (Defaults to False)	
Returns		Noting	

This command shows the live values of the selected ADCs.
It writes on screen the values of ADCs 1 to ***n***, waits ***wt*** seconds and repeat the readings.
If ***wt*** is not provided, it defaults to a 0.2 second wait.
If the optional parameter ***single*** is set to True, only ADC number ***n*** will be shown.
In order to end the command execution you need to interrupt it with CTRL+C.
If the optional parameter **returnData** is True, all the measurements will be returned in a list with one vector for each ADC channel.

Example 27:
Show the live change of a trimpot

In this example we will use a 10 kΩ trimpot and we will show the live value of the component output as measured by ADC 1.



Just run the code:

```
>>> slab.dcLive(1)
```

Move the trimpot cursor to see how the voltage change.

Use CTRL+C to end

Basic Ratiometric DC Commands

This section includes the basic commands that can set or measure ratiometric voltages in DC. A ratiometric voltage is a voltage expressed as a ratio respect to a reference voltage. The commands in this section use a Vref voltage as a reference so the values are inside a range from 0.0 (voltage set at GND) to 1.0 (voltage set at Vref).
In most boards the Vref voltage is equal or similar to the Vdd voltage.

Those commands are included to perform measurements that are intrinsically ratiometric and don't depend on knowing the real Vref value.
The board firmware works only on ratiometric values so all absolute voltage values are calculated from Vref and the ratiometric readings obtained from the board.

writeDAC(channel,value)

Arguments	channel	DAC to set 1, 2 or 3*	V1
	value	Ratiometric value between 0.0 and 1.0	
Returns		Nothing	

Base "D"

Set one DAC output to the selected ratiometric value.
*All SLab compatible boards feature at least 2 DACs, some feature a third DACs.

In some boards DAC2 output is connected to an user LED and it cannot force voltages over half Vdd, so, in that case, ratiometric values for this DAC are only guaranteed below 0.5.

Example:
Set DAC 1 voltage halfway between Vdd and GND

```
>>> slab.writeDAC(1,0.5)
```

readADC(channel)

Arguments	channel	ADC to read 1, 2, 3 or 4	V1
Returns		Ratiometric value between 0.0 and 1.0	

Base "A"

Reads the voltage of one of the four ADC inputs ADC1, ADC2, ADC3 and ADC4 expressed as a ratiometric value (0.0 for GND and 1.0 for Vref).

Example:
Read the ratiometric value of ADC 1 respect to Vref

```
>>> a1 = slab.readADC(1)
>>> print "Ratiometric voltage at ADC1 is "+str(a0)
```

Global DC Commands

Commands in this section affect the general operation of both ratiometric and voltage DC measurements.

setDCreadings(number)

Arguments	number	Number of readings (1 or more)	V1
Returns		Nothing	

Base "N"

DC readings are obtained averaging several ADC measurements. By default the hardware board will discard the first reading and average the following 10 measurements to provide a DC reading. This function can be used to change the number or readings that are averaged.
Using a number over 10 takes more time but reduces the measurement noise. Using a number below 10 reduces measurement time but increases noise.
The number of readings is set by an unsigned 16 bit number so its maximum is 65535.
For Gaussian noise, the noise reduction is about the square root of the number of averaged measurements.

Example:
Obtain a low noise measurement on ADC 3

```
>>> slab.setDCreadings(100)    # Reduce noise to about 1/10
>>> a3 = slab.readVoltage3()
>>> print "Voltage at ADC3 is "+str(a3)+ " Volt"
```

zero()

Arguments	None	V1
Returns	Nothing	

Set all DAC channels to zero.
It is recommended to only modify the circuit connections when it is not powered. That means that you shall remove the Vdd lead before performing circuit modifications. The problem is that if DACs are set to a non zero value, you could power the circuit through the DACs. That could yield important problems because active circuits don't usually like to have input voltages when they are not powered.
The ***zero*** command set all DACs to its minimum value so that you can remove the power supply without producing any hazard on the active devices on the circuit.
The ***zero*** command can also be used as a shortcut to set all DACs to zero at the same time. Note, however, that this command doesn't use the DAC calibration tables, so you will get zero at the DAC outputs of the board. This is not always the same value at the buffer outputs.

Example:
Perform a secure modification of the circuit
First we issue the zero command

```
>>> slab.zero()
```

Then we can remove the Vdd lead that powers the circuit and the ADCs and DACs buffers.

Generic Plot Commands

The generic plot commands allows you to plot equal size vectors, in X-Y plots. Those plots can be useful to represent the data obtained in the measurements.
All commands in this section are **Auxiliary** as they don't communicate with the hardware board.

You can always refer to Matplotlib to generate plots, but the ***plot11***, ***plot1n*** and ***plotnn*** commands are much easier to use if they are enough for your needs and also you don’t need to import this module.

As those plot commands, not measurement commands, no connection to the board is needed to use them.

plot11(x, y, title, xt, yt, logx, logy)

Arguments	x	X vector	V1
	y	Y vector	
	title	Title for the plot (Defaults to no title)	
	xt	Text for X axis (Defaults to no text)	
	yt	Text for Y axis (Defaults to no text)	
	logx	Use logarithmic X axis (Defaults to False)	
	logy	Use logarithmic X axis (Defaults to False)	
Returns		Nothing	

Auxiliary

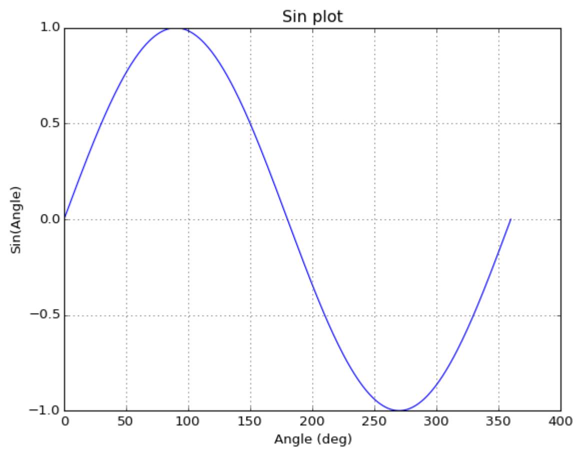
This is a basic plot command (one to one) that plots the values on a **y** vector against the values on a **x** vector.
If **x** is an empy list [], a sequence starting from 0 and incrementing on 1 for each point will be used for the X axis.
This command is useful for plotting results obtained when post processing measurement data using Python scripts.

Example 13A:
Draw a sin(x) plot

In order to draw this plot we need support from the numpy library.

```
>>> import numpy as np
>>> import slab
>>> x = np.arange(0,361,1)
>>> y = np.sin(np.pi*x/180)
>>> slab.plot11(x,y,"Sin plot","Angle (deg)","Sin(Angle)")
```

That will give us the following figure:



plot1n(x, ylist, title, xl, yl, labels, location, logx, logy)

Arguments	x	X vector	V1
	ylist	List of Y vectors	
	title	Title for the plot (Defaults to no title)	
	xt	Text for X axis (Defaults to no text)	
	yt	Text for Y axis (Defaults to no text)	
	labels	List of labels (Defaults to no labels)	
	location	Location of the label legend (Defaults to 'best')	
	logx	Use logarithmic X axis (Defaults to False)	
	logy	Use logarithmic X axis (Defaults to False)	
Returns		Nothing	

Auxiliary

This is a plot command (one to n) that plots several **y** vectors against the values on a **x** vector.
If **x** is an empty list [], a sequence starting from 0 and incrementing on 1 for each point will be used for the X axis.
A list of **y** vectors is provided as the second argument.
A list of labels for the curves can be optionally provided together with a location that, by default, is selected as the automatic 'best' option.

Valid locations for the legend are: ‘best’ , ‘upper right’ , ‘upper left’ , ‘lower left’ , ‘lower right’ , ‘right’ , ‘center left’ , ‘center right’ , ‘lower center’ , ‘upper center’ and ‘center’. Refer to the Matplotlib documentation for more information.

This command is useful for plotting most results obtained when post processing measurement data using Python scripts. The only limitation is to share the same set of x values for all curves.

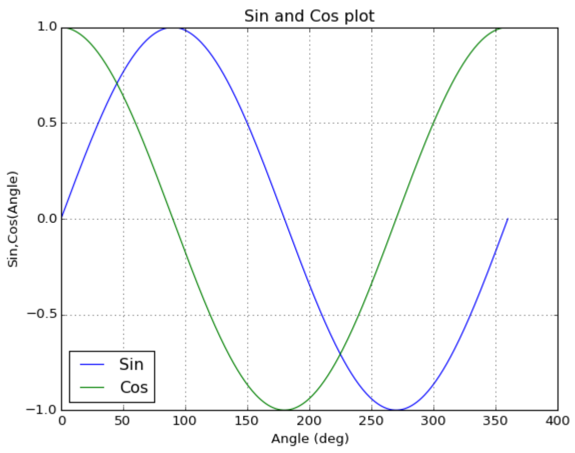
Example 13B:

Draw a sin(x), cos(x) plot

In order to draw this plot we need support from the numpy library.

```
>>> import numpy as np
>>> import slab
>>> x = np.arange(0,361,1)
>>> y1 = np.sin(np.pi*x/180)
>>> y2 = np.cos(np.pi*x/180)
>>> slab.plot1n(x,[y1,y2],"Sin and Cos plot",\
..."Angle (deg)","Sin,Cos(Angle)",["Sin","Cos"])
```

That will give us the following figure:



plotnn(xlist, ylist, title, xl, yl, labels, location, logx, logy)

Arguments	xlist	List of X vectors	V1
	ylist	List of Y vectors	
	title	Title for the plot (Defaults to no title)	
	xt	Text for X axis (Defaults to no text)	
	yt	Text for Y axis (Defaults to no text)	
	labels	List of labels (Defaults to no labels)	
	location	Location of the label legend (Defaults to 'best')	
	logx	Use logarithmic X axis (Defaults to False)	
	logy	Use logarithmic X axis (Defaults to False)	
Returns		Nothing	

Auxiliary

This is a plot command (n to n) that plots several **y** vectors against the values of several **x** vectors.
A list of **x** vectors is provided as the first argument and a list of **y** vectors is provided as the second argument.
A list of labels for the curves can be optionally provided together with a location that, by default, is selected as the automatic ‘best’ option.
Valid locations for the legend are: ‘best’ , ‘upper right’ , ‘upper left’ , ‘lower left’ , ‘lower right’ , ‘right’ , ‘center left’ , ‘center right’ , ‘lower center’ , ‘upper center’ and ‘center’. Refer to the Matplotlib documentation for more information.

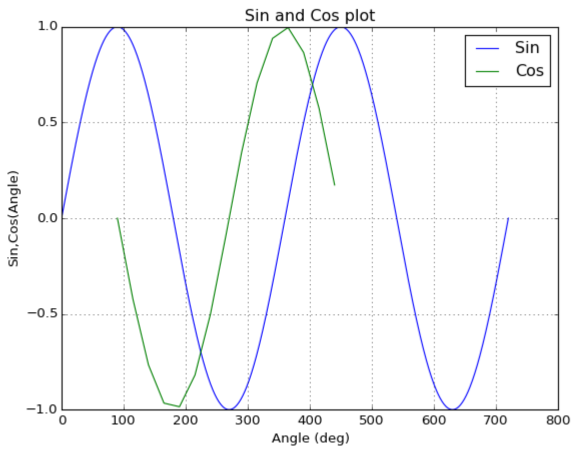
This command is similar to the ***plot1n*** command, but it doesn’t need to share the same x values in all curves.

Example 13C:
Draw a sin(x), cos(x) plot with different ranges and resolutions

In order to draw this plot we need support from the numpy library.

```
>>> import numpy as np
>>> import slab
>>> x1 = np.arange(0,721,1)
>>> y1 = np.sin(np.pi*x1/180)
>>> x2 = np.arange(90,451,25)
>>> y2 = np.cos(np.pi*x2/180)
>>> slab.plotnn([x1,x2],[y1,y2],"Sin and Cos plot","Angle (deg)","Sin,Cos(Angle)",["Sin","Cos"])
```

That will give us the following figure:



Sweep DC Commands

The automation of measures that provides the SLab library enables an easy processing of measured data. The sweep DC commands enable you need to make DC measurements for several values of the voltage generated on a DAC.

