Labber

USER MANUAL

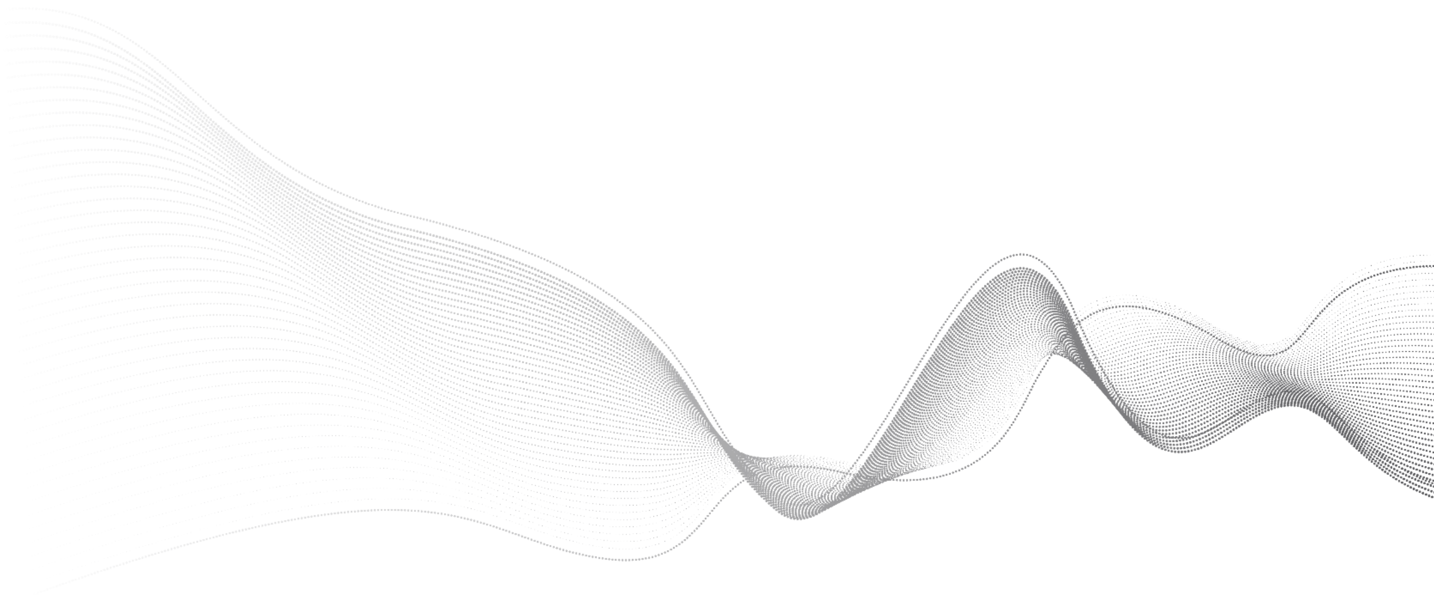


Table of Contents

1. Introduction

The software package consists of three separate programs. The *Instrument Server* handles the communication with the instruments, the *Measurement* program allows instrument values to be controlled and measured in used-defined sequences, while the *Log Browser* is used to organize and analyze the acquired data. The relation between the parts is visualized in Fig. Overview.

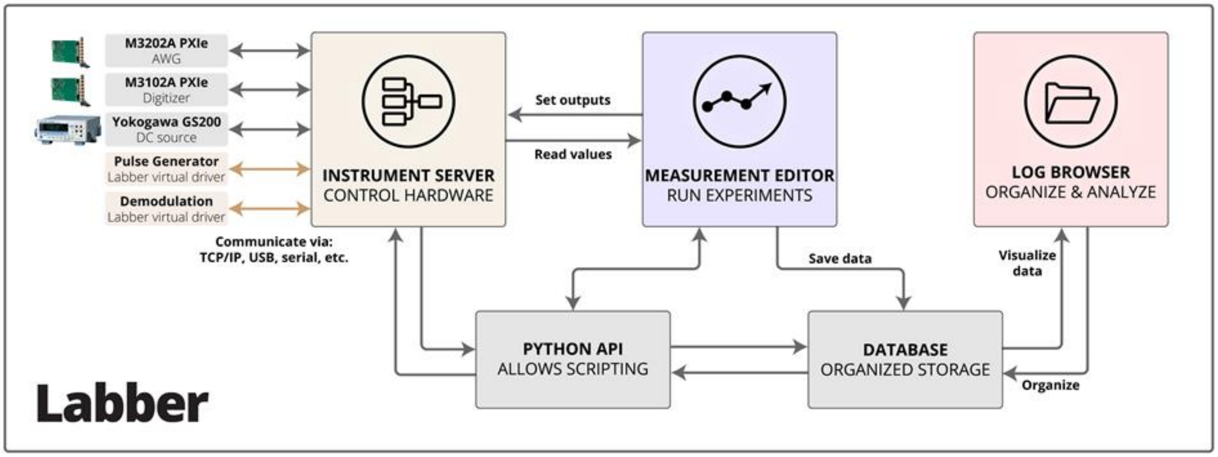


Fig 1.1. Overview and structure of the components in the Labber software package.

In a typical experimental setup, the *Instrument Server* keeps track of and communicates with all the instruments and equipment available in the setup. The communication can be over GPIB, serial, USB, TCPIP, or any other interface. During an experiment, the *Measurement* program will connect to an *Instrument Server* to output values to one specific instrument, or to read data from another one. Note that the *Measurement* program only talks to the *Instrument Server*, and not directly with the instruments. This modular approach allows the same generic procedure to be used for setting/reading values, regardless of the instrument type or the communication interface.

The *Measurement* program saves the experimental configuration, the instrument settings and the acquired data into a central log database. The *Log Browser* provides a fast and efficient method for browsing, visualizing and organizing the measured data. Finally, the *Log Viewer* provides functionality for data analysis and for generating high-quality plots and figures.

In addition to the *Instrument Server*, *Measurement* and *Log Browser* programs, there is a Python API which allows all functionality to be accessed programmatically for scripting purposes or for writing custom applications.

2. Installation

2.1. Installation - Microsoft Windows

The setup file will install *Labber* to the default Microsoft Windows installation directory, as well as create folders for storing data and local driver files in the user's home directory. The default directories for local files can be set in the *Preferences* window, see Section PrefsFolder).

After installation, the *Instrument Server*, *Log Browser* and *Measurement* programs can be started by clicking the corresponding file from the Windows start menu. Note that the *Log Browser* and *Measurement* programs can be opened from within the *Instrument Server*, so it is usually sufficient to start just the server program.

2.1.1. Microsoft Windows - Troubleshooting

Depending on security settings, some virus scanners may prohibit *Labber* from being installed or run on you computer. If you're experiencing difficulties installing or running the program, try to temporarily disable the virus scanner.

Some Microsoft Windows distributions lack a few support files needed by the program to run correctly. If the program won't start, download and install the redistributable support files for Microsoft Visual C++ from `http://www.microsoft.com/en-us/download/details.aspx?id=26368`. Click on *"Download"* and select the file `vcredist_x86.exe`.

2.1.2. Microsoft Windows - Defender SmartScreen warnings

On certain Windows distributions, a dialog may pop up when installing *Labber* stating that the applications is unrecognized and hasn't been screened by Microsoft. To install *Labber*, you need to override the dialog by clicking on *"More info"*, and then click on the *"Run anyway"*-button.

2.2. Installation - Mac OS X

On Macintosh OS X, open up the installer disk image (*.dmg*-file) and drag the *Labber* folder to the *Applications* folder. The *Labber* folder contains the *Instrument Server*, *Log Browser* and *Measurement* applications, as well as driver files and a few example scripts. To start one of the programs, double-click the corresponding app file.

If the program fail to open due to OS X's restrictions to only allow apps from the Apple App store, open the Mac's System Preferences window, select *Security & Privacy*, go to the *"General"* pane and set *"Allow apps downloaded from"* to *"Anywhere"*. If you don't want to lift the restrictions completely, the settings in *Security & Privacy* can be set to *"Mac App Store and identified developers"*. In this case, the user needs to open each of the *Labber* apps once by right-clicking the app icon, holding down the *alt/option* or the *control* key on the keyboard, then clicking open. This will instruct OS X that these apps are allowed to run. The operation needs to be performed once for all three apps, to make sure that the apps will be able to call each other.

2.2.1. MacOS 10.15 Catalina and newer

Starting with MacOS 10.15 Catalina, Apple removed the option to allow installation of apps from unregistered software developers. We are currently in the process of getting approval from Apple for *Labber*, but in the meantime, it is still possible to run *Labber* by overriding the settings for each of the subcomponents used by the software. When opening the *Instrument server* the first time, MacOS will show a dialog informing that the app *"cannot be opened because the developer cannot be verified"*. When the dialog shows up, leave it open and launch the MacOS System Preferences window, select *Security & Privacy*, and go to the *"General"* pane. Next, click *"Cancel"* in the warning dialog. At this point, a text message and a *"Open Anyway"* button will pop up in the *Security & Privacy* window, which can be used to override the warning.

Next, there will be a similar warning for opening *Python*, and a similar technique (click *"Cancel"*, then click *"Open Anyway"* in the Preferences window) can be used to override the warning. Unfortunately, the procedure needs to be repeated for each of the Python packages used by *Labber* (approximately 10-15 in total), and you may need to quit and restart the *Instrument server* a few times to make sure all packages are accepted. Once all packages have been accepted, *Labber* will start as usual when launching the application subsequently.

2.3. Installation - Linux

On Linux, open up the debian package file (*.deb*-file) to install Labber. This will install *Labber* to the folder */usr/share/Labber*, as well as create folders for storing data and local driver files in the user's home directory. The default directories for local files can be set in the *Preferences* window, see Section PrefsFolder).

After installation, the *Instrument Server*, *Log Browser* and *Measurement* programs can be started by opening a terminal and typing `labber-instrumentserver`, `labber-logbrowser` or `labber-measurement` at the prompt. If you are running a desktop manager, there will also be launcher icon available for the three programs. Note that the *Log Browser* and *Measurement* programs can be opened from within the *Instrument Server*, so it is usually sufficient to start just the server program.

2.4. VISA distribution

To communicate with instruments over the VISA protocol, a VISA distribution needs to be installed on the computer. A VISA distribution can be downloaded from National Instruments, see `http://www.ni.com/visa/`.

2.5. Network and firewall settings

The *Instrument Server*, *Log Browser* and *Measurement* programs communicate using TCP/IP, which makes it possible to perform measurements involving instruments connected to different computers, even on different networks. The default settings assign TCP port 9406 for server/client communication and TCP port 9407 for sending internal notifications between the program parts. If you want to perform measurements in a multi-computer network and firewall is enabled on your system, the firewall must be configured to allow traffic on these ports. The port numbers can be changed in case they are occupied on your system (see Chapter Prefs).

2.6. Program folders

In addition to the folders with executables the program uses a few extra folder locations, as listed below.

2.6.1. Data folder

The program needs a folder for saving the measured data. By default, this folder is set to *“<User home directory>/Labber/Data”*, but it can be changed at any time from the *Preferences* window (see Section PrefsFolder).

2.6.2. Instrument drivers

The program has two separate folders for storing instrument drivers, one main folder (called *“Instrument drivers”* in the *Preferences* window) and one local folder (called *“Local drivers”* in the *Preferences*). The main driver folder resides in the same folder location as the executables, and should not be altered in a typical setup. The local driver folder is set to *“<User home directory>/Labber/Drivers”*, but its location can be changed in the *Preferences*.

When creating a new instrument driver, the driver definition file should always be placed in the *“Local drivers”* folder. This allows the user’s own drivers to be kept separately from the drivers provided with *Labber*, and it also prevents drivers written by users from being deleted when updating the *Labber* program to a newer version. See Section Drivers for more information on creating instrument drivers.

2.6.3. Scripting

The Python API that contains scripting helper functions are located in the *Script* folder of the main program directory. See Section scriptPython for more information on scripting.

3. Instrument Server

3.1. Program startup

When starting the *Instrument Server*, the program will create a tray icon and a tray menu for controlling the server, showing preferences and launching the *Measurement* and the *Log Browser* programs (see Fig. 3.1). The tray menu also shows the status of the network server. Whenever the network server is running, clients are allowed to connect to the server to communicate with the instruments. Note that the network server keeps running in the background even after the server window has been closed. To stop the server, either select *"Stop Network Server"* from the tray menu or quit the server by selecting the *"Quit Server"* menu item.

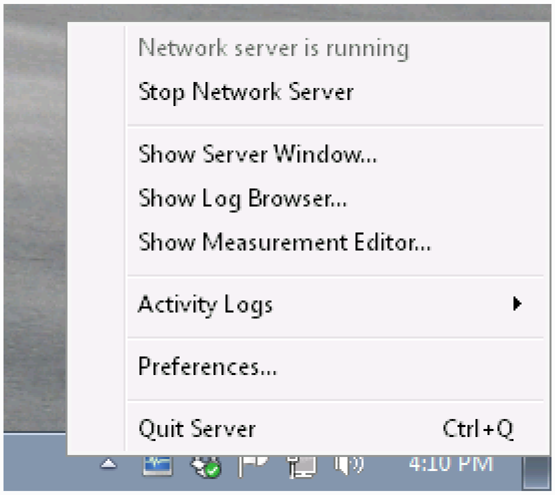
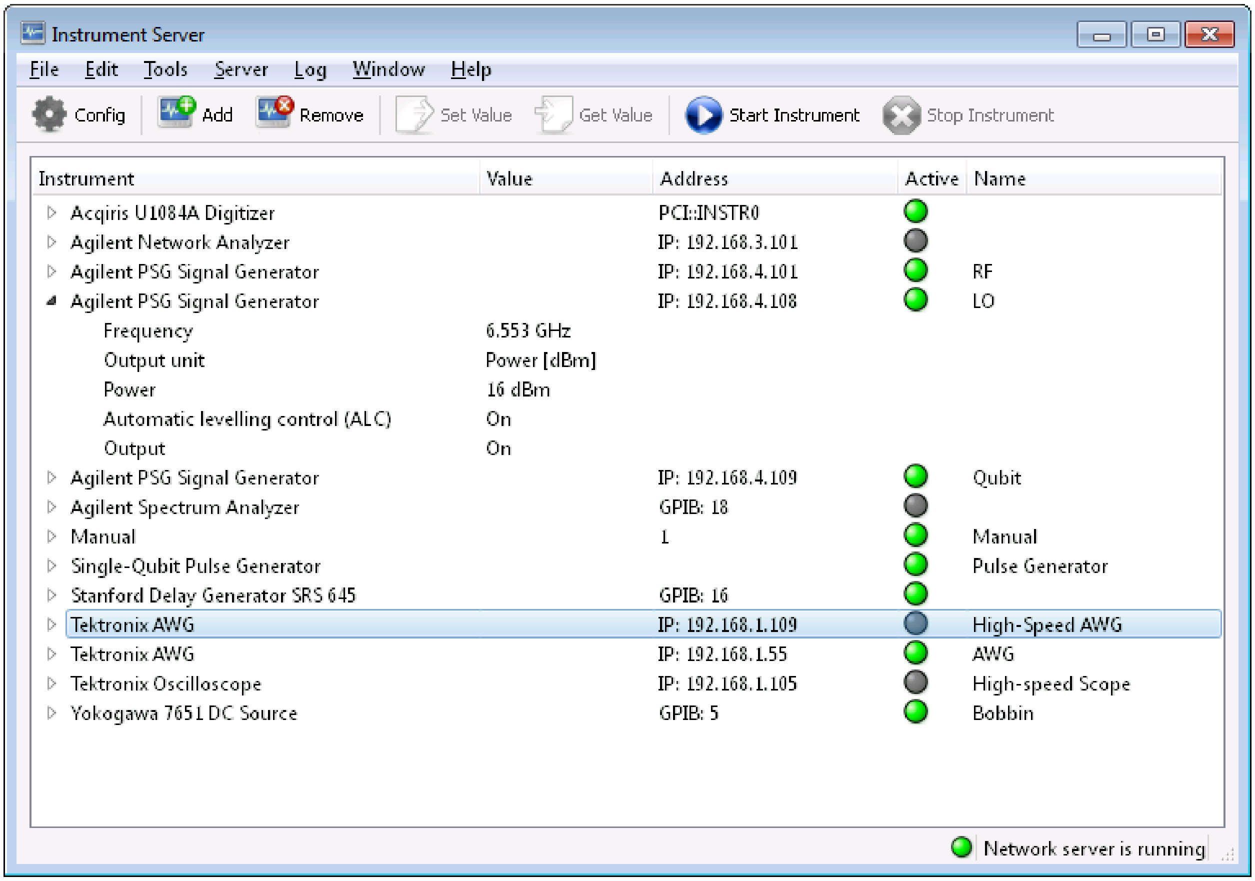


Fig. 3.1 System tray menu for the Instrument Server program. In addition to controlling the server settings, the menu provides options for starting the Measurement and the Log Browser programs.



Instrument	Value	Address	Active	Name
▷ Acqiris U1084A Digitizer		PCI::INSTR0	🟢	
▷ Agilent Network Analyzer		IP: 192.168.3.101	⚫	
▷ Agilent PSG Signal Generator		IP: 192.168.4.101	🟢	RF
◢ Agilent PSG Signal Generator		IP: 192.168.4.108	🟢	LO
Frequency	6.553 GHz			
Output unit	Power [dBm]			
Power	16 dBm			
Automatic levelling control (ALC)	On			
Output	On			
▷ Agilent PSG Signal Generator		IP: 192.168.4.109	🟢	Qubit
▷ Agilent Spectrum Analyzer		GPIB: 18	⚫	
▷ Manual		1	🟢	Manual
▷ Single-Qubit Pulse Generator			🟢	Pulse Generator
▷ Stanford Delay Generator SRS 645		GPIB: 16	🟢	
▷ Tektronix AWG		IP: 192.168.1.109	⚫	High-Speed AWG
▷ Tektronix AWG		IP: 192.168.1.55	🟢	AWG
▷ Tektronix Oscilloscope		IP: 192.168.1.105	⚫	High-speed Scope
▷ Yokogawa 7651 DC Source		GPIB: 5	🟢	Bobbin

🟢 Network server is running

Fig. 3.2 The main Instrument Server window.

3.2. Server window

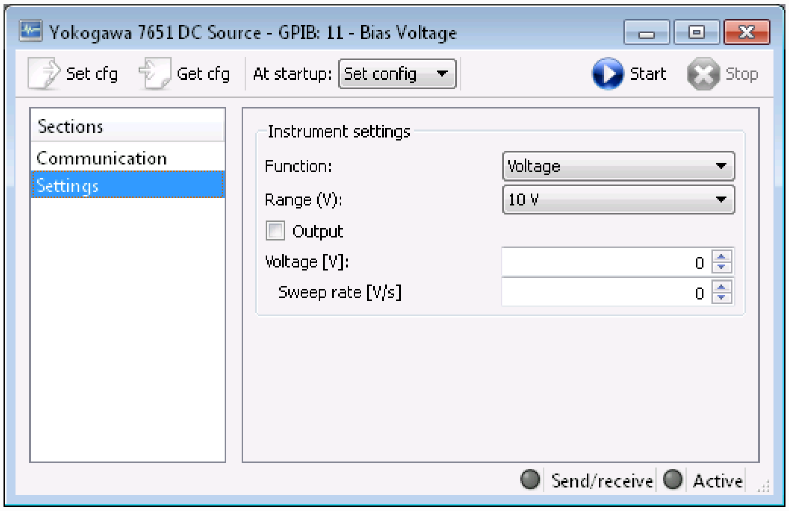
The main server window contains a list with all instruments defined in the setup (see Fig. Server). The standard procedure of *Instrument Server* is to populate this list with the instruments that can be controlled by the computer. Once the instrument have been defined and properly configured, they are ready to be used by the *Measurement* program.

3.3. Adding instruments

To add an instrument, click the *"Add"* button or select *"Edit/Add…"* from the pull-down menu. The program will scan the global and local *Instrument driver*-folders (defined in the preferences, see Section PrefsFolder), and bring up a list with available drivers. Select the instrument type to be added and define the communication interface and address. The instrument can also be given a unique name, which is convenient if many instruments of the same type are present in the setup.

3.4. Configuring instruments

Once an instrumented has been added to the server, it needs to be configured to perform the desired operation. Select the instrument to be configured in the instrument list and click the *"Config"* button (or just double-click the instrument name). This will bring up a window with instrument configuration settings (see Fig. 3.3 for an example of a driver for a DC source). The window contains a list with (at least) two sections with controls:


Yokogawa 7651 DC Source - GPIB: 11 - Bias Voltage

At startup: Set config

Sections: Communication, Settings

Instrument settings

Function: Voltage

Range (V): 10 V

□ Output

Voltage [V]: 0

Sweep rate [V/s] 0

Fig. 3.3 Driver configuration window for a current source. In addition to controls defining the instrument configuration, there are buttons for sending and retrieving the configuration from the hardware. Some quantities, like "Voltage" in the figure, have additional controls for defining sweep rates.

Communication:

This section contains communication controls that define the interface type and address. In addition, if the driver support multiple instrument models with different installed options, the model type and available options will be shown here.

Settings:

This section (and all other sections, if present) contain instrument-specific configuration settings.

The toolbar at the top of the window provide the following buttons and controls for communicating with the hardware:

Set cfg:

Send the configuration defined in the dialog to the instrument hardware. This requires the communication interface and address to be properly defined.

Get cfg:

Read the configuration from the instrument hardware and update the driver dialog.

At startup:

This controls defines the operation to be performed directly after the instrument driver has started. The default is *"Set config"*, which will configure the instrument hardware according to the settings in the driver dialog. Other options are *"Get config"*, which will read the configuration from the instrument hardware and update the *Labber* driver configuration, or *"Do nothing"*, in which case neither the hardware configuration nor the *Labber* driver configuration are updated.

Start:

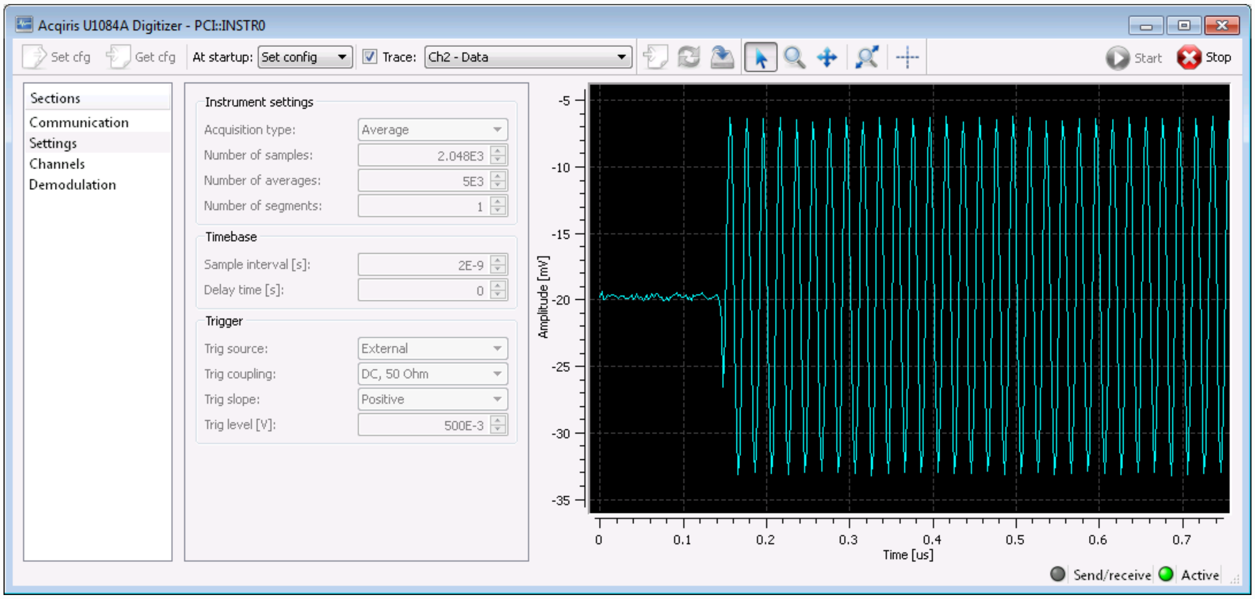
When clicking this button, the *Instrument Server* will connect to the instrument and perform the operation defined by the *"At startup"*-control. After successfully performing these tasks, the instrument will be in the *Active* state (marked by an indicator in the lower-right hand corner of the driver window and in the *Instrument server* window). Note that once the driver is active, any subsequent changes made to any of the controls will directly be sent to the instrument hardware. If an instrument is controlled by a client, it is no longer possible to change the configuration from the driver window (all controls will be grayed out). Values can still be sent to or read from the instrument, but only by using the *"Set Value"* or *"Get Value"* buttons in the server window, or if a client asks a value to be measured/updated. The "grayed out"-behavior can also be turned on by default from the *"Server"* section of the *Preferences* dialog (see Section PrefsServer).

Stop:

This will take the instrument driver out of the *Active* state, stop any eventual instrument operation and close the communication interface.

3.5. Instruments with vector-valued quantities

Some instruments like oscilloscopes, network analyzers and digitizers measure not only scalar values but also traces containing vector values. Drivers for such instruments contain a few extra controls (see Fig. 3.4. for an example). If the *"Show trace"* checkbox is enabled, the user can acquire and plot the current instrument data by selecting a trace to show and clicking the *"Get trace"* button. The *"Save trace..."* button allows the trace currently visible to be saved to the log database. Note that instrument driver must be started and in the active state to acquire and show data traces.



Acqiris U1084A Digitizer - PCI::INSTR0

Set cfg | Get cfg | At startup: Set config | ☑ Trace: Ch2 - Data | Start | Stop

Sections: Communication, Settings, Channels, Demodulation

Instrument settings
Acquisition type: Average
Number of samples: 2.048E3
Number of averages: 5E3
Number of segments: 1

Timebase
Sample interval [s]: 2E-9
Delay time [s]: 0

Trigger
Trig source: External
Trig coupling: DC, 50 Ohm
Trig slope: Positive
Trig level [V]: 500E-3

Fig. 3.4. Example of an instrument driver that returns vector-valued data.

3.6. Keeping track of open client connections

Once all instruments are defined, clients can connect to the server to control and measure instruments quantities. The server-client model of *Labber* is very flexible:

The *Measurement* program can setup experiments that involve instruments connected to different servers on different computers, and a single server can handle simultaneous calls from multiple measurement programs. This flexibility also brings potential complications, like situations where two clients simultaneously try to access the same instrument. To avoid these complications, the server provides a way for clients to exclusively lock an instrument, thereby preventing other clients from accessing it. The locks are described in more detail in Section MeasDriverCfg.

To keep track of open connections and locked instruments, the *Instrument Server* program features a dialog that lists open client connections and the instruments those clients are using. The dialog is shown by selecting *“Server/Show Open Connections…”* from the menu bar.

3.7. Troubleshooting - Timing statistics

The *Instrument Server* program keeps track of the time each instrument driver need to perform operations, which can be useful information when benchmarking instrument communication. To turn on the timing statistics, select *“Tools/Show Timing Statistics”* from the *Instrument Server* menu bar. This will add two columns to the main *Instrument Server* window, one displaying the total number of calls performed to a specific instrument quantity, and one displaying average the time per call. The timing statistics can be reset for all instrument by selecting *“Tools/Reset Statistics”*, or for individual quantities by right-clicking the item and selecting *“Reset Timing Statistics”*.

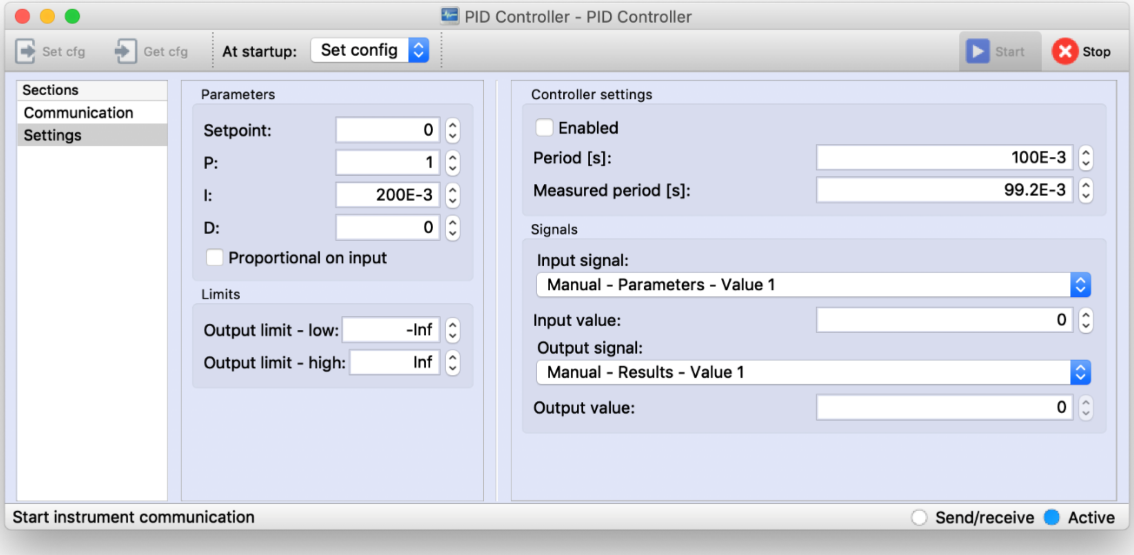
3.8. Troubleshooting - Instrument and Network logs

The *Instrument Server* program keeps logs of recent activities, both for instrument and network communication. The log files are useful if problems arise with instrument communication or if clients have difficulties connecting to the server. To inspect the log files, select *“Log/View Instrument Log”* or *“Log/View Network Log”* from the *Instrument Server* menu bar. The log files provide dated entries with the data strings sent to or received from instruments or from clients.

The amount of logging detail can be controlled in the preferences dialog (see Section PrefsServer); select *“Debug”* for the most detailed information. However, once the problems have been resolved and the instruments and networks are working as expected, it is recommended to reduce the logging detail level to minimize overhead.

4. Controller

In addition to standard instruments, *Labber* also provides special controller instruments for implementing functionality such as PID controller loops. The controller instruments works by reading an input value from a separate instrument such as a thermometer, applying a controller logic to regulate temperature (for example), and then sending the controller output value to another instruments such as a heater.



At startup: Set config

Parameters

Setpoint: 0

P: 1

I: 200E-3

D: 0

☐ Proportional on input

Limits

Output limit - low: -Inf

Output limit - high: Inf

Controller settings

☐ Enabled

Period [s]: 100E-3

Measured period [s]: 99.2E-3

Signals

Input signal: Manual - Parameters - Value 1

Input value: 0

Output signal: Manual - Results - Value 1

Output value: 0

Start instrument communication

○ Send/receive ● Active

Fig. 4.1. The user interface for the PID Controller driver.

4.1. Controller operation

To use the controller functionality, start by adding a controller instrument to the *Instrument server* by clicking the *Add Instrument*-button in the *Instrument server* toolbar. *Labber* provides a built-in PID controller, and additional custom controllers can be created as described in Section ControllerDriver. In addition to the usual sections and settings specific to the particular driver, a *controller* driver also have a number of extra settings related to running the controller loop. An example of the built-in *PID Controller* driver dialog is shown in Fig. 4.1, with the controller settings seen in the right-hand side of the figure. The dialog contains the following settings:

Enabled:

If checked, the controller loop will run in the background and call the input/output instruments at a fixed interval set by the `Period`-setting.

Period:

Intended controller loop period, in seconds.

Measured period:

Actual controller period, which may be different than the set period depending on the time it takes to read/write the input/output values from/to the instruments.

Input signal:

Input signal for the optimizer.

Input value:

Current input value.

Output signal:

Output signal for the optimizer.

Output value:

Current output value.

To set up the controller, first start the instrument driver by clicking the *Start*-button. Next, select the proper *Input* and *Output signals* from the pull-down controls. Finally, set the intended controller period, make sure that both the *Input* and *Output* instruments are running, and then press the *Enabled* checkbox to start the controller. The controller loop will now run in the background and call the input/output instruments at a fixed interval set by the `Period`-setting.

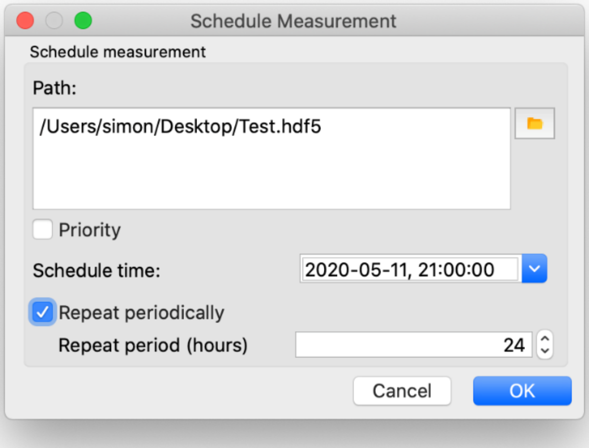
4.2. Improving controller performance

If the controller needs to run at a high repetition rate, set the `Period` control to `0.0` to run the controller loop continuously without gaps. The actual controller loop period will not be zero due to the time it takes to read/write values from the instruments.

Note that the updating the user interface introduces a slightly delay, so for the fastest operation it is advised to run the controller with its dialog window closed. The measured controller loop period can be probed even if the controller window is closed by expanding the `PID Controller/Controller settings` items in the main *Instrument server* dialog.

5. Scheduler

The *Labber Instrument server* contains a scheduler that allows the user to define a queue of experiments to run, as well as functionality for repeating a specific measurement at fixed intervals, for example once per day. The scheduler automatically launches and executes the measurement program whenever an experiment is due.



Schedule Measurement

Schedule measurement

Path: /Users/simon/Desktop/Test.hdf5

□ Priority

Schedule time: 2020-05-11, 21:00:00

☑ Repeat periodically

Repeat period (hours): 24

Fig. 5.1. The user interface for scheduling a measurement.

5.1. Scheduling measurements

Measurements can be scheduled from the user interface, or from the Python API. To schedule an experiment from the user interface, open the *Instrument server* program and select *"Scheduler/Schedule Measurement"* from the main pull-down menu. This will open a dialog (see Fig. 5.1) with the following settings:

Path:

Path to *Labber* measurement configuration to run,
in `.labber`, `.json` or `.hdf5` format.