Commands in this section perform measurements and provide data vectors that can be used to post process the measurements. In fact, most commands on the DC sub module rely on those commands.

Vectors are provided as NumPy arrays. So they can be processed using normal operators and the ones provided in the NumPy library.

The final processed data can be represented using the plot commands of the previous section or by the use of the Matplotlib functions.

dcSweep(ndac, v1, v2, vi, wt)

Arguments	ndac	DAC number	V1
	v1	Start voltage	
	v2	End Voltage	
	vi	Voltage Step (Defaults to 0.1V)	
	wt	Wait time (Defaults to 0.1s)	
Returns		List of 5 vectors: x, y1, y2, y3, y4	

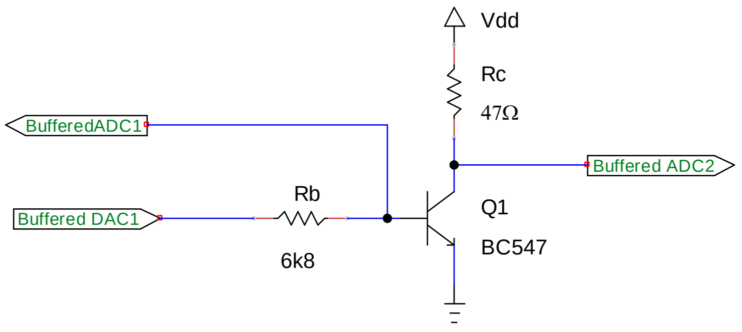
This command obtains the voltages measured by ADCs 1, 2, 3 and 4 for a set of voltages set at the indicated DAC.

DAC number is a number from 1 to the maximum of available DACs.

Measurements start with DAC at **v1** voltage and end at **v2** voltage. Voltage is increased in a **vi** value between one measurement and the next one. If **vi** is omitted, it defaults to 0.1 V.
Last voltage used will be below **v2**.
Between the change of the DAC voltage and the measurement with the ADCs the module waits **wt** seconds. If **wt** is omitted it defaults to 0.1 s.

Example 14:
Measure the β current gain as function of collector current for high currents on a BC547 bipolar junction transistor (BJT) using a STM32 Nucleo F303RE board

We will setup the following circuit:

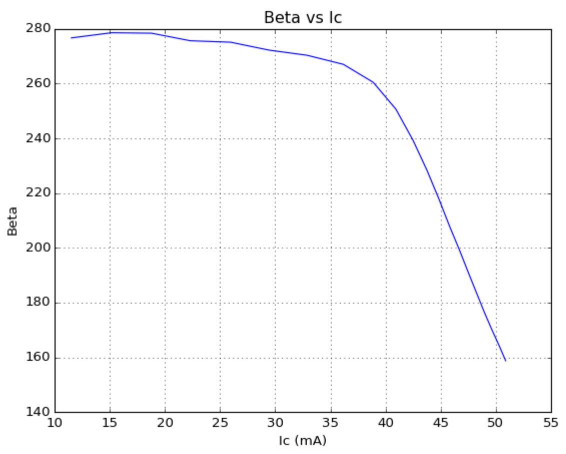


A 47Ω Rc resistor limits the collector current to 70mA. That is below the 100mA maximum rating of the transistor.

In order to obtain the gain we issue the following commands. First we increase the number of measurements to average to increase the accuracy. Then we perform a dc sweep on DAC 1 setting the DAC to zero at the end. After that, we calculate beta. Finally we draw the plot.

```
>>> slab.setDCreadings(50)
>>> x,y1,y2,y3,y4 = slab.dcSweep(1,1.0,3.0,0.1)
>>> slab.zero()
>>> Ib = (x - y1)/6.8       # In mA
>>> Ic = (3.3 - y2)/0.047   # In mA
>>> beta = Ic/Ib
>>> slab.plot11(Ic,beta,"Beta vs Ic","Ic (mA)","Beta")
```

The resulting plot is shown in the figure below. We can see that β degrades above 35mA.



Observe that maximum current is 50mA. As maximum current on the BC547 transistor is 100mA, that could be considered high current for this device. Collector resistance is 47Ω so at 50mA we get a 2,34V drop on Rc and almost 1V V_{CE} voltage so we know we are out of the saturation region.

dcSweepPlot(ndac, v1, v2, vi, na, wt, returnData)

Arguments	ndac	DAC number	V1
	v1	Start voltage	
	v2	End Voltage	
	vi	Voltage Step (Defaults to 0.1V)	
	na	Number of analog channels to show (Default to 4)	
	wt	Wait time (Defaults to 0.1s)	
	returnDate	Enable return of plot data (Defaults to False)	
Returns		See text. Generates a plot.	

This command is similar to the previous ***dcSweep*** command, but, after the measurements, the data is plotted.
Using the optional parameter **na** you can select the number of ADC channels to show. By default this parameter is set to 4 so all 4 ADC channels are shown.
If you set the optional ***returnData*** to True or you have previously called ***plotReturnData(True)***, the plotted data will be returned as vectors. The first vector will be the DAC voltage and the rest of vectors the selected ADC channels.

realtimePlot(nadc, wt, n, returnData)

	nadc	Number of ADCs to record (Defaults to 1)	
	wt	Wait time between readings (Defaults to 0.2 s)	
	n	Number of points to show (Defaults to All)	
	returnData	Return measurement data (Defaults to False)	
Returns		List of ndac + 1 vectors	

This shows the voltage of the selected ***nadc*** number of ADCs updating the graph every ***wt*** time. In order to end the measurement, close the graph window.

The command issues one DC measurement request to the hardware board for each measurement it provides. That means than only very low frequency measurements can be performed and that the channels measured can be misaligned in time.
Due to the use of DC requests, the measurement time is affected by the ***setDCreadings*** command.

This command is not designed to obtain timing measurements but to grasp the long time variation of the circuit voltages. Use the transient commands if you want to perform timing measurements.

After measurements end the final recorded data is shown in a static graph.

The optional parameter ***n*** select the number of points to show on the realtime graph. That way you can see better only the ***n*** last captured points.

The time between measurements include ***wt*** and the time to perform the measurements and show the graph. Due to that fact, the time axis, that uses real times, can be different from the expected from the ***wt*** value. This is especially true for low ***wt*** values.

If you set the optional ***returnData*** to True or you have previously called ***plotReturnData(True)***, the function returns the measured data as a list of nadc vectors:

- First vector is time
- Rest of vectors are ADC1, ADC2,... for the selected **nadc** value

Basic Transient Commands

The previous DC commands all obtain their data by performing individual measurements using the Hardware Board. Time between measurements cannot be precisely controlled and, as each measurement is sent to the PC trough the serial channel, time between measurements cannot be less than the time needed to send one measurement data.

Transient commands generate time dependent measurements using specific Hardware Board orders. Those orders store a sequence of precisely timed measurements on the board memory that are delivered to the PC when the measurement ends.

The ADCs of some of the supported boards can provide sample rates as high as 5 Mega samples per second. Current firmwares, however are not optimized enough to reach this limit. You can check the maximum sample rate for a connected board using the ***printBoardInfo( )*** command. Minimum time between samples can be as low as about 25µs. That gives a maximum sample rate up to 40kHz. Don't expect to obtain nothing near what a normal oscilloscope can provide. That means that the SLab system cannot be used for signals with frequencies much higher than normal audio frequencies. In fact, at the limit of the audio frequency range you get a low number of samples per cycle.

At each sample time, from 1 to 4 ADC values are recorded and, in some cases, one DAC value is set. ADC readings and DAC writings are not guaranteed to be simultaneous but they will be completed before the next sample. The time uncertainty is then one sample period at the maximum available sample frequency.

Although the board reports the minimum sample period, the time needed to set the DAC and read the ADCs depends on the number of ADCs to read so it is not guaranteed that the board will be able to always maintain the minimum sample period. In case that, for one sample, all writings and readings cannot be completed before the next sample, you will get a **Sample Overrun** exception. In that case increment the sample period to give the DAC and the ADCs more time to work at each sample.

Commands in this section set up the transient measurements for more complex commands in the following sections.

Transient commands, especially when using long sample times, can take some time. Most transient command can be aborted by using the **halt button** in the board when it is available.

setSampleTime(time)

Arguments	time	Time between samples in seconds	V1
Returns		Sample Time	

Base "R"

This command sets the sample time period **Ts** between readings on a transient measurement. Sample time has a four significant digits resolution. The ***printBoardInfo*** command can be used to obtain the valid range of the sample time.
The function returns the set sample time with four digit resolution that is really set on the board.

If no sample time is set, or after a reset, it defaults to 1ms.

Last set sample time is stored in a SLab sampleTime variable so you can refer to its value as ***slab.sampleTime*** if you import the module as default name **slab**. Beware not to modify this SLab internal variable.

Depending on the work requested to the board, the minimum sample time cannot always be met. For instance, a board could operate at a sample time when measuring one ADC but could need to increase the sample time when capturing 2 ADCs.
If the board is still working on a sample when another sample has to be obtained you get a **sample overrun**. In that case, as the interval between samples cannot be guaranteed, the system reports a sample overrun exception showing the following message, and no measurement data is obtained.

```
** SLab exception
** Sample overrun
```

Note that this exception will be thrown during measurements, never when setting the sample time using this command.

Example:
Set sample time to 1ms. That will give a sample frequency of 1kHz.

```
>>> slab.setSampleTime(0.001)
```

setTransientStorage(samples,na)

Arguments	samples	Number of samples to measure	V1
	na	Number of analog channels to read (Defaults to 1)	
Returns		Nothing	

Base "S"

This command has an alias name ***tranStore*** that is shorter to write.

This command determines the amount of data to be recorded when using the commands of the following sections.

The **na** parameter determines the number of ADCs to be read at each sample time. They will always be sampled in order, son if you set **na** as 3, ADC 1, 2 and 3 will be used, in this order, to measure the voltages at each sample time.
If **na** is not provided it defaults to 1, so only ADC 1 will be measured.

Depending on the board hadware and firmware, the reading of several channels is not simultaneous, that means that at high sample rates there will be some skew on the measurements. Moreover, requesting more ADC measurements require more time, so the sample rate limit could be lower when you increase the number of ADCs to read.

The **samples** parameter determines the number of samples to collect for all selected ADCs. The total measurement time **Tm** can be calculated from the sample time **Ts** as:

$$T_m = T_s \cdot samples$$

The required total number of values to collect **Ns** is the product of **samples** and **na**.

$$N_s = na \cdot samples$$

Ns should not exceed the buffer sample size of the hardware board. You can use the ***printBoardInfo*** command in order to check this value. The board uses a unified memory approach so the sample memory is shared between the transient measurements and the wave generator.

Before this command is called, or after a reset, **na** defaults to 1 and **samples** default to 1000.

Example:
Configure transient measurements to measure ADC 1 and 2 at 200 points in time.

```
>>> slab.setTransientStorage(200,2)
```

Or you could also have written with the alias name ***tranStore***:

```
>>> slab.tranStore(200,2)
```

Plot Transient Commands

Commands in this section generate time dependent transient plots using specific Hardware Board orders.

tranAsyncPlot(returnData)

Arguments	returnData	Return plot data (Defaults to False)	V1
Returns	See text. Generates a plot		

Performs a transient measurement using the parameters configured in the previous ***setSampleTime*** and ***setTransientStorage*** commands. The default configuration before calling those commands is to get 1000 samples of ADC 1 at 1kHz frequency for a total 1s measurement time.