Priority:

Checkbox for setting priority in scheduling system. If a prioritized and non-prioritized measurement are both ready for execution at a specific time, the prioritized one will run first.

Schedule time:

Scheduled time for measurement to run. If the date is in the past, the measurement will execute as soon as the dialog is closed.

Repeat periodically:

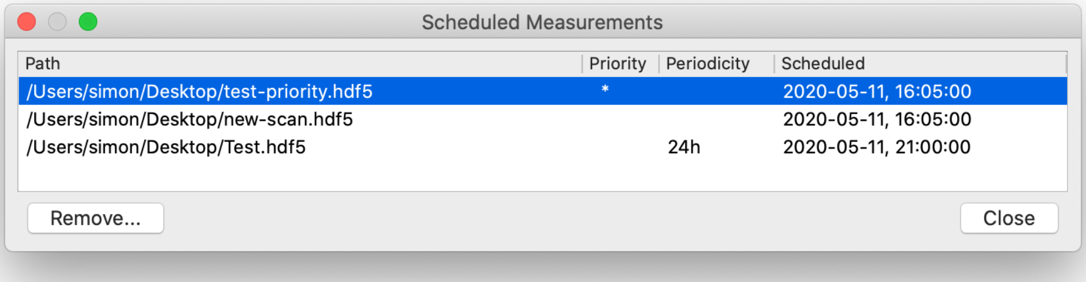
If checked, the experiment will be repeated at a fixed interval. If unchecked, the measurement will run only once.

Repeat period:

Repeat interval, in hours.

When closing the dialog, the measurement configuration will be added to the queue of experiments to execute. If there are no other experiments in the queue, and if the *"Schedule time"* is right now or in the past, the measurement will start as soon as the dialog is closed.

To view a list of scheduled measurements from the user interface, select *"Scheduler/Schedule Measurement"* from the *Instrument server* pull-down menu. In addition to displaying the measurements currently scheduled in the queue, the dialog has an option to remove a scheduled measurements (Fig. 5.2).



Path	Priority	Periodicity	Scheduled
/Users/simon/Desktop/test-priority.hdf5	*		2020-05-11, 16:05:00
/Users/simon/Desktop/new-scan.hdf5			2020-05-11, 16:05:00
/Users/simon/Desktop/Test.hdf5		24h	2020-05-11, 21:00:00

Fig. 5.2. The user interface for displaying the list of scheduled measurements.

5.2. Scheduler settings

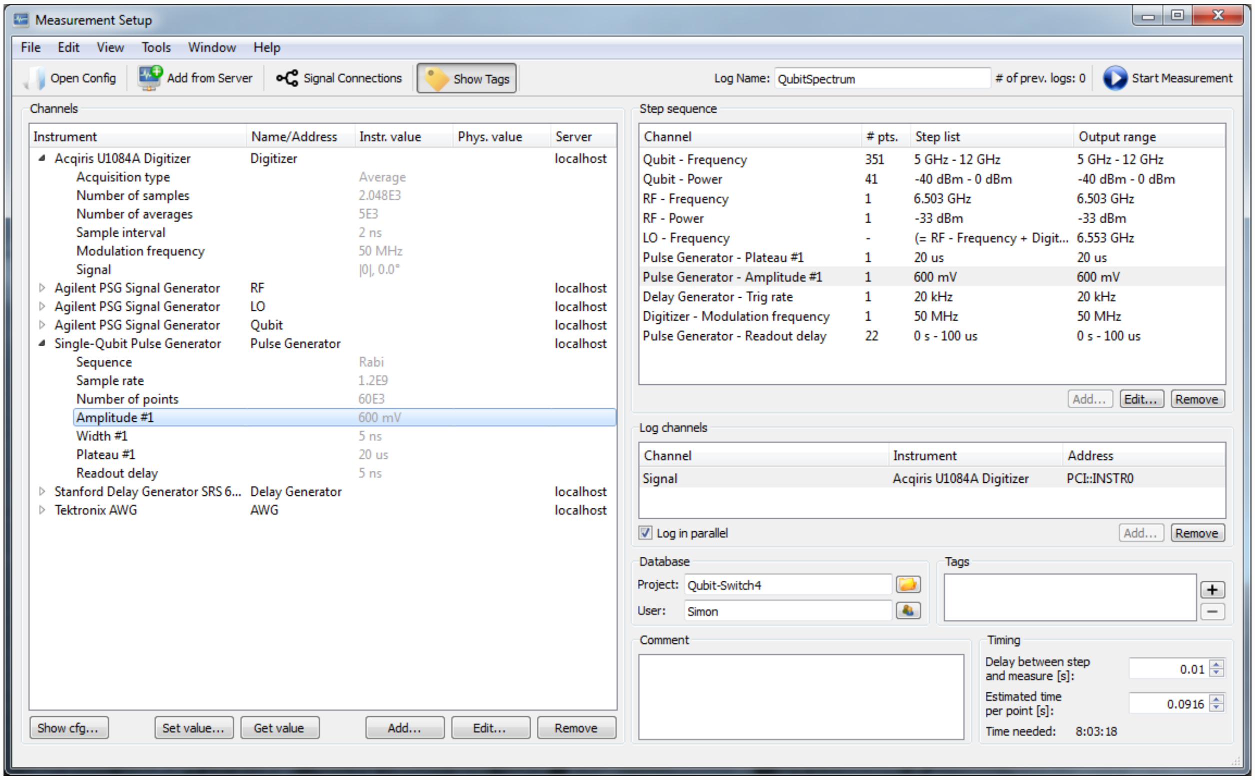
By default, the scheduled experiments will run in a separate process from the one used by the main *Measurement* program. This allows a queued experiment to execute at the same time as one launched from the *Measurement* program. However, this may cause issues if both experiments are trying to access the same resource, for example a specific instrument in the *Instrument server*.

This can be avoided by unchecking the setting *“Run queued experiments in separate process”* under the section *“Measurement/Advanced”* in the *Labber* preferences. If unchecked, an experiment started from the *Measurement* user interface will not start immediately upon pressing *Start* in the dialog, but rather be added to the scheduler queue and execute when the other experiments in the queue have finished.

Note that a restart of both the *Instrument server* and the *Measurement* program may be required for the changes to fully go into effect.

6. Measurement program

The *Measurement* program allows instrument quantities to be measured as a function of other parameters. The program is highly flexible, allowing multi-dimensional sweeps involving any instrument quantity defined in the *Instrument Server*. The *Measurement* is started by from the system tray menu (*"Show Measurement Editor"*) or by selecting *"Window/Show Measurement Editor"* from the main *Instrument Server* window.



Log Name: QubitSpectrum # of prev. logs: 0

Channels

Instrument	Name/Address	Instr. value	Phys. value	Server
Acqiris U1084A Digitizer	Digitizer			localhost
Acquisition type		Average		
Number of samples		2.048E3		
Number of averages		5E3		
Sample interval		2 ns		
Modulation frequency		50 MHz		
Signal		|0|, 0.0°		
Agilent PSG Signal Generator	RF			localhost
Agilent PSG Signal Generator	LO			localhost
Agilent PSG Signal Generator	Qubit			localhost
Single-Qubit Pulse Generator	Pulse Generator			localhost
Sequence		Rabi		
Sample rate		1.2E9		
Number of points		60E3		
Amplitude #1		600 mV		
Width #1		5 ns		
Plateau #1		20 us		
Readout delay		5 ns		
Stanford Delay Generator SRS 6...	Delay Generator			localhost
Tektronix AWG	AWG			localhost

Step sequence

Channel	# pts.	Step list	Output range
Qubit - Frequency	351	5 GHz - 12 GHz	5 GHz - 12 GHz
Qubit - Power	41	-40 dBm - 0 dBm	-40 dBm - 0 dBm
RF - Frequency	1	6.503 GHz	6.503 GHz
RF - Power	1	-33 dBm	-33 dBm
LO - Frequency	-	(= RF - Frequency + Digit...	6.553 GHz
Pulse Generator - Plateau #1	1	20 us	20 us
Pulse Generator - Amplitude #1	1	600 mV	600 mV
Delay Generator - Trig rate	1	20 kHz	20 kHz
Digitizer - Modulation frequency	1	50 MHz	50 MHz
Pulse Generator - Readout delay	22	0 s - 100 us	0 s - 100 us

Log channels

Channel	Instrument	Address
Signal	Acqiris U1084A Digitizer	PCI::INSTR0

☑ Log in parallel

Project: Qubit-Switch4

User: Simon

Delay between step and measure [s]: 0.01

Estimated time per point [s]: 0.0916

Time needed: 8:03:18

Fig. 6.1. The main Measurement configuration window.

6.1. Measurement configuration

The main measurement configuration window is shown in Fig. 6.1. The left-hand panel contains a list with instrument quantities (or *Channels*) involved in the experiment, the top-right section defines the sequence of *Channels* to sweep, while the lower-right panel shows a list of channels to measure. Measurements are easily configured by dragging *Channels* between the lists.

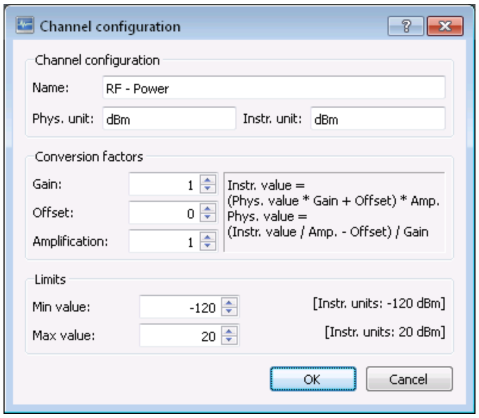
6.2. Adding channels

The first step for setting up a measurement is to define the *Channels* involved in the experiment. A *Channel* represents an instrument quantity on an *Instrument Server*, together with additional properties like name, unit, conversion factors and limits. The easiest way to define channels is to add instruments already present on an *Instrument Server*, which is done by clicking *"Add Instruments from Server"* in the main *Measurement* configuration window. This will bring up a dialog with options for connecting to an *Instrument Server*, and a list of instruments that can be added to the measurement.

There is also an option for adding channels without having the corresponding instrument previously defined on a server. Choosing the *"Edit/Add Instruments…"* from the menu bar will bring up a dialog where the user can select which instrument to use and how to communicate with it. In this case, the user needs to specify both the communication protocol of the instrument as well as the server address, so that the new instrument can be created on the *Instrument Server* when starting the measurement.

By default, the program will add *channels* for every quantity active in the instrument configuration. To minimize clutter and allowing an easy overview of the measurement setup, it's advisable to remove channels that will not be controlled from an experiment. This is done by selecting a quantity and pressing the *"Remove"*-button below the list. Quantities can always be re-introduced later by clicking the *"Add"*-button. In addition, the value of any instrument quantity can by controlled by opening the *Instrument driver* configuration window (either by double-clicking the instrument name in the *Channels* list or by selecting an instrument and pressing *"Show cfg…"*).

Note that the *Instrument driver* configuration window serves different purposes in the *Measurement* program and in the *Instrument Server*. In the *Instrument Server*, the instrument configuration dialog is used to *directly* control the hardware settings, meaning that any changes to the dialog will directly affect the state of the hardware. In contrast, in the *Measurement* program the dialog is used to define a configuration that will be used in a specific *Measurement*, but no changes are made to the hardware until the measurement is started. To avoid confusion, *Instrument driver* configuration windows have a different background color when opened within the *Measurement* program and in the *Instrument Server*.



Channel configuration

Channel configuration

Name: RF - Power

Phys. unit: dBm Instr. unit: dBm

Conversion factors

Gain: 1

Offset: 0

Amplification: 1

Instr. value =
(Phys. value * Gain + Offset) * Amp.
Phys. value =
(Instr. value / Amp. - Offset) / Gain

Limits

Min value: -120 [Instr. units: -120 dBm]

Max value: 20 [Instr. units: 20 dBm]

Fig. 6.2. The Channel configuration window allows the user to define properties of the physical quantity measured by an instrument.

Channels also contain properties for describing the physical quantity measured by an instrument. The properties are set in the *Channel configuration* window (see Fig. Channels), which is brought up by selecting a channel and clicking the *“Edit…”*-button. The dialog allows the user to set channel max/min limits, and to define conversion factors between the physical quantity investigated in the experiment and the quantity measured by the instrument. An example where such a conversion is useful is when current biasing a circuit by applying a voltage over a large resistor in series with the circuit. In this case, the physical quantity would be current (with units Ampere), while the instrument quantity would be voltage. The equations for converting between physical and instrument units are defined in the text box next to the *Conversion factors*-controls.

6.2.1. Instrument configuration - locks

After the channels have been added to the *Measurement* configuration, the corresponding *Instrument driver* configuration window can be shown by double-clicking the instrument name or selecting the instrument and clicking *“Show Config…”*. In addition to the settings listed when describing the *Instrument Server* (see Section ConfigInstr), the dialog contain contains an extra checkbox (*“Lock instrument on server”*, under the section *“Communication”*) for determining whether the instrument will be used exclusively by the current experiment. If the control is checked (default behavior), no other clients can connect to or change the instrument values during the duration of the measurement. See Section OpenClients for more information about locks.

6.3. Sending and retrieving values from instruments

The *"Set Value…"*- and *"Get Value"*-buttons below the channel list allow the user to quickly set or retrieve the current instrument value. Note that the *"Set value…"*-operation will immediately send the new value to the instrument hardware.

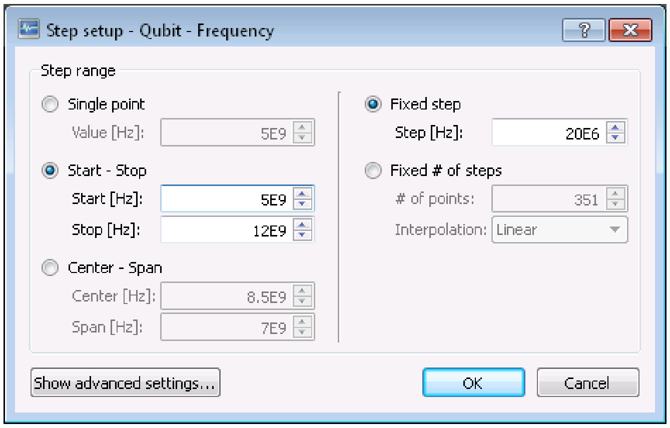
Instrument values can also be controlled from the *Instrument driver* configuration window, which is opened by double-clicking the instrument name or selecting the instrument and clicking *"Show Config…"*. However, in contrast to the *"Set/Get Value"*-buttons, changing the value of a control in the *Instrument driver* window will only update the local value kept in the *Labber* configuration. The actual instrument hardware is not updated until the user clicks *"Set Cfg"* in the driver window, or when the measurement is started (provided that the *"At Measurement Start"*-option in the driver window is set to *"Set config"*).

6.4. Defining step sequences

A measurement consists of a list of *Step sequences* that output values to instruments in a specified order. To define a *Step sequence*, drag the channel to sweep from the *Channels* list on the left to the *Step sequence* list on the top right of the main *Measurement* configuration window. This will bring up the *Basic settings*-dialog for defining the range of values to output. Once defined, the step items can be re-ordered by dragging the entries within the list.

6.4.1. Step setup - Basic settings

The basic settings dialog allows the user to define single-point step values or basic ranges, either by defining start-stop or center-span values. The step size is specified either by setting a fixed step size, or by defining the total number of points in the step range, see Fig. 6.3. When defining the number of points in the range, the user can set the interpolation to be linear or logarithmic. Note that logarithmic interpolation only works if all values in the step range are positive.