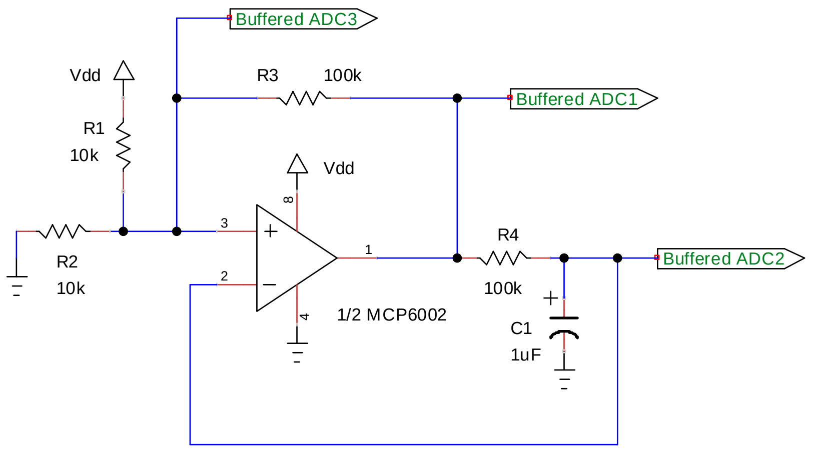
The data obtained from the selected channels is plot against time.
If optional parameter ***returnData*** is True or ***setPlotReturnData(True)*** was called previously, data is also returned as a list of **na**+1 vectors where **na** is the number of analog ADC channels selected with a previous call to ***setTransientStorage***. First vector is time and the rest are the ADC readings for each channel. If you don't need those vectors you can discard them.

If timing requirements for sample time cannot be met, a sample overrun exception is generated.

Example 15A:

Check the operation of an operational amplifier astable circuit.

In this example we are going to test the following circuit. We don't really need ADC 1 to be buffered because the operational amplifier output is at low impedance. ADC 2 and 3, however, need to be buffered to be sure that we don't affect the circuit under test.

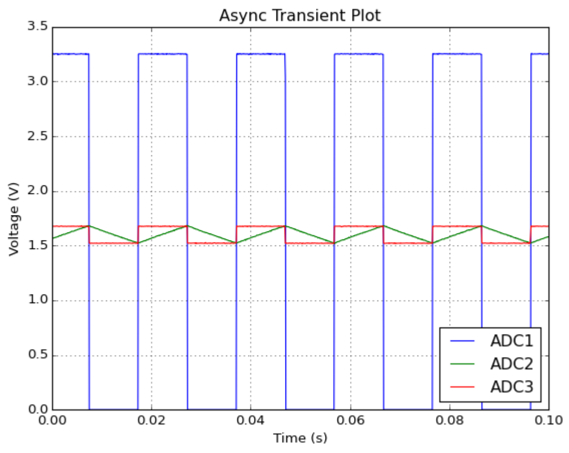


The circuit is an astable circuit built around an inverting hysteresis comparator. The operational amplifier and resistors R1, R2 and R3 compose the comparator. Resistor R4 and capacitor C1 create the astable by feeding the output of a low pass filter driven by the comparator to the input of the comparator.

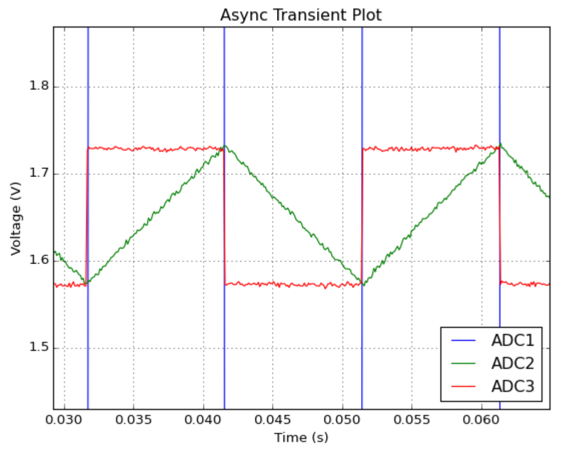
The following code is used to measure the output of the comparator at ADC1 and the input at ADC2.

```
>>> slab.setSampleTime(0.0001)
>>> slab.setTransientStorage(3,0,1000)
>>> slab.tranAsyncPlot()
```

First line sets sample time to 100µs. That is equivalent to a sample frequency of 10 kHz. Second line sets the board to store three readings ADC1, ADC2 and ADC3 at each sample point. Finally, the third line performs the measurement so we get the following image:



From the image, using pan and zoom we can obtain the operating frequency (50.6 Hz) and the voltage trip points of the comparator (1.29V and 1.75V).



We can also see how the comparator threshold (ADC3) changes when the output of the opamp changes. See how the output changes each time the two opamp inputs (ADC2 and ADC3) have the same value.

In the above example we have discarded the data vectors. If we want to post process them we can use the following alternative last line:

```
>>> setPlotReturnData(True)
>>> time,adc1,adc2,adc3 = slab.tranAsyncPlot()
```

If there is a **HALT** button on the board, you can use it to abort the command.

tranTriggeredPlot(level, mode, timeout, returnData)

Arguments	level	Voltage for the trigger on ADC 1	V1
	mode	Mode *tmodeRise* or *tmodeFall* (Defaults to *tmodeRise*)	
	timeout	Timeout in integer seconds (Defaults to no timeout)	
	returnData	Return of plot data (Defaults to False)	
Returns		See text. Generates a plot	

Performs a transient measurement using the parameters configured in the previous ***setSampleTime*** and ***setTransientStorage*** commands. The default configuration before calling those commands is to get 1000 samples of ADC 1 at 1kHz frequency. The command will wait to start the measurement until a trigger condition is met. Half of the samples will be obtained before the trigger condition and the other half after that.

The trigger condition is composed of a trigger voltage on ADC 1 and a trigger mode that can be ***tModeRise*** (voltage crosses the trigger voltage when rising) or ***tModeFall*** (cross at rise). By default, if no mode is provided, rise mode will be used.

The data obtained from the selected channels is plot against time setting zero time at the trigger point.

If optional parameter ***timeout*** is used, the triggering will timeout after the selected number of seconds. If a triggering condition has not been detected before the timeout, a timeout exception will be generated. The maximum timeout value is 255 seconds.

If optional parameter ***returnData*** is True or ***setPlotReturnData(True)*** was called previously, data is also returned as a list of **na**+1 vectors where **na** is the number of analog ADC channels selected with a previous call to ***setTransientStorage***. First vector is time and the rest are the ADC readings for each channel. If you don't need those vectors you can discard them.

If timing requirements for sample time cannot be met, a sample overrun exception is generated.

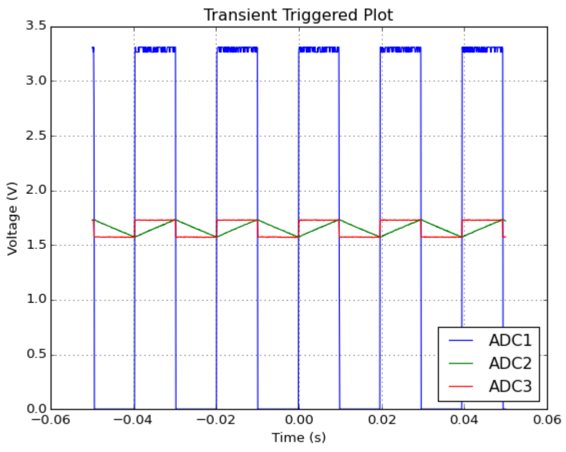
If there is a **HALT** button on the board, you can use it to abort the command.
That is especially interesting for this command as it can lock the board if the triggering condition never takes place.

Example 15B:
Perform a triggered read capture of an astable circuit.

We will use the same circuit of Example 15A shown at the description of the previous ***tranAsyncPlot*** command. In order to capture around the rising time of the operational amplifier output we use the command:

```
>>> slab.tranTriggeredPlot(1.6,slab.tmodeRise)
```

And we will get the following plot:



Observe that trigger zero time is just on the rising edge of the ADC1 signal.

In the above example we have discarded the data vectors. If we want to post process them we can use the following alternative lines:

```
>>> setPlotReturnData(True)
>>> data = slab.tranTriggeredPlot(1.6,slab.tmodeRise)
```

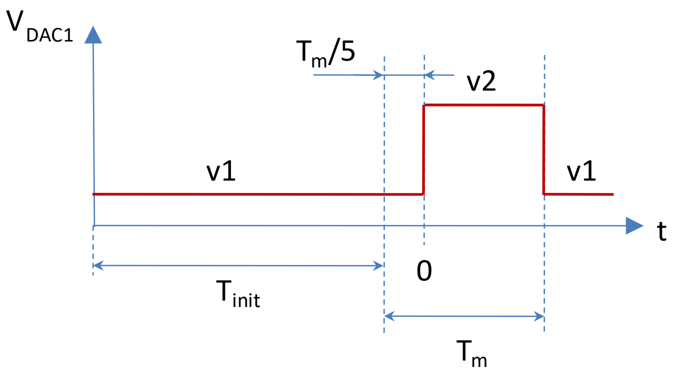
stepPlot (v1, v2, tinit, returnData)

Arguments	v1	Voltage before time zero	V1
	v2	Voltage after time zero	
	tinit	Initialization time (Defaults to one second)	
	returnData	Return of plot data (Defaults to False)	
Returns		List of vectors (Also generates a plot)	

Performs a step response plot. Mesurement is performed during the time selected in the previous ***setSampleTime*** and ***setTransientStorage*** commands. A step between voltages v1 and v2 is performed on DAC 1 at 1/5 of the measuring time. After the measurement time ends, DAC 1 returns to the v1 value.

Before the measurement, an initialization time ***tinit*** tnat defaults to one second is added to guarantee that the circuit is stable before the measurement. No measurements are performed during the initialization time.

The following figure shows the waveform generated at DAC 1. The obtained plot will only include the T_m measurement time and the zero time will be set at the step change from v1 to v2.



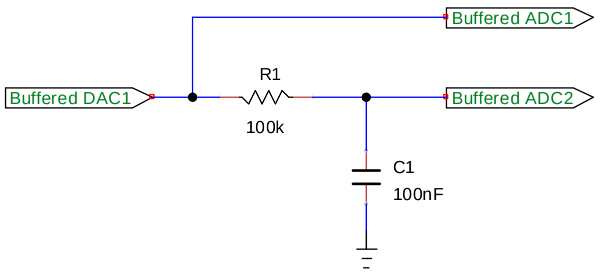
If optional parameter ***returnData*** is True or ***setPlotReturnData(True)*** was called previously, data is also returned as a list of **na**+1 vectors where **na** is the number of analog ADC channels selected with a previous call to ***setTransientStorage***. First vector is time and the rest are the ADC readings for each channel. If you don't need those vectors you can discard them.

If timing requirements for sample time cannot be met, a sample overrun exception is generated.

Example 16:

Measure the step response of a low pass RC filter.

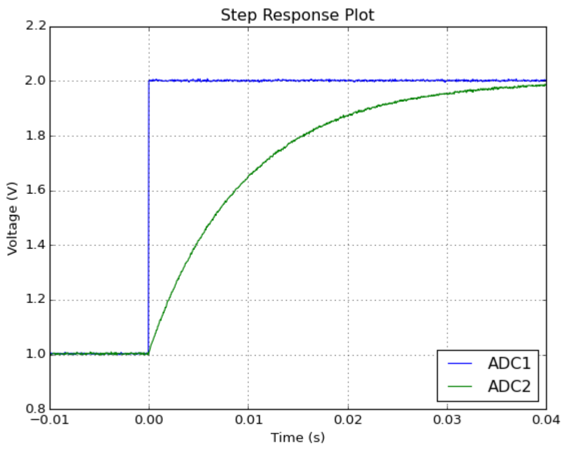
We setup the following circuit for the measurement:



Then we input the following commands that set the sample time to 50us, and request storage for ADC 1 and 2 for 1000 samples. That gives a total measurement time of 50ms. Finally we request the step plot.

```
>>> slab.setSampleTime(0.00005)
>>> slab.setTransientStorage(2,0,1000)
>>> slab.stepPlot(1.0,2.0)
```

That will give the following step plot response:



We observe that the time constant of the circuit is 9.4ms quite close to the expected 10ms value. Note that total measurement time is 50ms and we get 1/5 (10ms) before the step and 4/5 (40ms) after the step. If we try to repeat the measurement using a too small sample time we could get a sample overrun exception.

In the above example we have discarded the data vectors. If we want to post process them we can use the following alternative line:

```
>>> time,adc1,adc2 = slab.stepPlot(1.0,2.0,returnData=True)
```

If there is a **HALT** button on the board, you can use it to abort the command.