Step setup - Qubit - Frequency

Step range

○ Single point
Value [Hz]: 5E9

● Start - Stop
Start [Hz]: 5E9
Stop [Hz]: 12E9

○ Center - Span
Center [Hz]: 8.5E9
Span [Hz]: 7E9

● Fixed step
Step [Hz]: 20E6

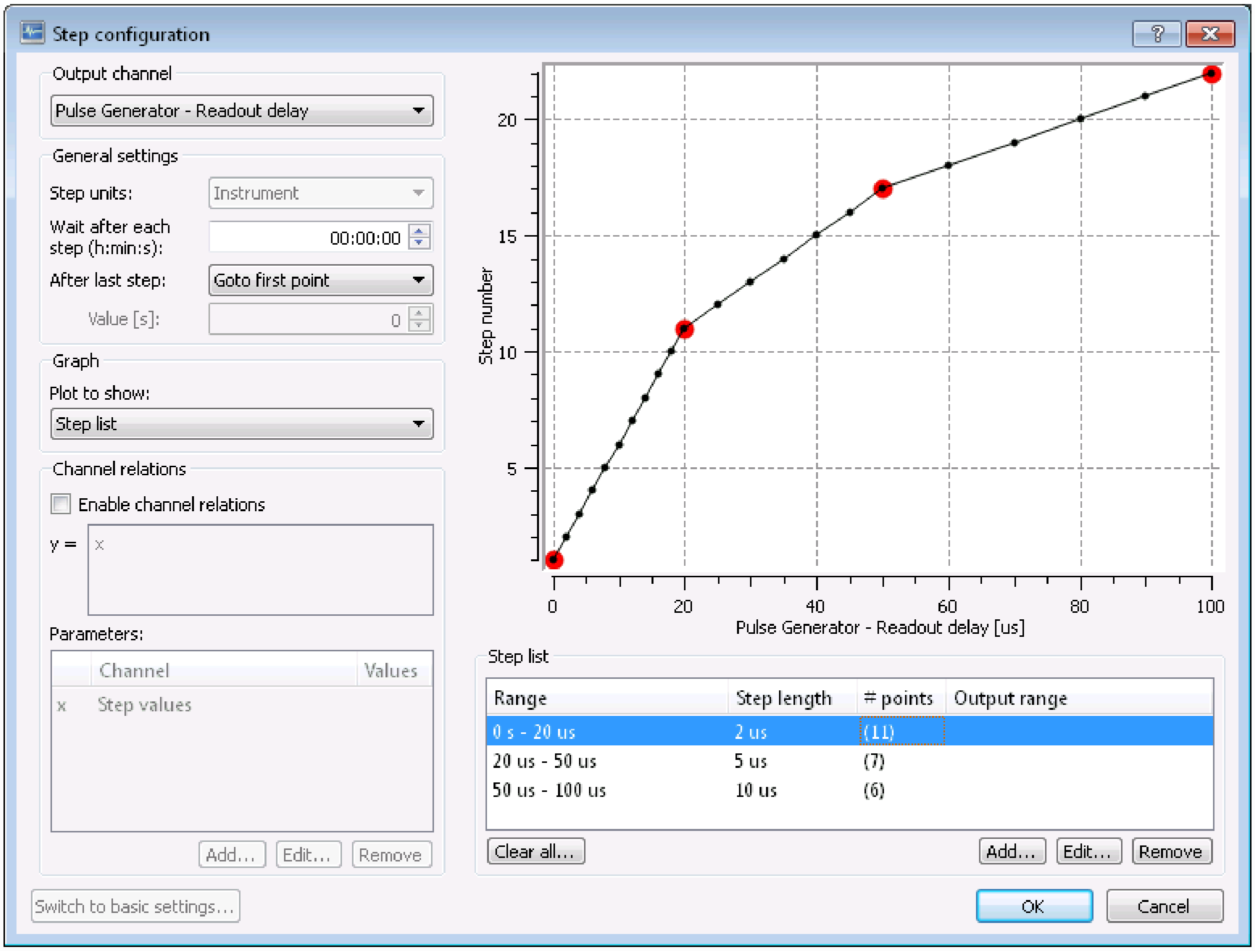
○ Fixed # of steps
# of points: 351
Interpolation: Linear

Fig. 6.3. Basic dialog for defining step sequences.

If the channel is sweepable or if there are scaling factors defined between physical/instrument units, the dialog contains a few extra controls as described in Section AdvancedStepSetup.

6.4.2. Step setup - Advanced settings

The advanced settings dialog contains a few extra controls to provide better control of the step parameters. First, there is a list with step ranges, allowing multiple ranges to be defined with different step sizes (see Fig. 6.3 for an example). Use the *"Add..."*, *"Edit..."* and *"Remove"*-buttons to add/edit ranges, and drag the entries in the list to make the values appear in the right order. The graph in the upper-right corner of the dialog shows a visual representation of the step output values, with the step number on the y-axis and the corresponding output value on the x-axis.



Output channel: Pulse Generator - Readout delay

Step units: Instrument

Wait after each step (h:min:s): 00:00:00

After last step: Goto first point

Value [s]: 0

Plot to show: Step list

☐ Enable channel relations

y = x

Range	Step length	# points	Output range
0 s - 20 us	2 us	(11)	
20 us - 50 us	5 us	(7)	
50 us - 100 us	10 us	(6)	

Fig. 6.4. Advanced dialog for defining step sequences.

In addition, the upper-left part of the dialog contain the following controls for fine-tuning the step sequence:

Step units:

The control sets whether the step values are given in instrument or physical units of the channel (see Section Channels). The step values are updated to reflect the unit settings whenever the control is updated. The control is only visible if physical/instrument unit conversion factors have been defined in the *Channel* setup dialog.

Wait after each step:

Time to wait after a step value has changed. Note that the actual time to wait will be the maximum of this time and the delay time between step and measure as set in the main *Measurement* configuration window (see Section Timing).

Alternate step direction:

If checked, the execution of the step sequence in multi-dimensional experiments will alternate between forward and reversed direction, eliminating the need to go back to the first step point between loops. This feature is also useful when looking for hysteresis when sweeping a field up and down.

After last step:

Defines the operation to perform after the step sequence has completed. Possible values are *"Goto first point"*, *"Stay at final"* or *"Goto specified value"*. Default behavior is *"Goto first point"*.

6.4.3. Step setup - Sweep mode

If the step channel is sweepable, the sweep mode controls provide a few extra user interface elements for controlling the sweep settings. For more information on how to write drivers that supports sweeping, see Section SweepDriver.

Sweep mode:

The program supports three different sweep modes:

Sweep mode - Off:

Sweep mode is off, step values are set directly.

Sweep mode - Between points:

The instrument is swept between step points, but the output is held constant while acquiring data for the log channels.

Sweep mode - Continuous:

In this mode, the instrument is configured to continuously sweep from the first to the last value in the step list. The log channels are being measured at the points defined in the step list, but the program will not stop sweeping the output channel while acquiring data for the log channels.

Rate:

Sweep rate, in units s^{-1}. The sweep rate is also shown in units min^{-1}. Note that the sweep rate in this dialog will overwrite the sweep rate defined in the configuration window of the corresponding instrument driver.

Time between points:

If sweep mode is *Continuous*, this text shows the typical interval between measurement points for the given sweep rate and step list.

Use different sweep rate outside loops:

If checked, a numerical control will appear below the checkbox, allowing the user to define a separate sweep rate for sweeping to the init/final values and for sweeping between loops. If unchecked, the program will use the common sweep rate defined in the control above when setting init/final/between loop values.

6.4.4. Step setup - Channel relations

One useful feature of the *Advanced step configuration* is the ability to define relations between channels. For instance, imagine a situation where we want to sweep two voltages *V1* and *V2* in a way that *V2* is always exactly 1.5 V higher than *V1*. To implement this, we first define the step sequence for *V1* as usual. Next, we create a step configuration for *V2* and switch to the advanced settings. Clicking the *“Enable channel relations”* will allow us to enter an equation relating the output of *V2* to other channels. The values of other channels are accessible through parameters, shown in the list on the right-hand side of the dialog. The *“Add...”*, *“Edit...”* and *“Remove”*-buttons below the parameter list are used to edit the parameter names. The parameter `"x"` refers to the step values as defined in the step list in the upper part of the dialog.

For this particular example, we would enter `p1 + 1.5` in the equation box for channel *V2* (assuming that the parameter `p1` is linked to channel *V1*). The equation string can involve basic mathematical functions like `cos(x)`, `sin(x)`, `sqrt(x)`, `exp(x)`, etc... Also, note that the raised operator (‸) is given by two multiplication signs (`**`).

Before starting a measurement, it is good practice to check that the relation equation produces the intended output. By default, the graph in the top-right corner of the dialog shows the step values generated by the step list, but it can also be configured to visualize the output of the relations as a function of any other channel in the measurement. The graph contents are set by the *“Plot to show”*-control to the left of the figure.

6.5. Log channels

The *Log channels* list defines the channels to measure at each step point. To add log channels, simply drag a channel entry from the main *Channels* list on the left to the *Log channels* list. Note that it is not possible to add the same channel to both the step and the log list.

If the checkbox *“Log in parallel”* is checked, the program will try to measure all channels simultaneously at each step point. If the *“Log in parallel”* is unchecked, the channels will instead be measured sequentially, starting with the top-most one in the list. The log channels can be reordered by dragging the items within the list.

6.5.1. Log channels limits

Each log channel has an associated range limit, which is defined by double-clicking the log channel or clicking the *“Edit…”* button below the log channel list. If the measured value falls outside the defined limits during a measurement, one of the following actions will be taken:

Nothing

No action is taken, the measurement continues as usual.

Continue to next step item

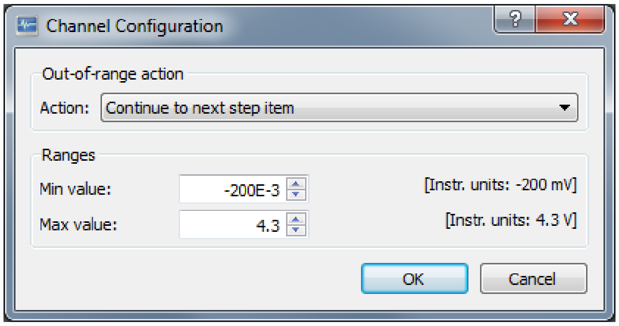
The measurement program stops execution of the innermost step sequence, and continues to the next item of the second step sequence.

Stop, stay at current values

Stop the measurement, hold all instruments at the current values.

Stop, go to init/final configuration

Stop the measurement, go to final values as defined in the *Advanced step setup dialog* (see Section AdvancedStepSetup). The default is to go to the initial values.



Out-of-range action

Action: Continue to next step item

Ranges

Min value: -200E-3 [Instr. units: -200 mV]

Max value: 4.3 [Instr. units: 4.3 V]

Fig. 6.5. Limit options for log channels.

6.6. Timing

The *Timing* section in the lower-right corner of the main *Measurement* configuration window allows the user to set a time to wait between outputting new values to the step channels and measuring the log channels. In addition, the section gives an estimate for how long time it'll take to run the measurement. The estimate is based on the duration of the previous experiment; the expected time needed per point can be adjusted manually, if required.

6.7. Log name, Project and User tags, Comments

When running a measurement, both the measurement configuration and the obtained data are saved into a single file in the database folder (see Section LogDatabase for a discussion of the folder hierarchy and the structure of the log database). The log file name is defined by the *Log name* control in the toolbar in the upper-right part of the main *Measurement* configuration window. When starting a measurement with a file name that already exists, a dialog will pop up presenting the user with the following options:

Create New

The new measurement will be save into a new log file, with a modified file name ("*_2*", "*_3*", "*_4*", etc, will be appended to the log name).

Append Data

The new measurement will append new data to the old log. This option is only available if the measurement is one-dimensional (that is, if only one of the step sequences contains more than a single value), and if the previously existing log has the same structure as the new one.

Overwrite

The old log is deleted before starting the new measurement.

The comment field allows experiment-specific descriptions to be added to the measurement configuration file. Note that there is no need to type any information related to instrument settings here; all the instrument configurations are automatically saved into the configuration file.

6.8. Tags

The *Project*, *User* and *Tags* controls provide ways of keeping the log database organized. The controls are shown by clicking the *"Show Tags"*-button in the dialog toolbar.
The *Project* field supports a hierarchy structure, with subprojects separated by a forward slash ("/"). For example, entering `"Sample2/DeviceA/IV-curves"` will put the log in the subproject `"IV-curves"` of subproject `"DeviceA"`, which is located in the project `"Sample2"`.
The *Project* tag can be enetred directly into the text field, or by clicking the folder icon next to the control to bring up a hierarchy tree with all projects defined in the database.

The *User* name can be entered directly into the text field, or by clicking the user icon to bring up a list with users already present in the log database. A log file can only belong to a single *Project* and *User*.

Contrary to the *Project* and *User* fields, a log can contain multiple *Tags*. The *Tags* are added/removed by clicking the plus/minus signs next to the tag list. Similar to the *Project* field, the *Tags* support a hierarchy tree of tags and subtags.

6.9. Starting a measurement

Once the step sequences, log channels and the log name have been defined, the measurement is ready for execution. When clicking the *"Start measurement"*-button in the upper-right corner of the *Measurement* dialog, the program will perform the following sequence:

1. The program will go through all step sequences to make sure that all of the step values are valid and within the min/max ranges allowed by the corresponding channels.
2. Next, connections will be established to the *Instruments Servers* of all instruments in the *Channels* list.

3. For every instrument, the program will either set or read the current the hardware configuration, depending on the value of the *"At Measurement Start"*-control of each instrument driver (see Section AtStartup). The recommended setting is *"Set cfg"*, since this will assure that instrument is hardware in the same state every time the measurement is performed. Note that the *"At Measurement Start"*-operation will be performed for all instruments defined in the measurement window, even for channels that aren't used in step sequences or as log channels.
4. The program will perform the measurement by stepping through all the values defined in the *Step sequences*. The order of the step sequences defines the order in which the values are outputted, starting with the values in the top-most step sequence. For example, consider a situation with two defined step sequences: one for *"Channel 1"* with values {1,2,3} and one for *"Channel 2"* with values {10,20}. There are a total of 3*2 = 6 step points. If *"Channel 1"* occurs before *"Channel 2"* in the sequence list, the program will set the values in the following order:

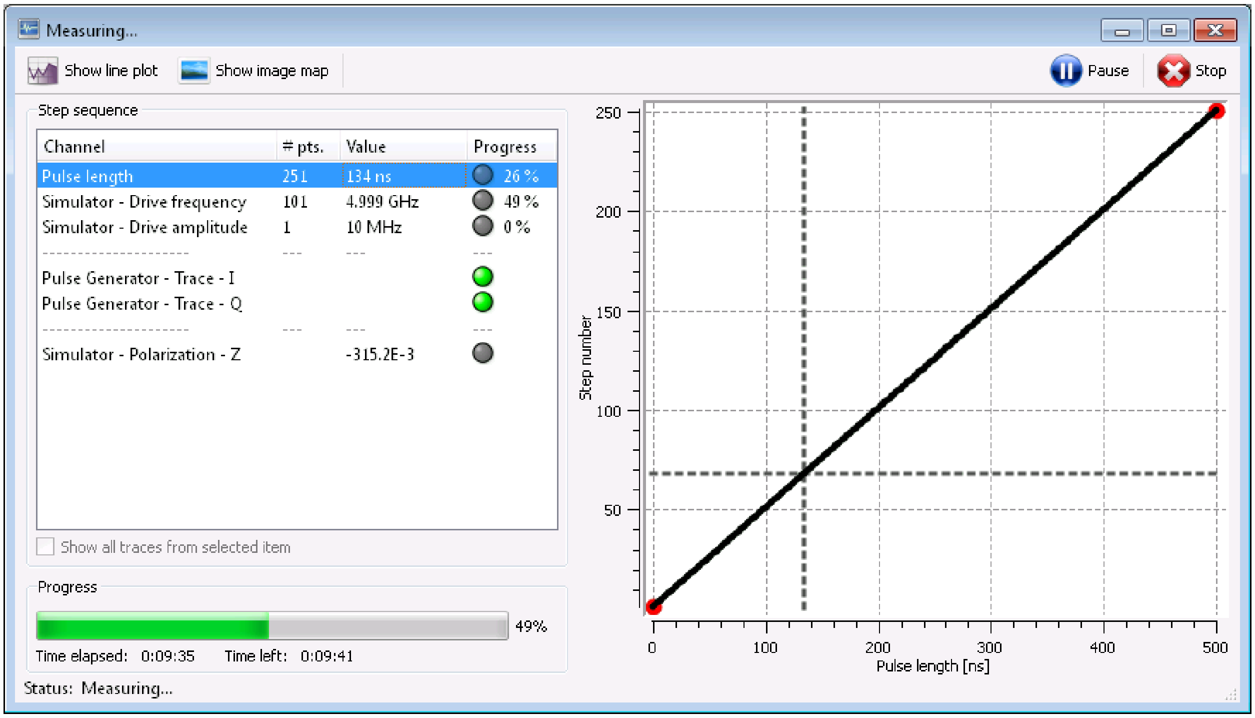
Step No.	Channel 1	Channel 2
1	1	10
2	2	10
3	3	10
4	1	20
5	2	20
6	3	20

5. On the other hand, if *"Channel 2"* occurs before *"Channel 1"*, the step order will be:

Step No.	Channel 1	Channel 2
1	1	10
2	1	20
3	2	10
4	2	20
5	3	10
6	3	20

6. The values of all log channels are measured at each step point.
7. When the measurement is finished, the program will close the connections to all instruments and all *Instrument Servers* and return to the main *Measurement* configuration window. The new log will be available for viewing in the *Log Browser* window (see Chapter BrowserDlg)

Figure 6.6 depicts the dialog shown when a measurement is running. The list on the left contains a list of the step and log channels defined in the measurement, together with current values and progress indicators (for step channels). The green light indicate that a value is currently being sent/received from an instrument. The graph on the right visualize the measurement progress for the step channel selected in the channel list on the left. Alternatively, if a log channel is selected, the graph will show the currently measured trace for that channel.



Channel	# pts.	Value	Progress
Pulse length	251	134 ns	● 26 %
Simulator - Drive frequency	101	4.999 GHz	● 49 %
Simulator - Drive amplitude	1	10 MHz	● 0 %
--------------------	---	---	---
Pulse Generator - Trace - I			●
Pulse Generator - Trace - Q			●
--------------------	---	---	---
Simulator - Polarization - Z		-315.2E-3	●

□ Show all traces from selected item

Progress

49%

Time elapsed: 0:09:35 Time left: 0:09:41

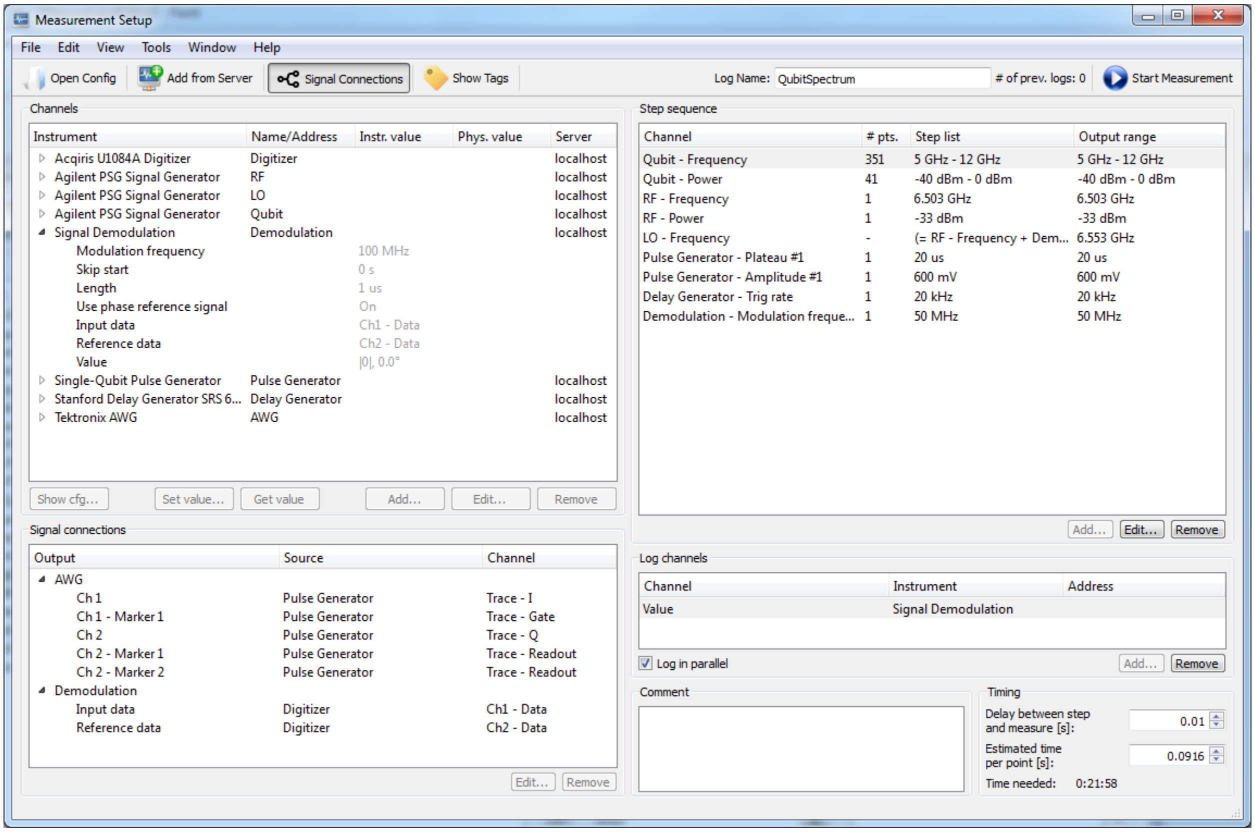
Status: Measuring...

Fig. 6.6. The Measurement window.

The tool bar at the top of the dialog contains buttons for showing the measured data in real-time, either as a line plot or as an image map. In addition, there are buttons for skipping traces, pausing and stopping the experiment. The *Skip* button will stop execution of the innermost step sequence, save the current trace, and then continue to the next step item.

6.10. Signal connections

Many experiments involve sending or reading waveforms from instruments like arbitrary waveform generators, digitizers or digital oscilloscopes. For example, imagine an experiment where we want to control the amplitude of a sine signal outputted using an arbitrary waveform generator. The process can be divided into two tasks: The first task is to numerically calculate a waveform with the correct amplitude, the second task is to send that waveform to the output of the arbitrary waveform generator. Another example would be to measure a waveform with an oscilloscope, and then apply some function to extract the signal's amplitude or frequency, which would allow us to record only a single or a few values characterizing the signal instead of saving the whole waveform to the log file.



Log Name: QubitSpectrum

# of prev. logs: 0

Channels

Instrument	Name/Address	Instr. value	Phys. value	Server
Acqiris U1084A Digitizer	Digitizer			localhost
Agilent PSG Signal Generator	RF			localhost
Agilent PSG Signal Generator	LO			localhost
Agilent PSG Signal Generator	Qubit			localhost
Signal Demodulation	Demodulation			localhost
Modulation frequency		100 MHz		
Skip start		0 s		
Length		1 us		
Use phase reference signal		On		
Input data		Ch1 - Data		
Reference data		Ch2 - Data		
Value		|0|, 0.0°		
Single-Qubit Pulse Generator	Pulse Generator			localhost
Stanford Delay Generator SRS 6...	Delay Generator			localhost
Tektronix AWG	AWG			localhost

Signal connections

Output	Source	Channel
AWG		
Ch 1	Pulse Generator	Trace - I
Ch 1 - Marker 1	Pulse Generator	Trace - Gate
Ch 2	Pulse Generator	Trace - Q
Ch 2 - Marker 1	Pulse Generator	Trace - Readout
Ch 2 - Marker 2	Pulse Generator	Trace - Readout
Demodulation		
Input data	Digitizer	Ch1 - Data
Reference data	Digitizer	Ch2 - Data

Step sequence

Channel	# pts.	Step list	Output range
Qubit - Frequency	351	5 GHz - 12 GHz	5 GHz - 12 GHz
Qubit - Power	41	-40 dBm - 0 dBm	-40 dBm - 0 dBm
RF - Frequency	1	6.503 GHz	6.503 GHz
RF - Power	1	-33 dBm	-33 dBm
LO - Frequency	-	(= RF - Frequency + Dem...	6.553 GHz
Pulse Generator - Plateau #1	1	20 us	20 us
Pulse Generator - Amplitude #1	1	600 mV	600 mV
Delay Generator - Trig rate	1	20 kHz	20 kHz
Demodulation - Modulation freque...	1	50 MHz	50 MHz

Log channels

Channel	Instrument	Address
Value	Signal Demodulation	

☑ Log in parallel

Comment

Timing

Delay between step and measure [s]: 0.01

Estimated time per point [s]: 0.0916

Time needed: 0:21:58

Fig. 6.7. A Measurement configuration with signal connections for pulse generation and signal demodulation.

The *Measurement* program provides *Signal Connections* for managing situations like this. The idea is to separate the signal generation or signal analyzing from the instrument communication, to make it possible to develop generic *Signal Generator* or *Signal Analyzer* drivers for creating or analyzing waveforms, which operate independently of the specific hardware that is used to output or measure the waveforms. In this way, the waveform generation/analyzing functions can be used interchangeably with instruments from different vendors. For more information on how to create your own *Signal Generator* or *Signal Analyzer* drivers, see Section DriverINI.

The *Signal Connections* button in the toolbar of the main *Measurement* configuration window is used to show/hide a list of signal connections defined in the current setup. The button is only enabled when the setup contains instruments that allow waveform generation/analyzing. The *Signal Connections* control contains a list with all the instrument quantities that can output or analyze a waveform. To make a connection, double-click one of the outputs and select the source signal from the dialog that pops up, or simply drag a channel that represents a signal source from the main *Channels* list onto the correct output in the *Signal Connections* list. Figure 6.7 shows an example of a *Measurement* configuration with a few signal connections.

Note that signal connections are also possible for scalar-valued channels. To make a signal connection between two scalar-valued channels, click the *"Show scalar-valued signals"*-checkbox below the signal connection list. Scalar-valued channels are listed in italics in the signal connection list, to distinguish them from the waveform signal connections.

As mentioned earlier in this section, two types of signal connections can be made: The first type is when a *Signal Generator* driver is used to generate waveforms that will be sent to the output of an arbitrary waveform generator, for example. The second type of connection is when a waveform that is acquired using an instrument such as a digitizer or an oscilloscope is sent to a *Signal Analyzer* driver. The example in Fig. 6.7 illustrates both examples: The signals generated by the "Pulse Generator" *Signal Generator* driver are configured to be sent to various output channels of a Tektronix Arbitrary Waveform Generator (labelled "AWG" in the figure), whereas the waveforms acquired by the "Acqiris U1084A Digitizer" will be sent to the of the "Signal demodulation" *Signal Analyzer* driver that will extract the amplitude of the waveform at a specific frequency.

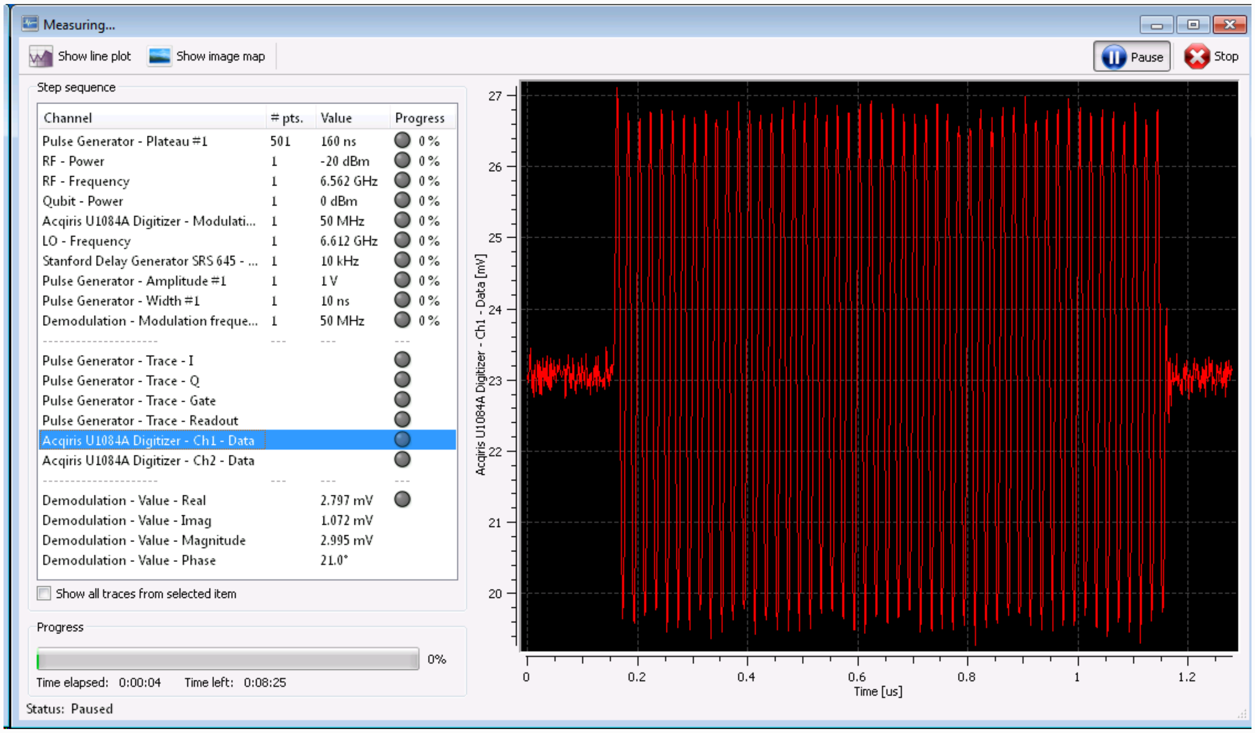
When running a *Measurement* that contains *Signal connections*, for each step in the step sequence the program will perform the following sequence:

1. Update step values: Update the values of channels defined in the *Step sequence*.
2. Generate and output signals: If present, calculate signals with *Signal Generator* drivers and send the resulting waveforms to the corresponding instruments outputs.
3. Wait: Wait for the time specified in the *Timing* section in the lower-right corner of the main *Measurement* configuration window.
4. Acquire and analyze signals: If present, measure instrument channels that acquire waveforms, and send the acquired waveforms to the corresponding *Signal Analyzer* drivers.
5. Log results: Measure the channels specified in the *Log channels* list and save them to disk. If a *Signal Analyzer* driver is in use, the *Log channels* list is where the user defines what quantities to store in the log file.

In the example of Fig. Signals, the step sequence will update a few parameters of the "Pulse Generator" *Signal Generator* driver. Once all parameters have been updated, the

“Pulse Generator” driver will calculate new waveforms that will be sent to the “AWG” output channels. After that, the program will wait for 0.1 second to give the sample time to settle, before acquiring two waveforms (“Ch1 - Data” and “Ch2 - Data”) with the “Acqiris U1840A Digitizer”. The measured will waveforms will be sent to the “Signal demodulation” *Signal Analyzer* driver, which will analyze the waveforms and return the result in the channel named “Value”, which will be stored in the log file.

When running an experiment that contains *Signal connections*, it is possible to look at the measured waveforms in real-time as they are being acquired and processed. In the window that is visible when an experiment is running, mark the channel to investigate in the step sequence list in the left-hand part of the dialog (see Fig. 6.8).



Channel	# pts.	Value	Progress
Pulse Generator - Plateau #1	501	160 ns	● 0 %
RF - Power	1	-20 dBm	● 0 %
RF - Frequency	1	6.562 GHz	● 0 %
Qubit - Power	1	0 dBm	● 0 %
Acqiris U1084A Digitizer - Modulati...	1	50 MHz	● 0 %
LO - Frequency	1	6.612 GHz	● 0 %
Stanford Delay Generator SRS 645 - ...	1	10 kHz	● 0 %
Pulse Generator - Amplitude #1	1	1 V	● 0 %
Pulse Generator - Width #1	1	10 ns	● 0 %
Demodulation - Modulation freque...	1	50 MHz	● 0 %
-------------------	---	---	---
Pulse Generator - Trace - I			●
Pulse Generator - Trace - Q			●
Pulse Generator - Trace - Gate			●
Pulse Generator - Trace - Readout			●
Acqiris U1084A Digitizer - Ch1 - Data			●
Acqiris U1084A Digitizer - Ch2 - Data			●
-------------------	---	---	---
Demodulation - Value - Real		2.797 mV	●
Demodulation - Value - Imag		1.072 mV	
Demodulation - Value - Magnitude		2.995 mV	
Demodulation - Value - Phase		21.0°	

□ Show all traces from selected item

Progress

0%

Time elapsed: 0:00:04 Time left: 0:08:25

Status: Paused

Fig. 6.8. The measurement window during an experiment with signal connections, showing a measured waveform.

6.11. Hardware timing and synchronization

In standard operation mode, *Labber* handles instrument synchronization by waiting for an instrument to report that all step channels values have been outputted before reading the log channels. Since this requires communicating with the instruments over the computer, the time synchronization may not be precise enough for certain applications. For such applications, *Labber* supports operating in arm/trig mode and hardware looping for enhancing the synchronization and timing precision.

6.11.1. Arm/trig mode

In Arm/trig mode, log instrument will be armed to wait for an external trigger before starting to acquire data. To turn on arm/trig mode, click the *Arm/trig mode* checkbox below the Step sequence configuration list in the *Measurement* setup window. The trigger channel must be an instrument channel represented by boolean or a button, and it is defined by the pull-down menu next to the check box. The instrument used to generate the trigger must also be represented in the step configuration list.

In Arm/trig mode, the following operations are performed at each point of the measurement sequence:

1. Set output values of the step channels, but instruct the instrument to wait for a trigger before outputting any signals.
2. Arm the log channels to get ready to acquire data.
3. Wait for the time specified in the *Timing* section in the lower-right corner of the main *Measurement* configuration window. This time can be zero.
4. Generate the trigger signal. The output of the trigger should be physically connected to the step and log instruments, so that the step instrument can start outputting signals and the log instruments can start acquiring data.
5. Read out the acquired data.

6.11.2. Hardware looping

Some instruments can perform looping of values within the instrument hardware. This allows for implementing more efficient looping, since there will be no need for the computer to send new values to the instrument at each step value. This mode requires that the instrument outputting values support hardware looping, that the instrument reading values supports both hardware looping and hardware arming, and that there is a trigger defined for instrument synchronization. If the instruments used in the *Measurement* configuration fulfill these requirements, hardware looping can be activated by clicking the *Hardware loop* checkbox next to the arm/trig mode trigger controls.

In hardware loop mode, the top-most step item in the *Step sequence* configuration list of the *Measurement* setup window will be controlled by the instrument hardware. Instead of setting and getting values point-by-point using the computer, the looping of the top-most step item will be handled in the following way:

1. Calculate the number of values n to step in the top-most step item.
2. Send all n values of the top-most step item to the output instrument, but instruct the instrument to wait for a trigger before outputting any signals.
3. Arm the log channels to get ready to acquire data. The log instruments will be configured to acquire n values.
4. Wait for the time specified in the *Timing* section in the lower-right corner of the main *Measurement* configuration window. This time can be zero.
5. Generate the trigger. The output of the trigger signal should be physically connected to the step and log instruments, so that the step instrument can start outputting signals and the log instruments can start acquiring data. The step instrument will output the n values in the prescribed order, and the log instrument will acquire n values.
6. Read out the acquired n values.

Since multiple values will be read out at once in hardware looping mode, the progress bar and the time estimate shown in the Measurement window during experiments may not update often enough be accurate.

6.12. File locks

The software uses a file locking system to prevent the *Log Browser*, *Log Viewer* or *Measurement Editor* from making changes to a file while a measurement is running. The locking mechanism works by creating an empty file with the same name as the log file that is being measured, but with the ending *.lock*. Under normal operation, the *.lock*-file will be removed when the measurement is completed, but if the experiment was unexpectedly interrupted (for example, if the computer suddenly lost power), the *.lock*-file will persist and prevent any changes from being made to the file. To manually remove the locking, go to the folder where the log file is stored, locate the *.lock*-file and delete it.