Non plotting Transient Commands

In the same way than the DC plot commands depend on the Complex DC commands to operate, the transient plot commands depend on the commands described in this section. Those commands provide voltage vectors that include equally spaced voltage measurements at a constant sample rate.

Using those vectors complex operations can be performed on the sampled data without generating automatic plots.

transientAsync()

Arguments	None	V1
Returns	Variable list of vectors	

Base "Y"

This command is equivalent to the ***tranAsyncPlot*** command but no plot is generated. Only the vector return list is obtained. In fact ***tranAsyncPlot*** calls this command to obtain the plot data.

transientTriggered(level, mode, timeout)

Arguments	level	Voltage for the trigger on ADC 1	V1
	mode	Mode *tmodeRise* or *tmodeFall* (Defaults to *tmodeRise*)	
	timeout	Timeout in integer seconds (Defaults to no timeout)	
Returns		Variable list of vectors	

Base "G"

This command is equivalent to the ***tranTriggeredPlot*** command but no plot is generated. Only the vector return list is obtained. In fact ***tranTriggeredPlot*** calls this command to obtain the plot data.

stepResponse(v1, v2, tinit)

Arguments	v1	Voltage before time zero	V1
	v2	Voltage after time zero	
	tinit	Initialization time (Defaults to one second)	
Returns		Variable list of vectors	

Base "P"

This command is equivalent to the ***stepPlot*** command but no plot is generated. Only the vector return list is obtained. In fact ***stepPlot*** calls this command to obtain the plot data.

Wave Commands

The previous transient commands are useful to test circuits that generate their own signals or to check simple responses like the step response. In order to test the operation of a circuit against a specific waveform we can use the wave commands in this section.

A wave is composed of a sequence of **np** values to be sent to DAC 1. The Slab module can generate waves of standard types like square, triangle, sawtooth and sine or arbitrary generated waveforms.

Wave frequency f_W, sample frequency f_S, sample time T_S and np are related by the equations:

$$f_S = \frac{1}{T_S} \qquad f_W = \frac{f_S}{np} = \frac{1}{np \cdot T_S}$$

The minimum sample time $T_{S\,min}$ for the board limits the maximum frequency of the generated wave $f_{W\,max}$.

$$f_{W\,max} = \frac{1}{np \cdot T_{S\,min}}$$

Approximate limits of wave frequency are calculated form **np** when loading a waveform, but as board CPU load depends on requested number of ADCs conversions and other details you can always get a **sample overrun** exception during the measurements. In that case you need to work with sample times not just at the minimum value but above it.

For instance, trying to generate a 400Hz sinewave with 100 points per cycle require a sample frequency of 40kHz that relates to a 25µs sample time. If you get a **sample overrun** for this signal, you can reduce the number of points per cycle to 50 to increase the sample time to 50µs.

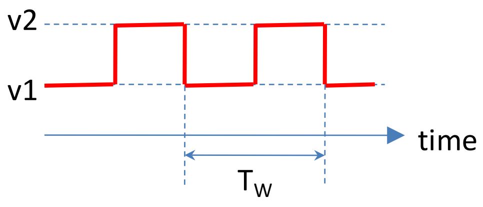
In theory, you need at least two points per cycle to generate a sine wave, but this is only true for a perfect anti imaging reconstruction filter after the DAC. If such filter is not available, you will get a tradeoff between the maximum wave frequency and the quality of the generated wave measured as the number of points per cycle.

The SLab system allows dual wave generation on DAC1 and DAC2. A secondary wave can be loaded by using the ***second*** parameter in wavetable loading commands. Sample memory on the hardware board needs to have space for both the primary and the secondary waves. Frequency cannot be adjusted on the secondary wave, the sample rate will the one selected on the primary wave.
As the sample memory is shared with the transient measurements, using too much memory for wave generation limits the size of the transient measurements.

waveSquare(v1,v2,np,returnList, second)

Arguments	v1	Start value (in Volts)	V1
	v2	End value (in Volts)	
	np	Number of points	
	returnList	Request returning the list of values (Defaults to False)	
	second	Load as secondary wave (Defaults to False)	
Returns		List of values if returnList is true	

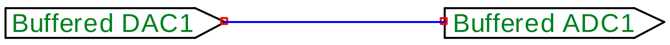
Load a square wave in the hardware board composed of **np** points. The first half will be at **v1** value and the second half will be at **v2** value.



The command returns a list of the **np** voltage values of the wave if the ***returnList*** parameter is True.
The wavetable is loaded for the secondary wave on DAC2 if the ***second*** parameter is True.

Example 22A:
Create a square waveform

We will create a square waveform, send it to the board though DAC 1 and read it using ADC 1. We connect the DAC 1 output to the ADC input.

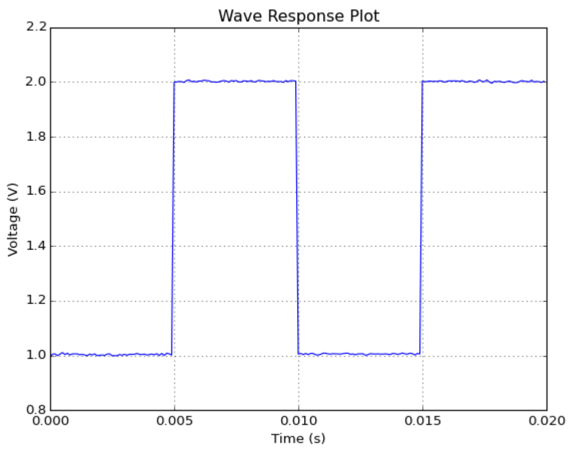


The following code performs all operations:

```
>>> slab.setSampleTime(0.0001)
>>> slab.setTransientStorage(1,0,200)

>>> slab.waveSquare(1.0,2.0,100)
>>> slab.wavePlot(1)
```

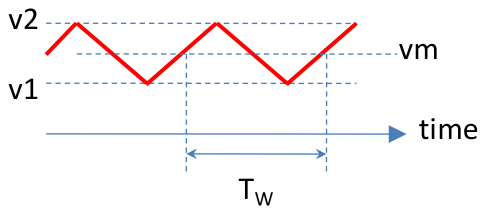
The last command will show:



waveTriangle(v1,v2,np,returnList, second)

Arguments	v1	Start value (in Volts)	V1
	v2	End value (in Volts)	
	np	Number of points	
	returnList	Request returning the list of values (Defaults to False)	
	second	Load as secondary wave (Defaults to False)	
Returns		List of values if returnList is true	

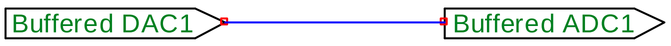
Load a triangle wave in the hardware board composed of **np** points. The wave will start at the mean value **vm** between **v1** and **v2**, go to **v2**, go to **v1**, and return to the start point.



The command returns a list of the **np** voltage values of the wave if the ***returnList*** parameter is True.
The wavetable is loaded for the secondary wave on DAC2 if the ***second*** parameter is True.

Example 22B:
Create a triangular waveform

We will create a triangular waveform, send it to the board though DAC 1 and read it using ADC 1. We connect the DAC 1 output to the ADC input.

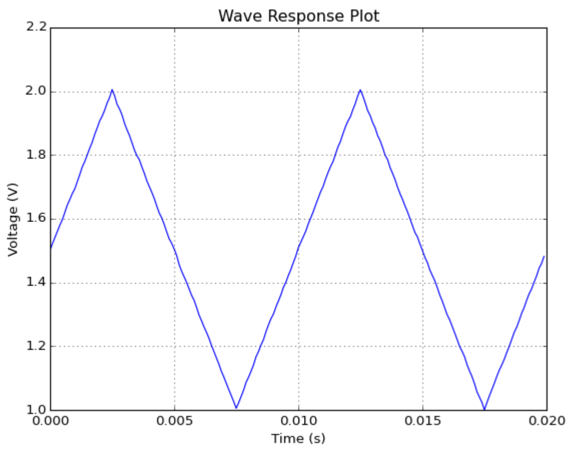


The following code performs all operations:

```
>>> slab.setSampleTime(0.0001)
>>> slab.setTransientStorage(1,0,200)

>>> slab.waveTriangle(1.0,2.0,100)
>>> slab.wavePlot(1)
```

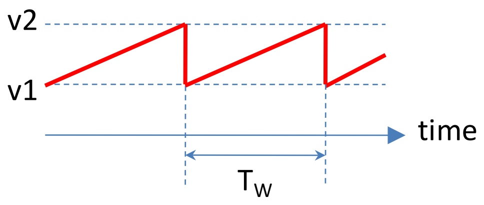
The last command will show:



waveSawtooth(v1,v2,np,returnList, second)

Arguments	v1	Start value (in Volts)	V1
	v2	End value (in Volts)	
	np	Number of points	
	returnList	Request returning the list of values (Defaults to False)	
	second	Load as secondary wave (Defaults to False)	
Returns		List of values if returnList is true	

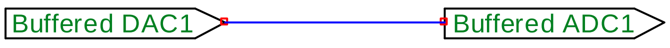
Load a sawtooth wave in the hardware board composed of **np** points. The wave will start at **v1**, go to **v2** during T_W, and immediately return to **v1** to repeat.



The command returns a list of the **np** voltage values of the wave if the ***returnList*** parameter is True.
The wavetable is loaded for the secondary wave on DAC2 if the ***second*** parameter is True.

Example 22C:
Create a sawtooth waveform

We will create a sawtooth waveform, send it to the board though DAC 1 and read it using ADC 1. We connect the DAC 1 output to the ADC input.

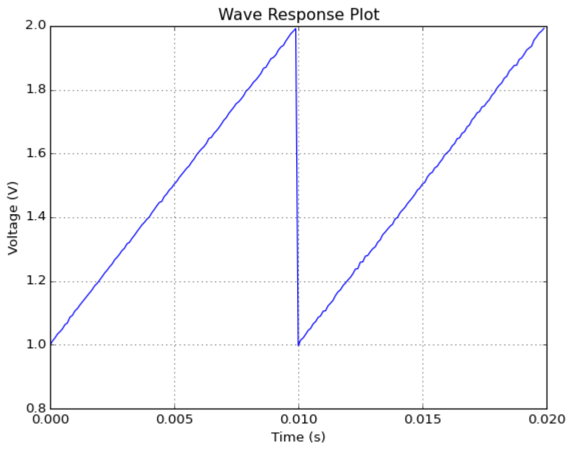


The following code performs all operations:

```
>>> slab.setSampleTime(0.0001)
>>> slab.setTransientStorage(1,0,200)

>>> slab.waveSawtooth(1.0,2.0,100)
>>> slab.wavePlot(1)
```

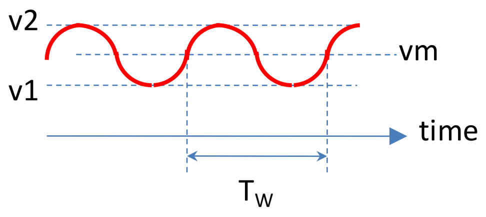
The last command will show:



waveSine(v1, v2, np, phase, returnList, second)

Arguments	v1	Start value (in Volts)	V1
	v2	End value (in Volts)	
	np	Number of points	
	phase	Phase (deg) (Defaults to zero)	
	returnList	Request returning the list of values (Defaults to False)	
	second	Load as secondary wave (Defaults to False)	
Returns		List of values if returnList is true	

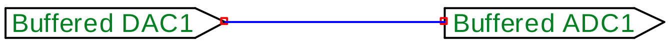
Load a sine wave in the hardware board composed of **np** points. The wave will start at the mean value **vm** between **v1** and **v2**, go to **v2**, go to **v1**, and return to the start point.



The indicated phase will be added if the ***phase*** parameter is used.
The command returns a list of the **np** voltage values of the wave if the ***returnList*** parameter is True.
The wavetable is loaded for the secondary wave on DAC2 if the ***second*** parameter is True.

Example 22D:
Create a sine waveform

We will create a sine waveform, send it to the board though DAC 1 and read it using ADC 1. We connect the DAC 1 output to the ADC input.