6.13. Measurement settings

In addition to the general preferences described in Section Prefs, there are a number of settings that can be uniquely defined for each specific *Measurement*. These specific *Measurement settings* can be accessed by clicking the *"Show Settings"* toolbar icon in the top-left corner of the *Measurement* dialog. The settings are described in detail below.

6.13.1. General

Send values in parallel:

When outputting multiple values in a measurement step sequence, define if data should be sent to all instruments in parallel, or sequentially one after each other. Default value is *True*.

Only send signal if source instrument has been updated:

If checked, Labber will only perform a signal connection if the source instrument has been updated since last call.

Data compression:

The value ranges from 0 (no compression) to 9 (maximum compression). Higher compression reduces the log file size, but may slightly increase time for loading/saving data.

6.13.2. Optimizer

The optimizer functionality and the corresponding settings are described in more detail in Section Optimizer below.

6.14. Comparing Measurement configurations

For complex measurement scenarios containing a large number of instruments, is it sometimes difficult to keep track of all setting and parameters involved in the experiment. For these cases, *Labber* provides a convenient feature to compare and highlight differences between the current scenario and a previous measurement saved in the Log database. The function is accessed by selecting *"Tools/Compare Configurations"* in the pull-down menu and selecting the measurement configuration to compare the current scenario to.

7. Optimizer

In *Optimizer* mode, the *Measurement* program will try to minimize the value of an expression based on the measured channels instead of looping through the step channels in a pre-determined sequence. This can be useful when the goal of the experiment is to minimize a certain quantity as opposed to mapping out the value of the quantity over the full parameter space.

7.1. Optimizer operation

To enable the optimizer, simply double-click on one of the *Step items* in the *Step sequence* list in the *Measurement* program, switch to *"Basic settings"* (if not already in that mode), then click the *"Optimize..."*-button to convert the step item to an optimizer parameter. Instead of sweeping over the parameter, *Labber* will try to optimize the cost function (see below) by varying the parameter over the range specified by the *"Min value"* and *"Max value"* controls in the dialog. The various options in the dialog are described in more detail in Section ParameterSettings below.

7.1.1. Cost function

The next step is to define the cost function and the general settings of the optimizer. These options are available by clicking the *"Show Settings"* toolbar icon in the top-left corner of the *Measurement* dialog, and clicking *Optimizer* in the section list in the left part of the dialog. The most import setting is the optimizer cost function, which is defined by the expression in the *"Minimization function"*-control. The cost function takes the latest measured values of the log channels as inputs and must return a single scalar value. The optimizer algorithm will then try to minimize the value of the cost function by iteratively varying the various optimizer parameters.

The inputs available to the cost function are the latest values of the measured log channels, provided in the numpy list `"y"`. Each element in the list corresponds to a channel, and the order of the elements is the same as the order at which the log channels appear in the *Measurement Editor*. If you are using a single log channel, its value can be accessed by `y[0]`. However, note that `y[0]` may be scalar or vector-valued, depending on if the particular log channel returns a trace of a single value. For the optimizer to work, the cost function must always return a scalar, so if your log channel is vector-valued you need to apply some operation to convert the vector to a scalar. For example, `mean(y[0])` would optimize with respect to the mean of the measured trace. In addition to `y`, the vector `x` with the latest values of the optimizer parameters is also available as an input to the minimization function. You can use any Python and numpy expression when defining the cost function.

7.1.2. Termination and convergence criteria

There are three possible criteria for defining when the optimizer should terminate the optimization process.

Absolute target reached:

> If the value of the cost function is less than the *Target value*, the optimizer will terminate.

Relative tolerance reached:

> If the change in the cost function between calls is smaller than the value given by *"Relative tolerance"*-setting, *AND* if the change in the optimizer parameter values between calls are smaller than the *"Precision"* setting of each parameter, the optimizer will terminate. Note that both criteria need to be fulfilled for termination.

Max number of evaluations reached:

> The optimizer will automatically terminate after performing the number of measurements specified by *"Max evaluations"*.

By default, the *"Target value"* is set to minus infinity, which means that it will never terminate the optimizer. In addition, the *"Relative tolerance"* is set to infinity by default, which means that only the *"Precision"* of the individual optimizer parameters matter for relative convergence.

Note that the termination/convergence criteria may differ for different optimizer algorithms, the description above only refers to the default *Nelder-Mead* optimizer provided by *Labber*.

7.1.3. Running an optimizer measurement

When running a measurement with the optimizer enabled, *Labber* automatically will add a step item named *"Optimizer iteration"* that handles the optimizer loop. Note that it is possible to run an experiment with a mix of optimized and non-optimized parameters, where the optimizer will execute to find the optimal value of one parameter while stepping over different values of another parameter.

7.2. Optimizer settings

In order to use the optimizer, both the general optimization protocol and the individual optimization parameters must be configured. The various settings are described below.

7.2.1. General optimizer settings

These settings define the cost function and the algorithm-specific settings of the optimizer, and can be accessed by clicking the *"Show Settings"* toolbar icon in the top-left corner of the *Measurement* dialog. The settings are described in detail below.

Method:

Algorithm used for optimization.

Max evaluations:

Maximum number of function evaluations/measurements performed before terminating the optimization.

Minimization function:

Function for optimizer to minimize. The measured channels are available in the variable `"y"`, which is a list of log channel values. Each list item may be a number or a numpy array, depending on the channel datatype. Default is `min(y[0])`, which will minimize the value of the first log channel.

Target value:

Absolute value of minimization function at which the optimization will terminate. Default value is `-inf`, which will prevent the optimizer to terminate until the other optimization goals are met.

Relative tolerance:

Change in minimization function between iterations that is acceptable for convergence. Default value is `inf`, which will make the optimizer run until the `Precision`-value of all involved parameters are met.

7.2.2. Individual parameter settings

These settings are individual to each optimization parameter and can be accessed by double-clicking a channel in the *Step sequence* list and going to *"Optimize..."*-mode.

Start value:

Initial value for parameter.

Initial step size:

Initial step size for the parameter.

Min value:

Lowest parameter value allowed during the optimization procedure.

Max value:

Highest parameter value allowed during the optimization procedure.

Precision:

Target precision for optimizer that will trigger optimizer termination.

7.3. Custom optimizers

It is possible to create custom optimizer modules to implement a specific optimization protocol. The sections below describe how to define and test a custom optimizer algorithm.

7.3.1. Defining custom optimizers

It is recommended to use one of the already present optimizer configuration files as a template. The custom optimizers should be contained in a single python `.py` file, which must contain a function called `optimize` that takes exactly two parameters:

config:

Python dict with optimizer settings. The keys have the same names as the labels of the optimizer settings in the *Measurement* program. The individual parameter settings are stored as a list in the same dictionary, with key `optimizer_channels`.

minimize_function:

Python callable that takes exactly one argument (`x`). The function will run the Labber measurement for the provided parameter values `x`, where each value in the vector `x` corresponds to an optimizer parameter. The function is typically passed directly to the `scipy` optimizer, see the provided optimizer `Nelder-Mead` for an example.

The function must return a Python dictionary with results from optimizer, using `scipy`'s `OptimizeResult` format. The only necessary key is "`x`", containing the final optimizer parameters.

When creating a new optimizer, the python file should be given a unique named and placed in the *local* optimizer folder (the folder named *"Local optimizers"* in the *Preferences* window), instead of the global one (*"Optimizer functions"* in *Preferences*). This allows the user's own optimizers to be kept separately from the optimizers provided by *Labber*, and it also prevents optimizers written by the user from being deleted when updating the *Labber* program to a newer version.

Note that even when making additions/changes to an existing optimizers from the global folder, the best practice is to copy that optimizer file from the global folder to the local folder, and only make changes to the optimizer version. If optimizers with the same names exist in both the local and the global optimizer folders, *Labber* will always use the optimizer in the *"Local optimizer"*-folder.

7.3.2. Defining optimizers settings

For custom optimizers, it is possible to define optimizer-specific configuration parameters in addition to the general settings in Section OptimizerSettings above. The optimizer-specific settings are defined by adding a function `define_optimizer_settings()` to the same python `.py` file that contain the optimizer code. The function should return a list of python dicts, where each dict represents a specific setting. The settings are defined in a similar way to quantities of an instrument driver (see Section Quantities), with the difference that the settings are specified in a python function instead of a `.ini` configuration file. Each setting must define the `name` and `datatype` parameter, all other parameters are optional.

The customs settings will show up in the *Optimizer*-section of the *Settings*-pane of the Labber *Measurement* dialog, allowing the user to change their values prior to running a measurement. The values of the custom parameters will then be accessible as entries in the `config` input in the `optimize` function defined above.

As an example, the code below will define custom settings with three parameters for the *Bayesian-Gaussian-Process* optimizer.

```python
def define_optimizer_settings():
    """Define extra settings for optimizer

    Returns
    -------
    optimizer_cfg : list of dict
        List of configuration items for optimizer, each item is a dict.
        Necessary keys are "name" and "datatype".

    """
    # Bayesian optimization settings
    optimizer_cfg = [
        dict(name='Acquisition function',
             datatype='COMBO',
             combo_defs=['LCB', 'EI', 'PI', 'gp_hedge'],
             def_value='gp_hedge',
             tooltip=('See https://scikit-optimize.github.io/ for more info'),
             ),
        dict(name='kappa',
             datatype='DOUBLE',
             def_value=1.96,
             state_item='Acquisition function',
             state_values=['LCB', 'gp_hedge'],
             tooltip=('Controls how much of the variance in the predicted ' +
                      'values should be taken into account. Higher value ' +
                      'favours exploration over exploitation and vice versa'),
             ),
        dict(name='xi',
             datatype='DOUBLE',
             def_value=0.1,
             state_item='Acquisition function',
             state_values=['EI', 'PI', 'gp_hedge'],
             tooltip=('Controls how much improvement one wants over the ' +
                      'previous best values. Higher value ' +
                      'favours exploration over exploitation and vice versa'),
            ),
    ]
    return optimizer_cfg
```

7.3.3. Using custom optimizers

To make the new optimizer available to *Labber*, place it in the local optimizer folder and click the menu alternative *“Tools/Reload Optimizers”* in the *Measurement Setup* dialog. This will scan the optimizer folders and update the *“Method”* control in the general optimizer settings.

It is highly recommended to first test the optimizer in a pure Python environment. For an example of how to test the optimizer, see the code at the end of the file `Nelded-mead.py` provided in the global optimizer folder.

8. Log Browser

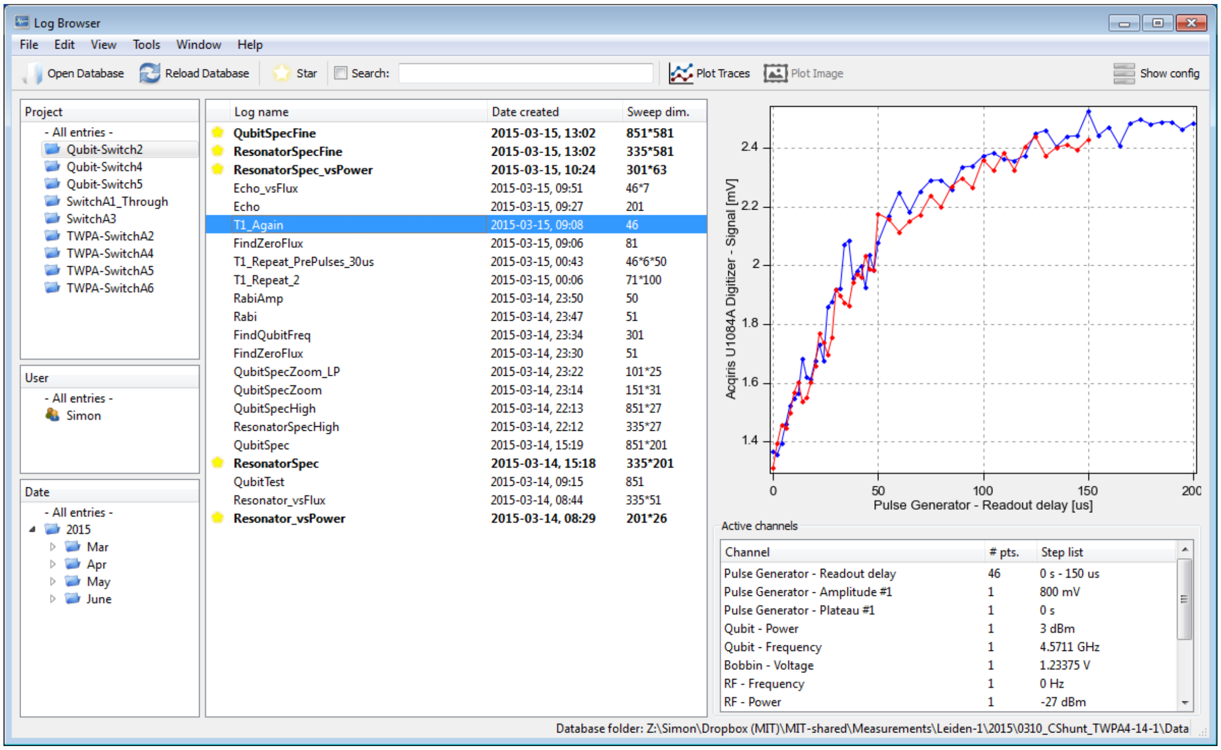
8.1. Database

The log database consist of a set of *Labber* log files organized in a special folder structure. When running a measurement, the program will save the configuration and data in a single file in the folder *"<Database folder>/xxxx/yy/Data_yyzz"*, where *<Database folder>* is the base database folder as set in the *Preferences* window (see Section PrefsFolder), *xxxx* is the current year (four digits), *yy* is the current month (two digits) and *yyzz* is the current month+day (total four digits). As an example, the database foLabber_manuallder for logs created on January 29, 2014 would be *"<Database folder>/2014/01/Data_0129"*.

The user can add additional data like images, scripts or even subfolders to folders within the main database folder. The *Log Browser* and the *Measurement* programs will ignore any additional files when scanning for *Labber* log files.

8.2. Log browser dialog

The *Log browser* is used to browse through the measured data and give a quick overview of the individual log files. The *Log Browser* is started by from the system tray menu (*"Show Log Browser"*) or by selecting *"Window/Show Log Browser"* from the main *Instrument Server* window. When starting, the program will scan through the default database folder for log files.



	Log name	Date created	Sweep dim.
★	**QubitSpecFine**	**2015-03-15, 13:02**	**851*581**
★	**ResonatorSpecFine**	**2015-03-15, 13:02**	**335*581**
★	**ResonatorSpec_vsPower**	**2015-03-15, 10:24**	**301*63**
	Echo_vsFlux	2015-03-15, 09:51	46*7
	Echo	2015-03-15, 09:27	201
	T1_Again	2015-03-15, 09:08	46
	FindZeroFlux	2015-03-15, 09:06	81
	T1_Repeat_PrePulses_30us	2015-03-15, 00:43	46*6*50
	T1_Repeat_2	2015-03-15, 00:06	71*100
	RabiAmp	2015-03-14, 23:50	50
	Rabi	2015-03-14, 23:47	51
	FindQubitFreq	2015-03-14, 23:34	301
	FindZeroFlux	2015-03-14, 23:30	51
	QubitSpecZoom_LP	2015-03-14, 23:22	101*25
	QubitSpecZoom	2015-03-14, 23:14	151*31
	QubitSpecHigh	2015-03-14, 22:13	851*27
	ResonatorSpecHigh	2015-03-14, 22:12	335*27
	QubitSpec	2015-03-14, 15:19	851*201
★	**ResonatorSpec**	**2015-03-14, 15:18**	**335*201**
	QubitTest	2015-03-14, 09:15	851
	Resonator_vsFlux	2015-03-14, 08:44	335*51
★	**Resonator_vsPower**	**2015-03-14, 08:29**	**201*26**

Channel	# pts.	Step list
Pulse Generator - Readout delay	46	0 s - 150 us
Pulse Generator - Amplitude #1	1	800 mV
Pulse Generator - Plateau #1	1	0 s
Qubit - Power	1	3 dBm
Qubit - Frequency	1	4.5711 GHz
Bobbin - Voltage	1	1.23375 V
RF - Frequency	1	0 Hz
RF - Power	1	-27 dBm

Database folder: Z:\Simon\Dropbox (MIT)\MIT-shared\Measurements\Leiden-1\2015\0310_CShunt_TWPA4-14-1\Data

Fig. 8.1. The Log Browser window.

Figure 8.1 shows the main *Log Browser* window. The dialog consists of a tool bar at the top, with sections on the left showing the database structure, a list in the center with log files, and finally a graph and info controls on the right giving a preview of the selected log file. The function of the various controls are described below.

8.2.1. Database hierarchy view

The four fields *Project*, *Tags*, *User* and *Date* on the left-hand side of the *Log Browser* dialog give an overview of the database hierarchy and allow the user to limit the selection of logs visible in the main log list. The *Project*, *Tags* and *Date* entries can be expanded to reveal subfolder selection. The filtering process is exclusive, meaning that only log files that fulfill the constraints of all the four fields are shown in the log list. Selecting the "- *All entries* - " item in one of the controls will disable the filtering for that field.

The *Project*, *Tags*, *User* and *Date* fields can be hidden from the *View*-menu in the main menubar.

8.2.2. Log list

The list in the center of the dialog shows the logs in the database that fulfill the constraints set by the hierarchy fields on the left. Logs that contain particularly important data can be starred, which will make them easier to spot in a large selection of logs. To star a log, either right-click the entry and select *"Star"* from the pop-up menu, or select a log and use the *"Star"*-button in the tool bar at the top of the window, or just press the space bar after a log has been selected.

The controls in the toolbar aboce the log list provide options for showing only starred log and for filtering logs by name. The log list can be sorted by *Name*, *Creation date* or *Sweep dimensions* by clicking the corresponding list header.

8.2.3. Graph / Log info

The graph in the top-right corner gives a quick preview of the contents of the selected log file. By default, the graph will show an image map if the data is two-dimensional in nature, and otherwise a line plot containing the first few entries in the log. The preview can be changed by creating a *View* in the *Log Viewer*, see Section Views for more information on *Views*. Below the graph, there are controls showing the *Step sequence*, the *Log channels* and the comment (if present) of the currently selected log.

8.2.4. Tool bar

The toolbar contains the following buttons:

Open Database:

Open another database than the default one used by the *Measurement* program. The dialog that opens should be pointed to the folder containing the *Year*-folders of the log database.

Reload Database:

Reload the current database and scan through all the log files. This is needed if log files have been manually added to the database folder.

Star:

Star/unstar the currently selected log.

Plot Traces:

This will open the currently selected log in the *Log Viewer*, see Section LogViewer for more information about the *Log Viewer* dialog.

Plot Image Map:

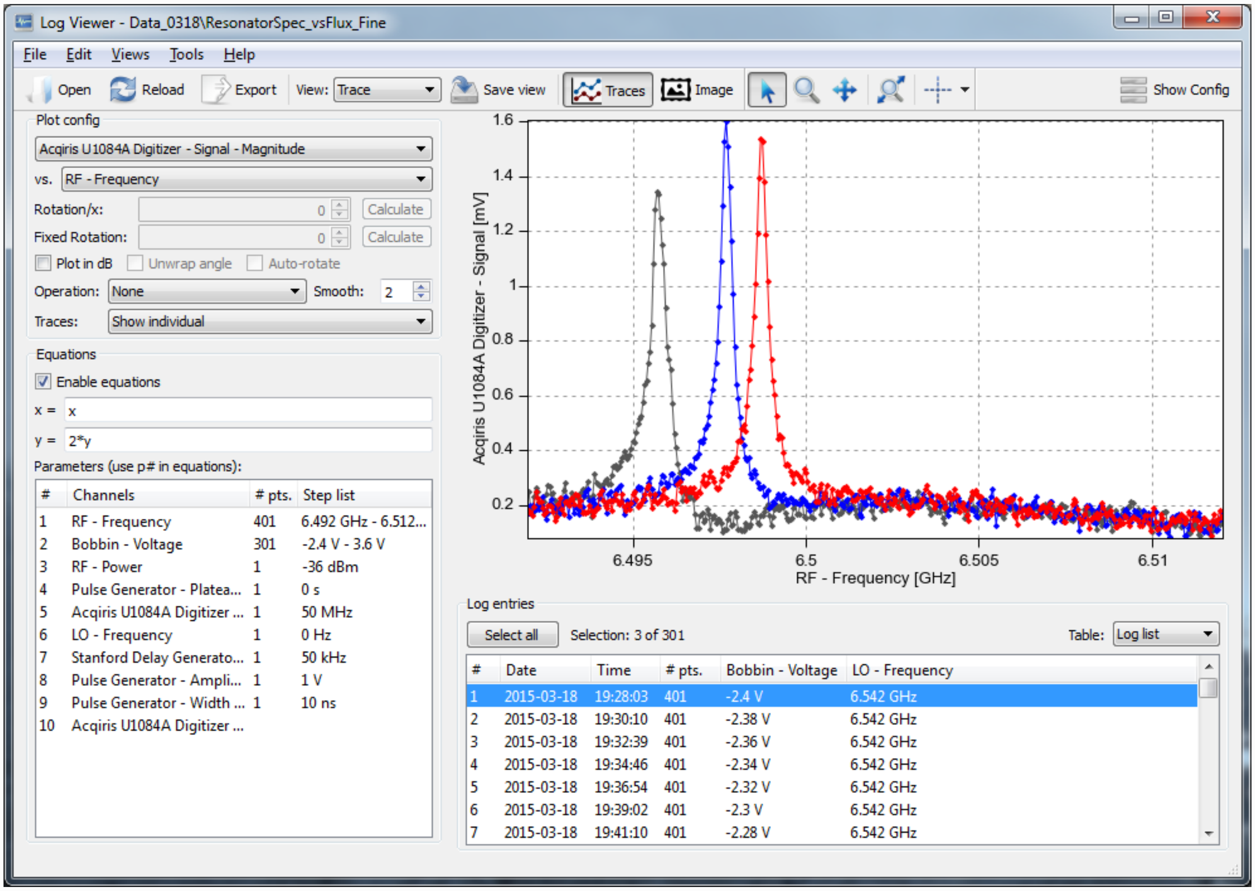
If the selected log contains 2-dimensional data, this button will open the data as an image plot in the *Log Viewer*.

Show config:

The button will show/hide an additional side bar with all the instrument configurations in the currently selected log. At the bottom of the sidebar, there will also be a checkbox *"Show all quantities"*; if checked, all instrument quantities and values are shown in the list, otherwise only the quantities present as channels in the *Measurement* configuration are displayed.

9. Log Viewer

The *Log Viewer* provides an environment for plotting and analyzing log files. The viewer is started by double-clicking a log in the *Log Browser*, which will bring up the main *Log Viewer* window (see Fig. 9.1). The dialog contains a list with all entries in the log, a plot showing the currently selected entries, and a set of controls on the left for configuring the plot. The data can be plotted as individual *Traces* or as an *Image*, and the user can quickly switch between the two modes using the *Traces/Image* buttons in the tool bar. The *Log Viewer* also provide options for saving a *View* of the log data, which allows plot settings and analysis configurations to be easily restored.



Log Viewer - Data_0318\ResonatorSpec_vsFlux_Fine

View: Trace

Plot config

Acqiris U1084A Digitizer - Signal - Magnitude

vs. RF - Frequency

Rotation/x: 0

Fixed Rotation: 0

☐ Plot in dB ☐ Unwrap angle ☐ Auto-rotate

Operation: None Smooth: 2

Traces: Show individual

Equations

☑ Enable equations

x = x

y = 2*y

Parameters (use p# in equations):

#	Channels	# pts.	Step list
1	RF - Frequency	401	6.492 GHz - 6.512...
2	Bobbin - Voltage	301	-2.4 V - 3.6 V
3	RF - Power	1	-36 dBm
4	Pulse Generator - Platea...	1	0 s
5	Acqiris U1084A Digitizer ...	1	50 MHz
6	LO - Frequency	1	0 Hz
7	Stanford Delay Generato...	1	50 kHz
8	Pulse Generator - Ampli...	1	1 V
9	Pulse Generator - Width ...	1	10 ns
10	Acqiris U1084A Digitizer ...		

Log entries

Selection: 3 of 301 Table: Log list

#	Date	Time	# pts.	Bobbin - Voltage	LO - Frequency
1	2015-03-18	19:28:03	401	-2.4 V	6.542 GHz
2	2015-03-18	19:30:10	401	-2.38 V	6.542 GHz
3	2015-03-18	19:32:39	401	-2.36 V	6.542 GHz
4	2015-03-18	19:34:46	401	-2.34 V	6.542 GHz
5	2015-03-18	19:36:54	401	-2.32 V	6.542 GHz
6	2015-03-18	19:39:02	401	-2.3 V	6.542 GHz
7	2015-03-18	19:41:10	401	-2.28 V	6.542 GHz

Fig. 9.1. The Log Viewer dialog in Trace mode.

9.1. Plot config

The plot configuration tools on the left-hand side of the dialog determines what is plotted in the graph. The controls are:

Y-channel:

The top-most control sets the channel to plot in the graph. If the channel contains complex values, the user can choose to plot
the *Real*, *Imaginary*, *Magnitude* or *Phase* of the signal.

X-channel:

The next control defines the x-axis in the plot.

Rotation/x (complex only):

This introduces a phase rotation per x-unit to a complex trace, which has the same effect has compensating for electrical delay when plotting signal transmission versus frequency. The *"Calculate"*-button next to the control estimates the rotation value that will best compensate such delays. The controls are only visible if the plotted quantity is complex.

Fixed Rotation (complex only):

This introduces a fixed phase rotation to a complex trace. The *"Calculate"*-button next to the control estimates the rotation value that will maximize the signal in the real component, while minimizing the signal in the imaginary one. The controls are only visible if the plotted quantity is complex.

Plot in dB (complex only):

If plotting magnitude, this converts the value to dB, using the formula 20*log^{10}(y). The control is only visible if the plotted quantity is complex.

Unwrap angle (complex only):

If plotting the phase, this will unwrap phase jumps around ±180°. The control is only visible if the plotted quantity is complex.

Auto-rotate (complex only):

If checked, the program will automatically rotate the phase of each trace to maximize the real part of the signal. This is the same as the the *"Fixed Rotation/Calculate"*-button, but the algorithm is applied for each trace individually, which means that the individual traces will generally be rotated by different amounts.

Operation:

Applies an operation to each selected trace. The following operations are available:

$\boldsymbol{y} - \langle \boldsymbol{y} \rangle$: Subtract the average value from each trace.

Normalize: The aces are normalized using the formula $(\boldsymbol{y} - \langle \boldsymbol{y} \rangle)/\mathbf{std}(\mathbf{y})$.

$\boldsymbol{d}/\boldsymbol{dx}$: Calculate the numerical derivative.

FFT: Numerical Fourier transform. Only positive frequency are shown in the resulting plot.

Histogram: Bin the data into a histogram, the number of bins is set by the "*Bins*" control.

Histogram-2D: Bin complex data into a 2D histogram, with the *x/y*-axes given by the real and imaginary parts of the data. The function only works for complex values.

Smooth:

Smooth the trace by taking a running average over the number of data points specified in the control.

Traces:

Applies an operation to the collection of all selected traces. The following trace operations are available:

Show individual: Standard operation, plot the individual traces as they are.

Subtract first: Subtract the first selected trace from the following traces.

Subtract previous: Subtract the previous trace. The plot will contain N/2 elements, with data (Trace 2 - Trace 1), (Trace 4 - Trace 3), (Trace 6 - Trace 5), etc…

***Average*:** Plots the average of all selected traces.

***Standard deviation*:** Plots the standard deviation of all selected traces.

9.1.1. Equations

If the *“Enable equations”*-control is checked, the x- and y-values in the plot are modified according to the equations given in the two text controls. The equation supports most standard mathematical functions like `cos(x)`, `sin(x)`, `sqrt(x)`, `exp(x)`, etc… Note that the raised operator (^) is implemented as two multiplication signs (`**`).

In addition to the variables x and y that represents the input data, the following parameters can be used in the equations:

***p*#:**

Value of other channels in the measurement. The channels are accessed by the parameter **p#**, where **#** is a number that represents the channel shown in the list below the equation controls. Note that the value will be complex if the channel represents a complex quantity; use `real(p#)`, `imag(p#)` or `abs(p#)` to get real, imaginary or the magnitude of the data.

***n*:**

A vector with values $\{\mathbf{1}, \mathbf{2}, \mathbf{3}, \ldots, \boldsymbol{n_{tot}}\}$, where $\boldsymbol{n_{tot}}$ is the number of elements in the trace.

***m*:**

Trace number, starting with **1** for the first measured trace, which is the same as the #-parameter in the log entry list.

m0:

Trace number, starting with **1** for the first selected trace.

9.1.2. Physical vs. Instrument units

Select *"Tools/Plot Dat in Instrument Units"* to show the data in instrument units instead of physical units. The default units (physical or instrument) can be set in the *Preferences* dialog, under *"Measurements/Units"*.

9.2. Entry list

The entry list shows the contest of the log file, with each entry representing a one-dimensional trace of data. For multi-dimensional logs, the *Log Viewer* supports two different modes, which are controlled by the *"Table"*-control to the right of the list. The two modes are:

Log list:

This is the default mode, where the list contains the entries in the order that they were measured, and where the selected entries are shown directly in the graph.

Multi-column:

In this mode, the list becomes multi-dimensional, with each column representing a step dimension in the *Measurement* configuration file. The mode supports data slicing along different dimensions. The slice directions is set by the *"Slice parameter"*-control directly above the log list.

9.3. Tool bar

The toolbar contains the following buttons:

Open Log:

Open another log file.

Reload Log:

Reload the current log file, which is useful if the measurement is ongoing and new data has been added to the log.

Export:

Show the *Export Figure* dialog, for exporting the currently selected data to an image file.

Views/Save View:

Select/save the current view. See Section Views for more information.

Traces/Image:

Switch between *Trace* and *Image* plot mode.

Plot controls:

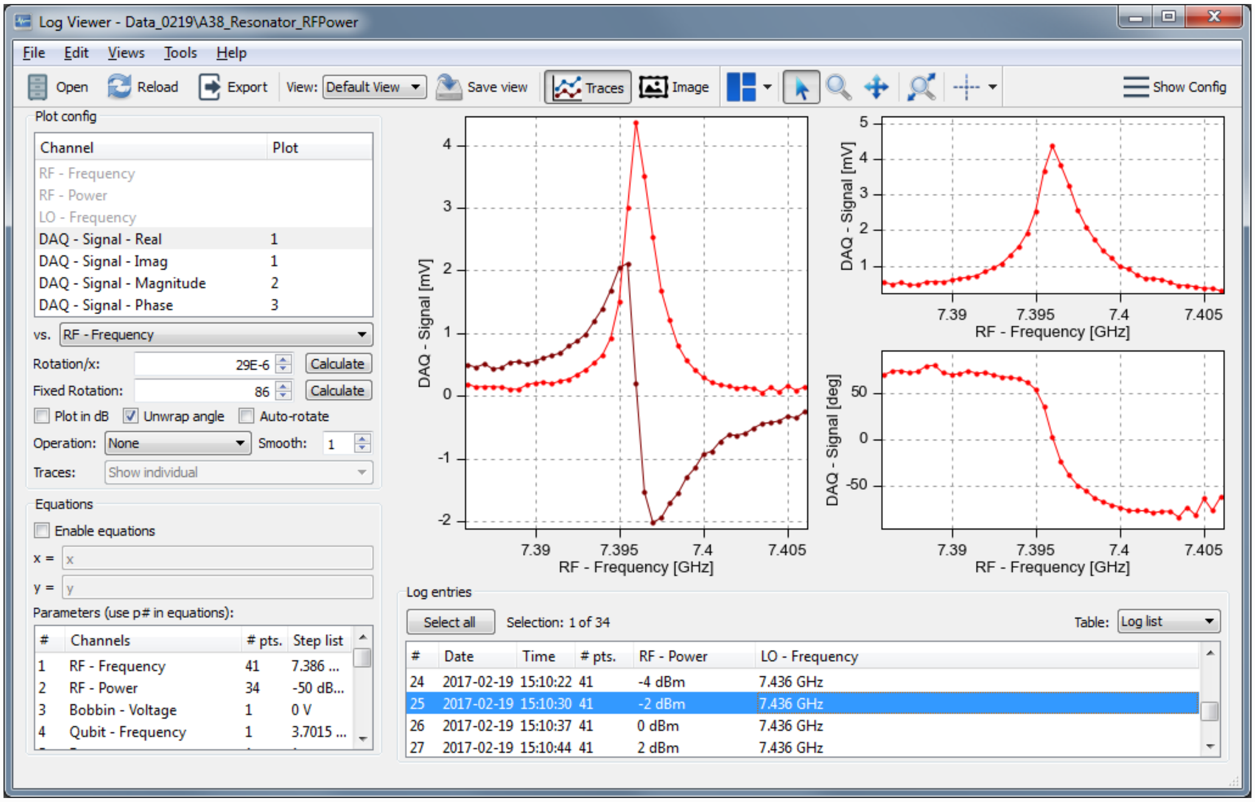
The plot controls contain tools for zooming/panning the graph, and for enabling/disabling the cursors. If in *Image* mode, there are a few extra buttons for transposing the data and enabling cross-sections and contrast controls.

Show config:

The button will show/hide an additional side bar with all the instrument configurations in the currently selected log. Below the list of quantities, there is a checkbox *"Show all quantities"*; if checked, all instrument quantities and values are shown in the list, otherwise only the quantities present as channels in the *Measurement* configuration are displayed. The *"Project"* and *"User"* controls at the bottom of the dialog allow the *Project* and *User* tags to be modified.

9.4. Multi-panel graph mode

The multi-panel graph allows multiple channels from a single log entry to be plotted in one or multiple graphs, as shown in Fig. 9.2. To enable multi-panel graph mode, select *"Views/Show multiple Graphs"* from the pull-down menu. The multi-panel mode is enabled by default if the log file contains more than one log channel. When the multi-panel graph mode is enabled, the toolbar at the top of the window contains an extra sub-menu for selecting number of figures to be shown, and for controlling whether the x- and y-axes of the figures should be synchronized or not.