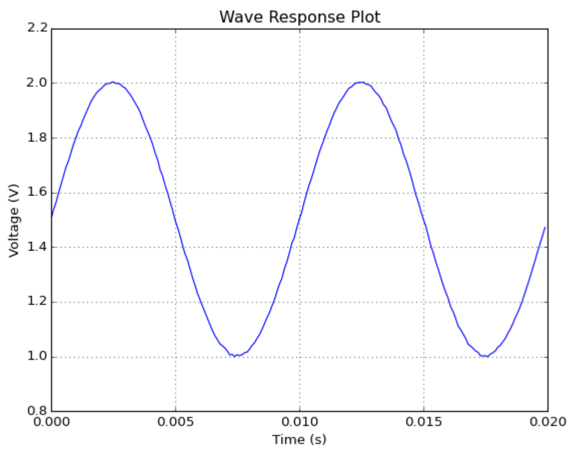


The following code performs all operations:

```
>>> slab.setSampleTime(0.0001)
>>> slab.setTransientStorage(1,0,200)

>>> slab.waveSine(1.0,2.0,100)
>>> slab.wavePlot(1)
```

The last command will show:



waveCosine(v1, v2, np, phase, returnList, second)

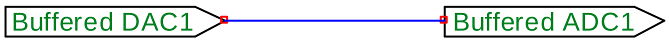
Arguments	v1	Start value (in Volts)	V1
	v2	End value (in Volts)	
	np	Number of points	
	phase	Phase (deg) (Defaults to zero)	
	returnList	Request returning the list of values (Defaults to False)	
	second	Load as secondary wave (Defaults to False)	
Returns		List of values if returnList is true	

Load a cosine wave in the hardware board composed of **np** points. The wave will start at **v2**, go to **v1**, and return to the start point.

The indicated phase will be added if the ***phase*** parameter is used.
The command returns a list of the **np** voltage values of the wave if the ***returnList*** parameter is True.
The wavetable is loaded for the secondary wave on DAC2 if the ***second*** parameter is True.

Example 22G:
Create a cosine waveform

We will create a cosine waveform, send it to the board though DAC 1 and read it using ADC 1. We connect the DAC 1 output to the ADC input.

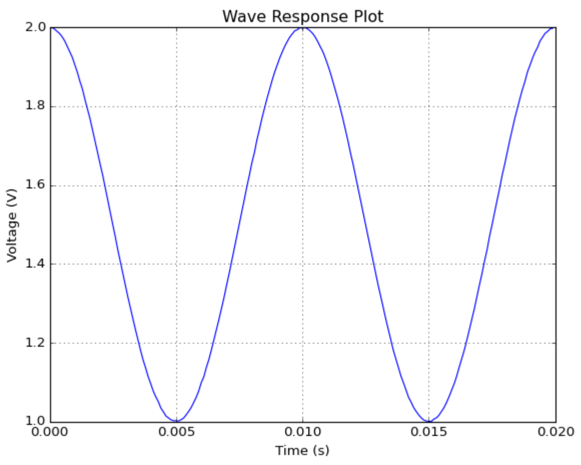


The following code performs all operations:

```
>>> slab.setSampleTime(0.0001)
>>> slab.setTransientStorage(1,0,200)

>>> slab.waveCosine(1.0,2.0,100)
>>> slab.wavePlot(1)
```

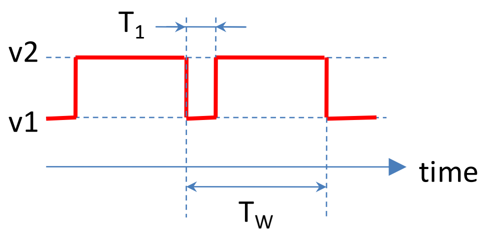
The last command will show:



wavePulse(v1,v2,np,n1, returnList, second)

Arguments	v1	Start value (in Volts)	V1
	v2	End value (in Volts)	
	np	Number of points	
	n1	Number of points at v1 value	
	returnList	Request returning the list of values (Defaults to False)	
	second	Load as secondary wave (Defaults to False)	
Returns		List of values if returnList is true	

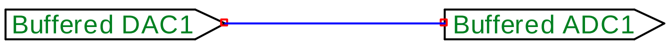
Load a pulse wave in the hardware board composed of **np** points. The first **n1** points will be at **v1** value and the rest will be at **v2** value.



The command returns a list of the **np** voltage values of the wave if the ***returnList*** parameter is True.
The wavetable is loaded for the secondary wave on DAC2 if the ***second*** parameter is True.

Example 22E:
Create a pulse waveform

We will create a pulse waveform, send it to the board though DAC 1 and read it using ADC 1. We connect the DAC 1 output to the ADC input.

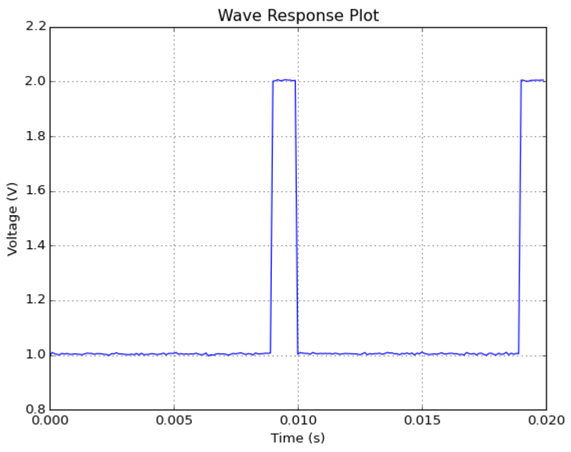


The following code performs all operations. Note that the last parameter in wavePulse is the number of samples at 1.0V, so, to get a 10% duty cycle we need to set this value to 90.

```
>>> slab.setSampleTime(0.0001)
>>> slab.setTransientStorage(1,0,200)

>>> slab.wavePulse(1.0,2.0,100,90)
>>> slab.wavePlot(1)
```

The last command will show:



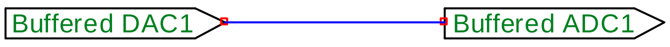
waveNoise(vm, vstd, n, returnList, second)

Arguments	vm	Mean value (in Volts)	V1
	vstd	Standard deviation (in Volts)	
	n	Number of points	
	returnList	Request returning the list of values (Defaults to False)	
	second	Load as secondary wave (Defaults to False)	
Returns		List of values if returnList is true	

Load a noise wave in the hardware board composed of **n** points. The waveform is taken from random samples that follow a normal distribution with the indicated mean value **vm** and standard deviation **vstd**.
As normal distribution is unbound, samples will be truncated if they go outside of the available DAC range.
The command returns a list of the **n** voltage values of the wave if the ***returnList*** parameter is True.
The wavetable is loaded for the secondary wave on DAC2 if the ***second*** parameter is True.

Example 22H:
Create a noise waveform

We will create a gaussian noise waveform, send it to the board though DAC 1 and read it using ADC 1. We connect the DAC 1 output to the ADC input.

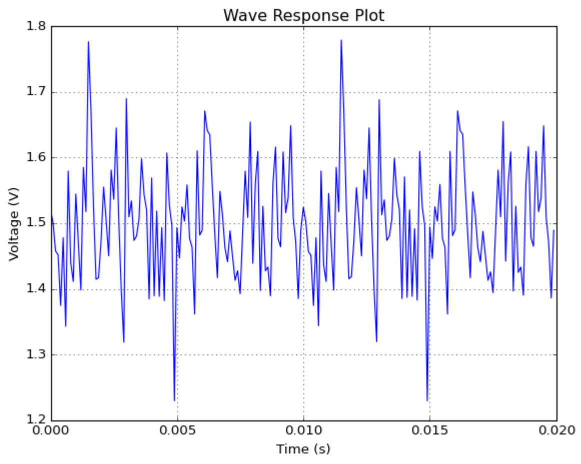


The following code performs all operations. Noise will be centered at 1.5V and have a std deviation of 0.1V. We will also measure the signal standard deviation.

```
>>> slab.setSampleTime(0.0001)
>>> slab.setTransientStorage(1,0,200)

>>> slab.waveNoise(1.5,0.1,100)
>>> t,a1 = slab.wavePlot(1,returnData=True)
>>> print "Std Dev is " + slab.std(a1) + " V"
```

The plot command will show:



And the std deviation is shown as:

```
Std Dev is 0.0995827973856 V
```

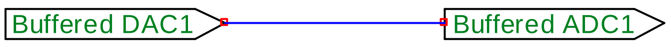
waveRandom(v1, v2, n, returnList, second)

Arguments	v1	Minimum value (in Volts)	V1
	v2	Maximum value (in Volts)	
	n	Number of points	
	returnList	Request returning the list of values (Defaults to False)	
	second	Load as secondary wave (Defaults to False)	
Returns		List of values if returnList is true	

Load a random wave in the hardware board composed of **n** points. The waveform is taken from random samples that follow a uniform distribution with the indicated minimum value **v1** and maximum value **v2**.
The command returns a list of the **n** voltage values of the wave if the ***returnList*** parameter is True.
The wavetable is loaded for the secondary wave on DAC2 if the ***second*** parameter is True.

Example 22I:
Create a random waveform

We will create a random waveform, send it to the board though DAC 1 and read it using ADC 1. We connect the DAC 1 output to the ADC input.

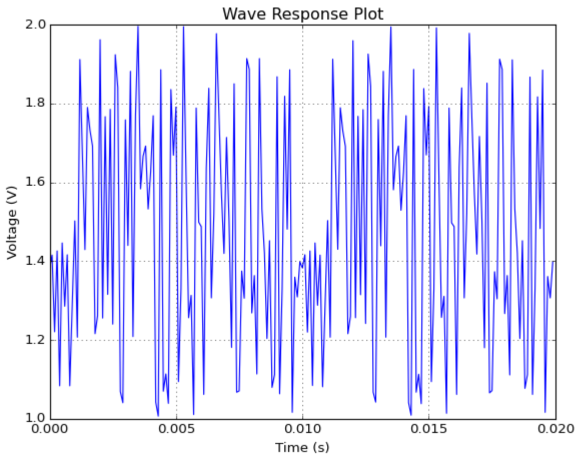


The following code performs all operations. The waveform will be random values between 1 V and 2 V.

```
>>> slab.setSampleTime(0.0001)
>>> slab.setTransientStorage(1,0,200)

>>> slab.waveRandom(1,2,100)
>>> slab.wavePlot(1
```

The last command will show:



loadWavetable(list, second)

Arguments	list	List of values	V1
	second	Load as secondary wave (Defaults to False)	
Returns		Nothing	

Base "W", "w"

Loads an arbitrary waveform defined as a list of float voltage values. The number of points of the wave will be the number of float values included in the provided list.
All the previous commands in this section use this base command to load the wave on the hardware board.

The hardware board firmware uses an unified memory model, so the buffer space for transient measurements is decreases when waveforms are loaded.

The wavetable is loaded for the secondary wave on DAC2 if the ***second*** parameter is True. The secondary wavetable is erased when a primary wavetable is loaded, so the wavetables shall be loaded in order.

Example 22F:
Create a stair waveform

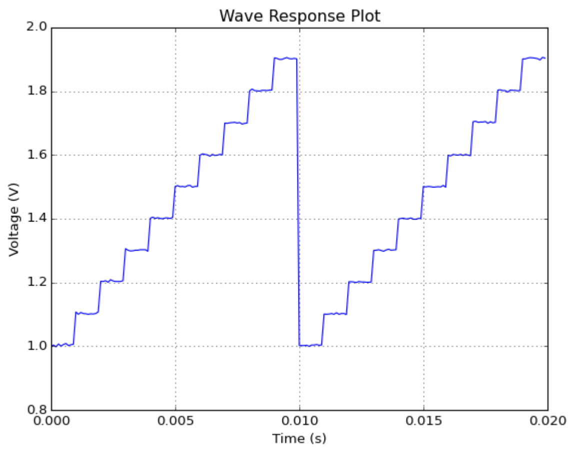
We can create a stair waveform by creating a list of 100 elements in subgroups of 10 equal value voltages:

```
>>> slab.setSampleTime(0.0001)
>>> slab.setTransientStorage(1,0,200)

>>> list = []
>>> for i in range(0,10):
>>>     for j in range(0,10):
>>>         list.append(1.0+0.1*i)
>>> slab.loadWavetable(list)

>>> slab.wavePlot(1)
```

The last command will show:



setWaveFrequency(freq)

Arguments	freq	Frequency to set (in Hz)	V1
Returns		Frequency set (in Hz)	

Changes the main wave frequency by adjusting the sample time. For a **np** points waveform of frequency f_W, the sample time T_S can be calculated:

$$T_S = \frac{1}{f_W \cdot np}$$

Observe that the frequency is calculated from the number of points of the last primary waveform that has been loaded.

An exception is generated if the T_S value cannot be set on the board.
If the T_S time is close to the minimum limit for the board, a **sample overrun** exception could happen during the next generation of the wave in the hardware board.

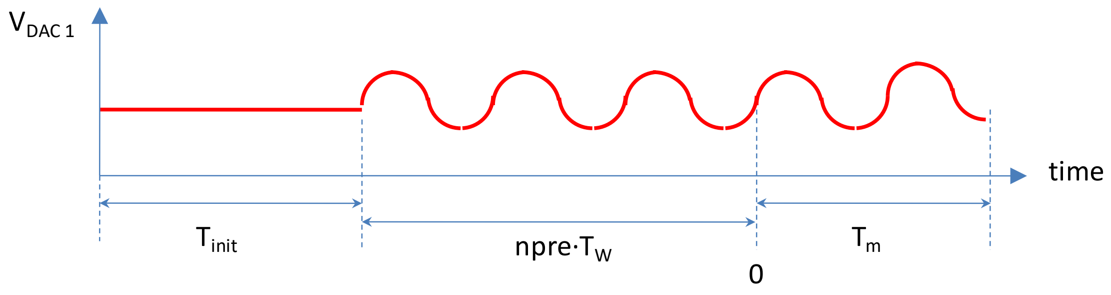
If there is a secondary wave loaded, it uses the same sample frequency than the primary wave.

Returns the real sample time set on the board. It can be different from the theoretical one because the sample time is given with four digit precision.

wavePlot(npre, tinit, dual, returnData)

Arguments	npre	Number of wave cycles before measurement (Defaults to zero)	V1
	tinit	Initialization time (Defaults to 1 second)	
	dual	Generate signals also on DAC2 (Defaults to False)	
	returnData	Return plot data (Defaults to False)	
Returns		See text. Generates a plot	

Plots the response of a circuit under test (CUT) against an input waveform generated on DAC 1.
A waveform must be previously selected using one of the previous commands.
The following figure shows the signal generated on DAC 1:



First, during a **tinit** time (that defaults to 1 second) the first value of the wave is maintained at the DAC 1 output. Then, **npre** full cycles of the wave are generated. Afterwards, the system output is measured during a measurement time T_m as indicated on a previous ***setTransientStorage*** command while the wave keeps being generated.
Note that T_m does not need to include a complete number of waveforms.
Data is only measured during the Tm time (from time 0 onwards). After the data is uploaded to the computer, the DAC 1 is set to the first wave value.

If optional parameter ***returnData*** is True or ***setPlotReturnData(True)*** was called previously, data is also returned as a list of **na**+1 vectors where **na** is the number of analog ADC channels selected with a previous call to ***setTransientStorage***. First vector is time and the rest are the ADC readings for each channel.

If the ***dual*** parameter is True, signals will be generated both on DAC1 and DAC2. DAC2 signal will use a previously loaded secondary wavetable. Note that both waves will start at the same time after **Tinit**. So, if ***npre*** is not zero and they have different frequencies they could be delayed by the start of the measurement.

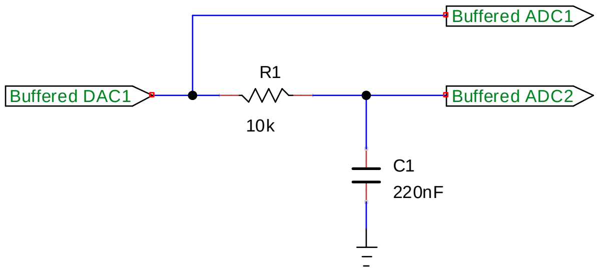
Example 17:

Show the response of a low pass filter to a tone at the corner frequency.

We will set a RC filter with R = 10kΩ and C = 220nF. That will give:

$$\omega_{bw} = \frac{1}{R \cdot C} = 454 \frac{rad}{s} \qquad f_{bw} = \frac{\omega_{bw}}{2\pi} = 72.3\ Hz$$

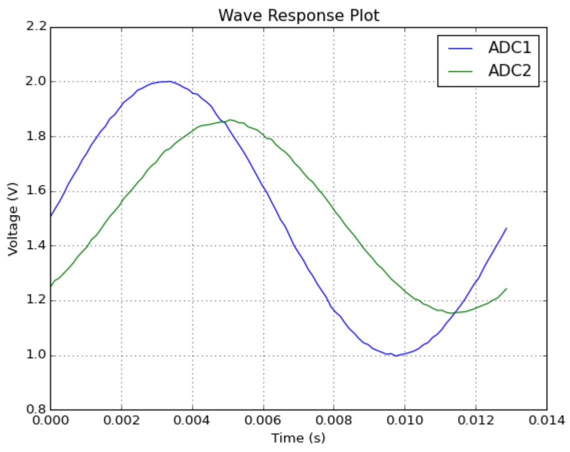
The circuit to test will be:



In order to make the measurements, we can use the following python code. It generates a 100 point sinewave at 72.3Hz and generates the response plot obtaining 100 readings (one full cycle) on ADC1 and ADC 2. A 10 cycle wave is generated previous to the measurement to guarantee that we are measuring the forced response of the circuit.

```
>>> slab.setTransientStorage(2,0,100)
>>> slab.waveSine(1.0,2.0,100)
>>> slab.setWaveFrequency(72.3)
>>> slab.wavePlot(10)
```

After the last line we will obtain the following plot.



If there is a **HALT** button on the board, you can use it to abort the command.

singleWavePlot(channel,npre, tinit, returnData)

Arguments	channel	Channel ADC to read (Defaults to 1)	V1
	npre	Number of wave cycles before measurement (Defaults to zero)	
	tinit	Initialization time (Defaults to 1 second)	
	returnData	Return plot data (Defaults to False)	
Returns		See text. Generates a plot	

This command is similar to the ***wavePlot*** command. The differences are:

- ***wavePlot*** stores information on the number of channels selected on the previous ***setTransientStorage*** command, always starting at ADC 1.
- ***singleWavePlot*** stores information only on the ADC channel selected in the argument regardless of the number of analog channels selected previously.

As this command only stores data from one ADC, it is possible to scan several channels at higher sample rates than in the ***wavePlot*** command. Note that channel data won't correspond to simultaneous captures although they will always synchronize with the DAC 1 output.

If you only need information for one channel, it is better to use the ***singleWavePlot*** command as its performance is better optimized for single captures.

If there is a **HALT** button on the board, you can use it to abort the command.

waveResponse(npre, tinit, dual)

Arguments	npre	Number of wave cycles before measurement (Defaults to zero)	V1
	tinit	Initialization time (Defaults to one second)	
	dual	Generate signals also on DAC2 (Defaults to False)	
Returns		List of values	

Base "V" , “v”

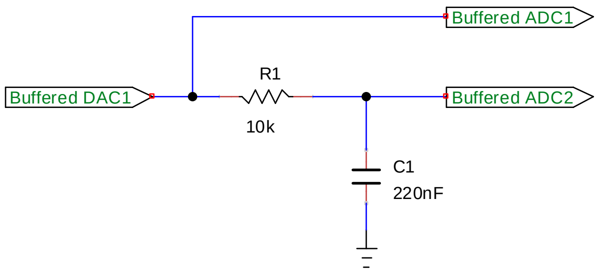
This command is equivalent to the ***wavePlot*** command but no plot is generated so only the vector return list is obtained. In fact, ***wavePlot*** calls this base command to get the plot data.

Example 18:
Show the voltage and current of a filter capacitor at the corner frequency.

We will set a RC filter as in example 17 with R = 10kΩ and C = 220nF. That will give:

$$\omega_{bw} = \frac{1}{R \cdot C} = 454 \frac{rad}{s} \qquad f_{bw} = \frac{\omega_{bw}}{2\pi} = 72.3\ Hz$$

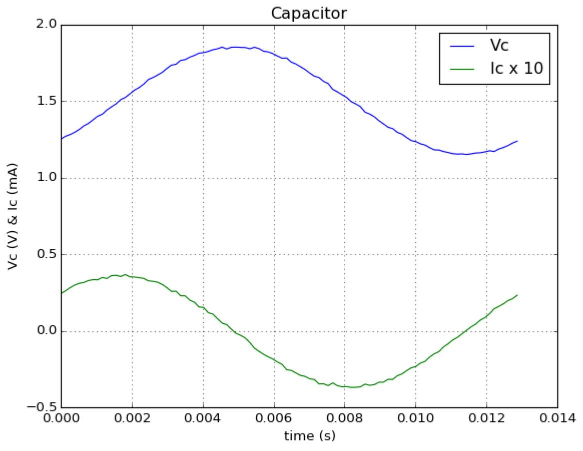
The circuit to test will be:



In order to make the measurements, we can use the following python code. It generates a 100 point sinewave at 72.3Hz and generates obtains the wave response during 100 readings (one full cycle) on ADC1 and ADC 2. A 10 cycle wave is generated previous to the measurement to guarantee that we are measuring the forced response of the circuit.

```
>>> slab.setTransientStorage(2,0,100)
>>> slab.waveSine(1.0,2.0,100)
>>> slab.setWaveFrequency(72.3)
>>> time,v1,v2 = slab.waveResponse(10)
>>> ic = v1-v2   # In mA * 10
>>> slab.plot1n(time,[v2,ic],"Capacitor","time (s)","Vc (V) & Ic (mA)",["Vc","Ic x 10"])
```

After the measurement, the capacitor current is computed and shown together with the voltage.



Note that at corner frequency capacitor reactance equals the resistance so:

$$|Z| = \sqrt{10k\Omega^2 + 10k\Omega^2} = 14k\Omega \qquad i_c = \frac{0.5\mathrm{V}}{14k\Omega} = 0.035\mathrm{mA}$$

That agrees with the results obtained in the plot.

singleWaveResponse(channel,npre, tinit)

Arguments	channel	Channel ADC to read (Defaults to 1)	V1
	npre	Number of wave cycles before measurement (Defaults to zero)	
	tinit	Initialization time (Defaults to one second)	
Returns		List of values	

Base "X"

This command is equivalent to the ***singleWavePlot*** command but no plot is generated so only the vector return list is obtained. In fact, ***singleWavePlot*** calls this base command to get the plot data.

wavePlay(n, tinit, dual)

Arguments	n	Number of wave cycles to send (Defaults to one)	V1
	tinit	Initialization time (Defaults to one second)	
	dual	Generate signals also on DAC2 (Defaults to False)	
Returns		Nothing	

Base "Q" , “q”

This command is similar to the ***waveResponse*** command but no measurements are preformed. It just sends the primary waveform to DAC1 the number ***n*** of times after a ***tinit*** start idle time.
If the parameter **n** is set to zero, the board will generate waveforms forever. You can always abort the generation using the **HALT** button if available on the board.
If the optional ***dual*** parameter is set to True, waves from the secondary wavetable are also sent to DAC2.

Vector Utility Commands

The commands in this section work with vectors or lists. As these commands don't use the hardware board, all of them are Auxiliary and you don't need to connect to use them. Some of these commands can be performed with equivalent standard Python or numpy functions.

highPeak(vector)

Arguments	vector	List or numpy array of values	V1
Returns		Maximum value on the list/array	

Auxiliary

Maximum from all values in a vector.

lowPeak(vector)

Arguments	vector	List or numpy array of values	V1
Returns		Minimum value on the list/array	

Auxiliary

Minimum from all values in a vector.

peak2peak(vector)

Arguments	vector	List or numpy array of values	V1
Returns		Value of maximum - minimum	

Auxiliary

Distance between maximum and minimum value:

$$peak2peak(v) = highPeak(v) - lowPeak(v)$$

halfRange(vector)

Arguments	vector	List or numpy array of values	V1
Returns		Half range between maximum and minimum	

Auxiliary

Calculates the half range between maximum and minimum:

$$halfRange(v) = \frac{highPeak(v) + lowPeak(v)}{2}$$

mean(vector)

Arguments	vector	List or numpy array of values	V1
Returns		Average of the values of list/array	

Auxiliary

Mean for all values in the vector:

$$mean(v) = \bar{v} = \frac{1}{n}\sum_{i=1}^{n} v_i$$

rms(vector)

Arguments	vector	List or numpy array of values	V1
Returns		RMS of the vector values	

Auxiliary

Root Mean Square (RMS) of all values in the vector:

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

The function does not subtract the mean value from the signal.

std(vector)

Arguments	vector	List or numpy array of values	V1
Returns		Standard deviation of values of list/array	

Auxiliary

Standard deviation of values in the vector:

$$std(v) = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(v_i - \bar{v})^2}$$

It equals the RMS value when the vector mean is zero.

Digital I/O Commands

The Hardware Board features digital I/O lines that complement the DAC and ADC lines. As the SLab system is mainly developed for analog measurement, current board firmware is limited in the usage of the digital lines.
The number **ndio** of available digital lines is reported with the ***printBoardInfo*** command.
Digital I/O lines are numbered from 1 to **ndio**. There is no zero line.

Basically we can set any line to several input and output modes. After a line is set to a mode, we can write or read the digital value of this line.

Digital I/O lines work with only two possible states True (or High Level) or False (or Low Level).

dioMode(line, mode)

Arguments	line	Digital I/O line to configure	V1
	mode	Mode to set (Defaults to *mInput*)	
Returns		Nothing	

Base "H"

Configures the line number ***line*** to the mode set with ***mode***. If no mode is provided it defaults to ***mInput***. The available modes are:

mInput : Normal input mode
mPullUp : Input mode with Pull-Up resistor
mPullDown : Input mode with Pull-Down resistor
mOutput : Output push-pull mode
mOpenDrain : Output Open Drain mode

Those modes should all be supported on the Nucleo Boards. For other hardware boards, they could not support all the modes. Check the board documentation to find about the available modes. In any case, if you try to use a mode not available on the board, you will get a remote board error exception like:

```
** SLab exception
** Remote Error : Bad command parameters
```

dioWrite(line, value)

Arguments	line	Digital I/O line to configure	V1
	mode	Value to set (True of False)	
Returns		Nothing	

Base "J"

Write the line number ***line*** to the value set with ***value***. Value can be True (High Level) or False (Low Level).
If the line was configured in one of the input modes, this command has no effect.

dioRead(line)

Argument	line	Digital I/O line to read	V1
Returns		Line state (True or False)	

Base "K"

Read the line number ***line*** to the value set with ***value***. The command returns the state of the line that can be True (High Level) or False (Low Level).
If the line was configured in one input mode, the state of the input is read.
If the line was configured in one output mode, you should read the last state that was set.

Calibration Commands

This section deals with calibration of the measurements performed on the hadware board. Although this is the last section, it is very important if you want to obtain measurements that are near the real physical values of the circuit.

Calibration should be performed periodically to correct time and ambient deviations of the hardware board and should also be performed whenever the buffering circuits of ADCs or DACs are changed.

This section of the reference will show first the calibration commands. Afterwards a step by step calibration procedure will be described.

setVdd(value, persistent)

Arguments	value	Voltage at Vdd (in Volts)	V1
	persistent	Make value persistent in future connections to board (False by default)	
Returns		Nothing	

This command sets the value of the Vdd board supply voltage.

setVref(value, persistent)

Arguments	value	Voltage Vref (in Volts)	V1
	persistent	Make value persistent in future connections to board (False by default)	
Returns		Nothing	

This command sets the Vref voltage that calibrates the ratiometric to voltage conversions. Ratiometric values are values expressed as a ratio to a known value:

$$ratio = \frac{voltage}{Vref}$$

As ADCs and DACs use ratiometric values, voltage measurements depend on knowing the ADC and DAC reference voltages. The hardware board reports its Vref reference voltage when it connects. This value, however, could be the nominal value, not the real value. If you measure the reference voltage you can input the obtained value to get more precise measurements.

If you set the optional second argument ***persistent*** to ***True***, the Vref voltage value will be saved on a "Cal_Vdd.dat" file and become persistent. That means that the set Vref value will override the value reported by the board in future connections.
If you want to remove the persistence of the Vref value just delete the "Cal_Vdd.dat" file. Note that this also removes the stored Vdd voltage.

This command does not interact with the hardware board as all board measurements are ratiometric and are always converted to voltage in the SLab Python module. The command is not considered *Auxiliary* because you have to be connected to the board to use this command. This prevents the vdd value, if not persistent, to be overridden by the value provided by the board upon connection.

Example:

```
>>> import slab_nucleo
>>> slab.setVref(3.325,True)
>>> slab.connect()
```

It makes the set Vref value persistent and you will get, upon connection information about the Vref value override.

```
Connected to Nucleo64-F303RE SLab v1.0
ADC Calibration data loaded
DAC Calibration data loaded
Vdd loaded from calibration as 3.329 V
Vref loaded as 3.325 V
```

manualCalibrateDAC1 ()

Arguments	None	V1
Returns	Nothing (Shows calibration curves)	

Manually calibrates DAC 1 and also sets Vdd and Vref values.
In order to perform the calibration you need to use a voltage measurement device to obtain the real voltage output values. The command includes all necessary descriptions to perform the calibration.
The command will ask you to input the readings of the voltage each time is needed.
The calibration curve is stored on a "Cal_DAC.dat" file so it is persistent between python execution sessions. Calibration is performed automatically on all DAC 1 operations if the calibration file is available.
The command also obtains the Vdd and Vref values and they are stored in a "Cal_Vdd.dat" file.
After the calibration ends, a calibration curve for DAC 1 will be shown.

This command has an alias command ***cal1***

Example: Calibrate1.py
Manually calibrate DAC 1

Just issue the command:

```
>>> slab.manualCalibrateDAC1()
```

Or its alias:

```
>>> slab.cal1()
```

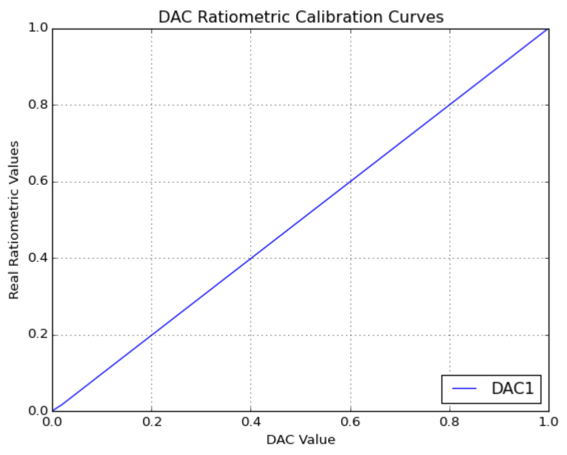
Or run the Python script **Calibrate1.py**. You will get something like:

```
Manual calibration of DAC 1
You will need a voltage measurement instrument (VM)

Put VM between the Vdd terminal and GND
Write down the voltage value and press enter

Voltage value [Volt]:
```

Just write down, followed by return, the voltage at Vdd. An follow the next instructions that require write the value of the DAC 1 output each time it is needed. The last voltage it will ask is the reading for ratiometric 1.0 input, so it corresponds to the reference Vref value.
After the measurements, a ratiometric calibration curve for DAC 1 is shown.



Note that in this case the calibration curve is almost perfect so DAC 1 didn't need any calibration at all and setting the Vref value with the ***setVref*** command could have been enough.

adcCalibrate()

Arguments	None	V1
Returns	Nothing (Shows calibration curves)	

The adcCalibrate command obtains the ratiometric calibration curves of the four ADC channels against 11 values between GND and Vref generated by DAC 1 that we will supposed to have a correct ratiometric response. You can use the ***manualCalibrateDAC1*** command to guarantee that. The calibration curve is shown on screen and is used to correct any future voltage or ratiometric measurement.
The calibration curves are stored on a "Cal_ADC.dat" file so they are persistent between python execution sessions. Calibration is performed automatically on all ADC readings if the calibration file is available.
In order to calibrate the ADCs, you shall connect the input of all four ADC channels ADC1 to ADC4, including any input buffer circuit to the output of DAC 1 before calling this command.

This command has an alias command ***cal2***

Example: Calibrate2.py
Calibrate the ADCs

Just issue the command:

```
>>> slab.adcCalibrate()
```

Or its alias:

```
>>> slab.cal2()
```

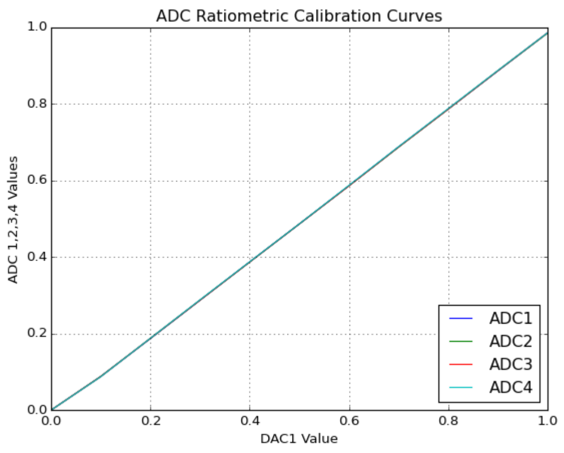
Or run the Python script **Calibrate2.py**. You will get something like:

```
Calibration of ADCs

Conect the DAC 1 output to all ADC inputs
Use the buffers in all connections

Press [Return] to continue
```

After you hit return and the measurements are done, a set of ratiometric calibration curves for the ADCs are shown.



In this case we see that the ADC reading for maximum DAC input of 1.0 is not 1.0. That means that references for DAC and ADC does not match. The calibration curve will take care of this difference.

dacCalibrate()

Arguments	None	V1
Returns	Nothing (Shows calibration curves)	

The dacCalibrate command performs a calibration of the DAC response against the response of the previously calibrated ADC channels.
This function should never be called if the ADCs are not previously calibrated or are not known to have the proper response. You can use the ***adcCalibrate*** command for that.
The dacCalibrate command obtain the ratiometric calibration curves of the DAC channels at 11 values between GND and Vdd against ADC channels with the same number. The calibration curve is shown on screen and is used to correct any future use of the DAC.
The calibration curves are stored on a "Cal_DAC.dat" file so they are persistent between python execution sessions. Calibration is performed automatically on all DAC operations if the calibration file is available.

In order to calibrate the DACs, you shall connect the output of every DAC to the input of the ADC with the same number. DAC 1 to ADC 1, DAC 2 to ADC 2 and, if available, DAC 3 to ADC 3.

This command has an alias command ***cal3***

Example: Calibrate3.py
Calibrate the DACs in a F303RE nucleo board.

Just issue the command:

```
>>> slab.dacCalibrate()
```

Or its alias:

```
>>> slab.cal3()
```

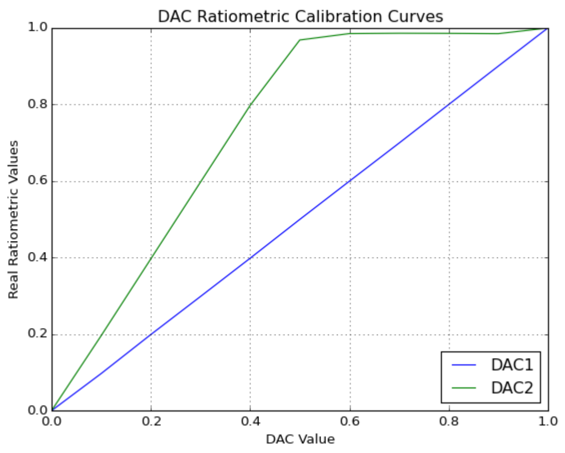
Or run the Python script **Calibrate3.py**. You will get something like:

```
Calibration of DACs

Conect the DAC outputs to ADC inputs with same number
DAC 1 to ADC 1 and DAC2 to ADC2 and son on...

Press [Return] to continue
```

After you hit return and the measurements are done, a set of ratiometric calibration curves for the DACs are shown.



Note that DAC2 has a gain factor of 2 in the ratiometric readings. This is because we used a gain 2 buffer in this channel to prevent turning on the user LED connected in the F303RE nucleo board to this channel output. As this gain is included in the calibration curve, it will automatically be compensated when setting the DAC output using the ***writeDAC*** or ***setVoltage*** commands.

checkCalibration()

Arguments	None	V1
Returns	Nothing (Shows calibration curves)	

This command check the calibration of the board.
The command will ask you to connect each DAC output with the ADC input with the same number. Any other ADC input shall be connected to DAC 1 output.
A set of ADC vs DAC voltage curves will be shown. In a properly calibrated system they shall have all a slope of 1 so y values equal x values.
While the curve is on screen, all DAC outputs will be set to 1.0V so that you can check the reference voltage.
When the command ends after closing the plot, all DAC outputs are set to zero.

This command has an alias command ***cal4***

Example: Calibrate4.py
Check the board calibration.

Just issue the command:

```
>>> slab.ckeckCalibration()
```

Or its alias:

```
>>> slab.cal4()
```

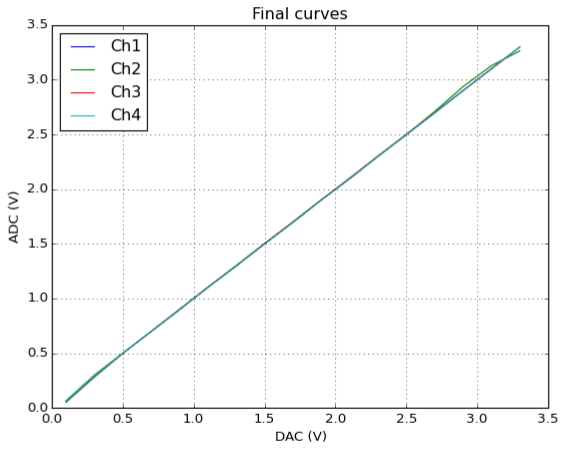
Or run the Python script **Calibrate4.py**. You will get something like:

```
Calibration check

Conect the DAC outputs to ADC inputs with same number
DAC 1 to ADC 1 and DAC2 to ADC2 and son on...
Connect the rest of ADCs to DAC 1

Press [Return] to continue
```

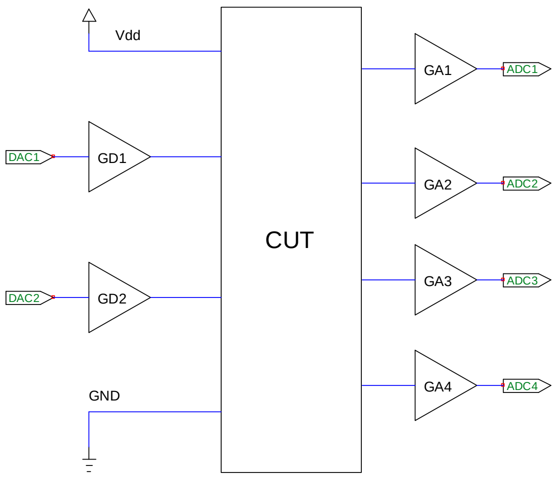
After you hit return and the measurements are done, a set of ADC vs DAC voltage curves are shown.



Note that calibration is not perfect, especially at the higher voltages, but it is quite good.

Calibration procedure

As indicated before, hardware boards usually have to high output impedance on DACs and too low input impedance on ADCs. That means that you need to add buffer circuits to all DACs and ADCs.



In the most basic setup, all buffer circuits feature unity gain so:

$$GD1 = GD2 = GA1 = GA2 = GA3 = GA4 = 1$$

In some cases, however, that would not be the case. For instance, several Nucleo boards connect the user LED to DAC 2 output so this output cannot go over half the supply voltage. In that case it is advisable to use GD2 = 2.

In other cases, the hardware board is limited to low voltage operation, like 3.3V and we need to measure higher voltages. In that case greater than one gain on DACs and lower than one gain on ADCs. For instance, in order to interface a 3.3V nucleo board (with DAC 2 limited to 1.7V) with a 10V circuit we could use:

$$G_0 = \frac{10V}{3.3V} \qquad GD1 = G_0 \quad GD2 = 2G_0 \quad GA1 = GA2 = GA3 = GA4 = \frac{1}{G_0}$$

The purpose of the calibration procedure is to be able to calibrate the board so that measurements are correct by compensating the gain and non linearity of the buffers, ADCs and DACs.

Calibration in the SLab system uses three elements:

- Vref voltage
- Ratiometric DAC curves CalDAC$_i$(x)
- Ratiometric ADC curves CalADC$_i$(x)

For each ADC number **i**, it shall hold:

$$\frac{Mesured\ Voltage}{Vref} = CalADC_i\left(\frac{Real\ Voltage}{Vref}\right)$$

So the calibration curve of the ADCs indicates the ratiometric value we measure on the ADC when we set each possible real ratiometric value at the input of the ADC.

In a similar way, for each DAC number **i**, it shall hold:

$$\frac{Real\ Voltage}{Vref} = CalDAC_i\left(\frac{Set\ Voltage}{Vref}\right)$$

So the calibration curves for the DACs indicates the ratiometric real value of voltages set for each intended voltage we set the DAC to generate.

Vref is the reference voltage of the ratiometric measurements. It usually is similar to the Vdd supply voltage of the board.

All operations on the DACs and ADCs use the three calibration elements so that we always work with the real values. Slab uses Vref and CalADC$_i$(x) to obtain real values for measured ADC values and uses Vref and CalDAC$_i$(x) to set the DAC inputs from the intended real value to set at the DAC outputs.

In order for the calibration system to operate correctly we need to guarantee two conditions:

- That calibration is constant. That is, each time the buffer circuit or the hardware board changes we need to recalibrate. It is also advisable to calibrate from time to time to correct from temporal deviations. Ideally you should calibrate at the start of each measurement session but that is not always practical.

- That calibration curves are monotonous. That is, that you can invert the curves to obtain real voltages from measured voltages in ADCs and set voltages from real voltages in DACs.

As DACs and ADCs are quite linear, SLab system uses calibration curves with only 11 points. Data is linear interpolated between points.

Hardware boards are not guaranteed to contain any reliable voltage reference, so we will need a measurement instrument capable of reading voltages to perform the calibration. A hand held multimeter usually will do. From this point we will call this instrument Voltage Metter (VM).

Calibration is performed in three stages. In the first stage we do a rough calibration of DAC 1 against the VM and also obtain the Vref and Vdd values. In the second stage we calibrate de ADCs against the calibrated DAC 1. Finally, in the third stage, we calibrate all the DACs against the calibrated ADCs. That means that DAC 1 is calibrated two times. A rough calibration in stage 1 and a finer calibration in stage 3.
There is an optional fourth stage to check the calibration.

The calibration can be performed by calling the appropriate commands, but to ease the methodology, three Python scripts are provided, one for each calibration stage.

During all the calibration procedure, all ADCs and DACs shall be connected to their buffer circuits.

Stage 1 : Rough DAC 1 calibration and Vref and Vdd measurement

To perform this first stage just call the ***Calibrate1.py*** script or use the equivalent command after connection:

```
>>> slab.manualCalibrateDAC1()
```

This command asks for the voltage at the Vdd node and then asks for the voltage at the buffered DAC 1 output for several input values. The Vref value is obtained from the maximum DAC output.
Finally, the obtained $\mathrm{CalDAC}_1(x)$ curve is shown.
If the calibration curve is not monotonous, the calibration aborts with an exception.

Stage 2 : Calibration of ADCs

To perform this second stage just call the ***Calibrate2.py*** script or use the equivalent command after connection:

```
>>> slab.adcCalibrate()
```

This command asks to connect DAC 1 output to all ADC inputs. Then it obtains the $CAlADC_i(x)$ curves by sweeping the DAC 1 output.

If any calibration curve is not monotonous, the calibration aborts with an exception.

Stage 3 : Calibration of DACs

To perform this last third stage just call the ***Calibrate3.py*** script or use the equivalent command after connection:

```
>>> slab.dacCalibrate()
```

This command asks to connect DAC 1 output to ADC 1 input and DAC 2 output to ADC 2 input. If there are more DACs, just connect their output to the ADC input with the same number. After that it obtains the $CAlDAC_i(x)$ curves by sweeping the DAC outputs and reading the ADC inputs.

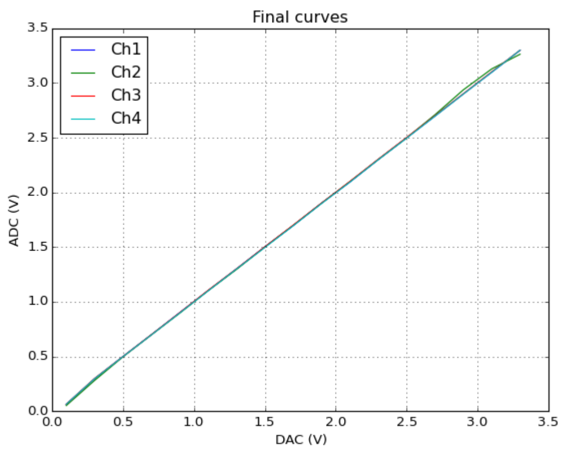
If any calibration curve is not monotonous, the calibration aborts with an exception.

Optional stage 4 : Check the calibration

After the calibration ends you can check it calling the ***Calibrate4.py*** script or using the equivalent command after connection:

```
>>> slab.checkCalibration()
```

This command asks the user to connect the DACs to the ADCs and obtains the DAC to ADC voltage transfer functions that should be straight lines with equal values on both axes as in the following figure:



While the curves are shown, DAC outputs are set to 1.0V so that you can check them using a voltage measurement instrument.

As stage 4 does not perform any calibration, only checks it, you can execute it at any time to check that the calibration has not drifted. If the final curves are not good, a new calibration shall be performed.

Internal Variables

The SLab Python module includes several internal variables that can be accessed to obtain some internal module data. You can access those variables, in the same way than functions, using the module namespace.
The module cannot guarantee that you don't modify the internal variables so that you break its functionality so beware of the recommendations given on any of those variables.

vdd
This is the internal variable that holds the supply voltage of the hardware board. This value can be set in several ways:

- It is obtained from the board using the ***connect*** command
- It is read from a " Vdd_Cal.dat" file if present
- It can be set using the ***setVdd*** command
- It is obtained in the first calibration stage

You should never make direct changes to this variable. You should use the ***setVdd*** command instead.

Example
Use the vdd value in a measurement

We suppose that we have a 1000 Ohm resistor between the vdd node and the node connected to ADC 1, we can calculate the current on the resistor ***ir*** as:

```
>>> ir = (slab.vdd - slab.readVoltage(1)) / 1000
```

vref
This is the internal variable that holds the reference voltage of the hardware board. This value can be set in several ways:

- It is obtained from the board using the ***connect*** command
- It is read from a " Vdd_Cal.dat" file if present
- It can be set using the ***setVref*** command
- It is obtained in the first calibration stage

You should never make direct changes to this variable. You should use the ***setVref*** command instead.

sampleTime

The sample time determines the time between samples on a transient measurement. This value shall always set by the proper functions like ***setSampleTime***. Like in the vdd case, sometimes we want to request the value of current sample time. In that case you can access to this variable to get the data.

Example

Show current sample time

```
>>> print "Sample time is " + str(slab.sampleTime) + " s"
```

linux

Linux and Windows have different serial communication procedures. In order to use the proper method, the operating system is detected when the slab module loads. If Linux is detected, it will usually show with a message.
In the case that you get the auto detect wrong and your Linux machine is not detected, the serial communication can be unable to operate.

Example

Force linux before connecting to a board.

```
>>> import slab
>>> slab.linux = True
>>> slab.connect()
```

References

STM32 Nucleo Page

http://www.st.com/en/evaluation-tools/stm32-mcu-nucleo.html

MBED Developer Page

https://developer.mbed.org/

Python page:

https://www.python.org/

Anaconda page:

https://www.continuum.io/downloads

TinyCad

https://sourceforge.net/projects/tinycad/

Circuit images on this document have been drawn using the free software TinyCad