Channel	Plot
RF - Frequency	
RF - Power	
LO - Frequency	
DAQ - Signal - Real	1
DAQ - Signal - Imag	1
DAQ - Signal - Magnitude	2
DAQ - Signal - Phase	3

vs. RF - Frequency

Rotation/x: 29E-6

Fixed Rotation: 86

□ Plot in dB ☑ Unwrap angle □ Auto-rotate

Operation: None Smooth: 1

Traces: Show individual

□ Enable equations

x = x

y = y

Parameters (use p# in equations):

#	Channels	# pts.	Step list
1	RF - Frequency	41	7.386 ...
2	RF - Power	34	-50 dB...
3	Bobbin - Voltage	1	0 V
4	Qubit - Frequency	1	3.7015 ...

Selection: 1 of 34

Table: Log list

#	Date	Time	# pts.	RF - Power	LO - Frequency
24	2017-02-19	15:10:22	41	-4 dBm	7.436 GHz
25	2017-02-19	15:10:30	41	-2 dBm	7.436 GHz
26	2017-02-19	15:10:37	41	0 dBm	7.436 GHz
27	2017-02-19	15:10:44	41	2 dBm	7.436 GHz

Fig. 9.2. The Log Viewer dialog in multi-panel graph mode.

In multi-panel graph mode, the *Y-channel* control in the *Plot config* group in the upper-left corner of the window is replaced by a list of all measured channels. The channels can be assigned to one or multiple graphs by right-clicking the channel name and selecting a graph, by right-clicking one of the graphs and selecting a channel, or by dragging a channel entry onto one of the graphs. The default multi-panel graph configuration can be set in the *Preferences* dialog.

9.5. Image mode

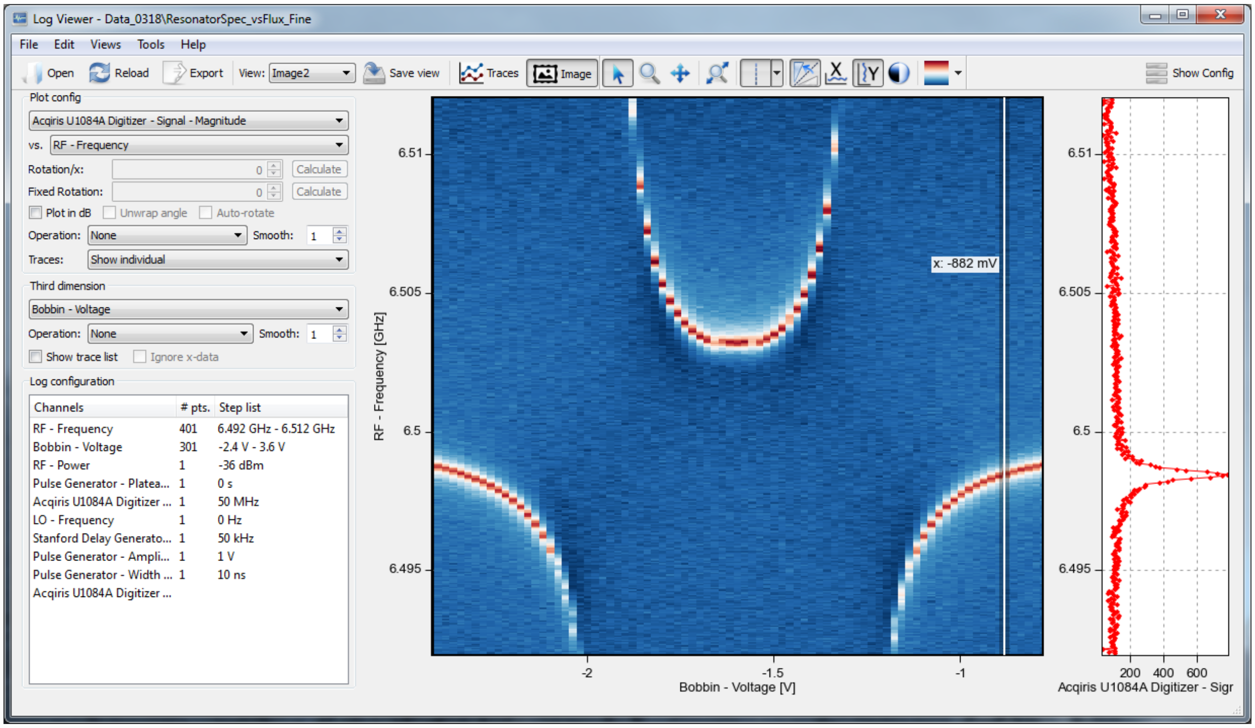
In *Image* mode, the graph with the individual traces is replaced by an image map, as depicted in Fig. 9.3. The third-dimension data is specified by the controls in the *"Third dimension"*-group in the left-hand side of the dialog. In addition to specifying the data source, there are controls for performing basic signal operations along the third dimension, similar to the trace operations described in Section PlotConfig. The *"Third dimension"*-group also contains the following buttons:

Show trace list:

If checked, a trace list is shown allowing the user to select which traces to include in the image map.

Ignore x-data:

If checked, the program will take the x-data from the first data trace and ignore the x-values of subsequent traces. This is useful for confining the representation to a square image plot for data where the x-values are changing from trace to trace.



Log Viewer - Data_0318\ResonatorSpec_vsFlux_Fine

View: Image2

Plot config

Acqiris U1084A Digitizer - Signal - Magnitude

vs. RF - Frequency

Rotation/x: 0

Fixed Rotation: 0

□ Plot in dB □ Unwrap angle □ Auto-rotate

Operation: None Smooth: 1

Traces: Show individual

Third dimension

Bobbin - Voltage

Operation: None Smooth: 1

□ Show trace list □ Ignore x-data

Log configuration

Channels	# pts.	Step list
RF - Frequency	401	6.492 GHz - 6.512 GHz
Bobbin - Voltage	301	-2.4 V - 3.6 V
RF - Power	1	-36 dBm
Pulse Generator - Platea...	1	0 s
Acqiris U1084A Digitizer ...	1	50 MHz
LO - Frequency	1	0 Hz
Stanford Delay Generato...	1	50 kHz
Pulse Generator - Ampli...	1	1 V
Pulse Generator - Width ...	1	10 ns
Acqiris U1084A Digitizer ...		

Fig. 9.3. The Log Viewer dialog in Image mode.

If the log files contains more than two dimension, there will be a *"Data selection"*-list appearing under the group of *"Third dimension"* controls. The list allows the user to select which subset of data to show in the image plot. In image mode, there are a few extra buttons in the toolbar at the top of the window.

Transpose:

Switch X- and Y-axes in the image map.

Contrast:

Show the contrast controls, allowing the user to set the contrast range of the image. The range can be manually controlled by shifting the range cursors in the spectrum plot. The *"Auto range"*-button will optimize the contrast by removing outliers, while the *"Full range"*-button will return to full range.

X/Y cross sections:

Show the *X/Y*-cross sections. The position of the cross section is controlled by moving the cursors around.

9.6. Views

Views provide a way to save the current plot settings and selection of log entries, so that the current view can be easily restored. The most recently defined *View* will also be the preview of the log that is shown in the preview graph of the *Log browser*.

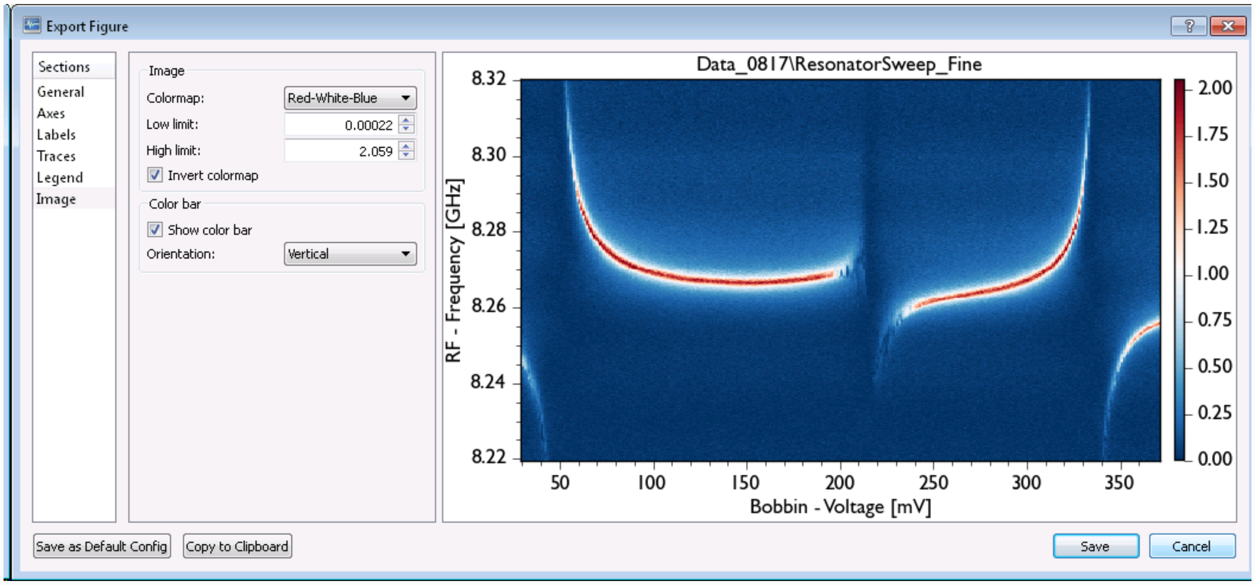
To save the current view, click the *"Save view"*-button in the tool bar or select *"Views/Save view…"* in the menu, define a name of the *View* and click the *"OK"*-button.

To restore a previously saved view, select the *View* to show from the *"View"*-control in the toolbar. *Views* can be renamed or deleted from the *Edit View*-dialog, which is accessed by selecting *"Views/Edit views…"* in the menu bar.

9.7. Exporting data

Data can be exported to a text file, to a *Matlab ".mat"* file, or as an image. The export options are available from the *"File"*-menu.

9.7.1. Exporting to Image



Image

Colormap: Red-White-Blue

Low limit: 0.00022

High limit: 2.059

☑ Invert colormap

Color bar

☑ Show color bar

Orientation: Vertical

Fig. 9.3. The Export Image dialog makes it easy to generate publication-quality figures.

The currently selected plot can be saved as an image file by either selecting *“Tools/Save Screenshot”*, or saved to the clipboard by selecting *“Tools/Copy Graph”*. To modify the labels and the style of the image, select *“Export/Export as Image…”* or click the *Export* button in the toolbar. This will open the *Export Figure* dialog, which makes it possible to modify the figure axes and labels to render publication-quality figures (see Fig. 9.3). The resulting image can be copied to the clipboard or saved in JPEG, PNG, SVG or Adobe PDF format.

9.7.2. Exporting to Text

When exporting the text, a dialog will open allowing the user to define what to export. The following options are available:

Data to export:

Determines whether to export all or only the traces currently selected in the *Log Viewer*.

Include header with log information:

If checked, a header with log info will be included at the top of the text file.

Include separate x-data with each trace:

If checked, each data entry will contain two rows of data, one for x and one for y. If unchecked, the first row is x-data, and all the following rows are y-data.

Include data for third dimension:

If checked, data for a third dimension is added to the end of the text file. The channel for which the third dimension data is taken is defined in the control below the check box.

9.7.3. Exporting to Matlab

The *Matlab* export will export the selected traces to a single *".mat"* file. The data is saved into a *Matlab* struct; the data structure is shown when opening the file in the *Matlab* workspace browser.

9.7.4. Custom Export

The *Custom Export*-function allows the user to export data into a custom format. Note that the custom export function always exports the raw data, without applying any operations such as smoothing, FFT, etc. The following options are available when exporting custom data:

Data to export:

Determines whether to export all or only the traces currently selected in the *Log Viewer*.

Custom script:

Path to python file containing the custom `exportData` function.

The custom export functionality needs to be implemented in a python function called `exportData`, which should be located in a separate python file (*.py*). An example of a custom export script can be found in the file *ExportScript.py* in the *Script* folder of the main program directory (see Section Folders for an overview of folder locations). The function definition of the `exportData`-function must have the following format:

```
def exportData(file_name, step_data, log_data, step_name, log_name,
               step_unit, log_unit, comment='')
```

file_name:

Output path for the exported data.

step_data:

Data for stepped channels. The data is defined in a nested python list of 1-d *numpy* arrays (one for each trace). The first index is the step channel number as defined in the *Measurement* dialog step list, the second index is the trace number. For example, to access the data for the innermost step channel and the third trace, use `step_data[0][2]`.

log_data:

Data for log channels, defined in the same way as the `step_data`.

step_name:

List of strings defining the step channel names.

log_name:

List of strings defining the log channel names.

step_unit:

List of strings defining the step channel units.

log_unit:

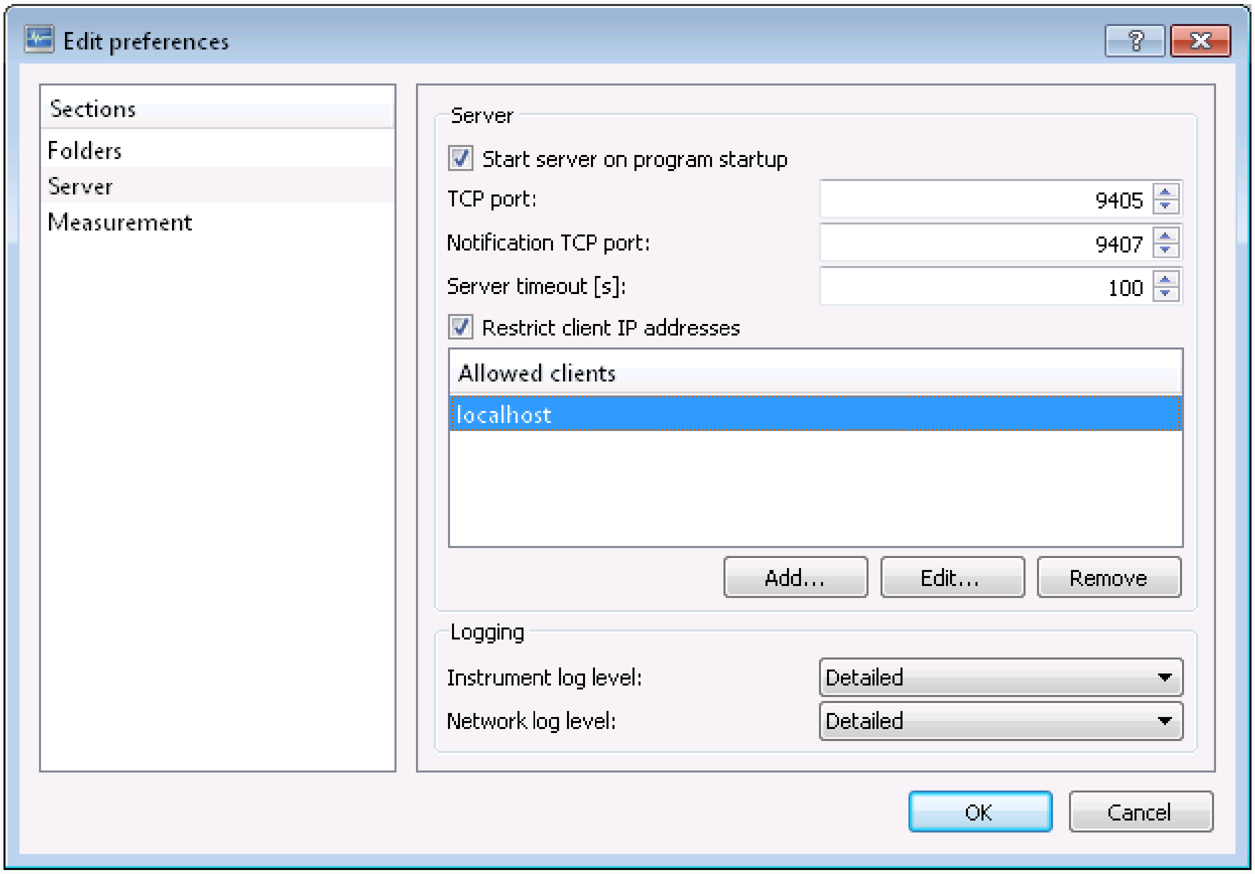
List of strings defining the log channel units.

comment:

Log comment.

10. Preferences

To access the Preferences dialog, select *Preferences…* from the *Instrument Server* tray icon menu, or *“Edit/Preferences…”* from the pull-down menu. The dialog has the following sections and settings:


Edit preferences

Sections
- Folders
- Server
- Measurement

Server

☑ Start server on program startup

TCP port: 9405

Notification TCP port: 9407

Server timeout [s]: 100

☑ Restrict client IP addresses

Allowed clients
- localhost

Logging

Instrument log level: Detailed

Network log level: Detailed

Fig. 9.4. Dialog for setting preferences.

10.1. Folders

The *Folders* section defines the location of various folders used by the program.

Database folder:

Main database folder for saving data from the *Measurement* program. Default value is *“<User home directory>/Labber/Data”*.

Instrument drivers:

Folder containing instrument drivers provided by *Labber*. Do not alter this folder location unless having good reasons for doing so.

Local drivers:

Folder containing user-defined instrument drivers.

Optimizer functions:

Folder containing optimizer functions provided by *Labber*. Do not alter this folder location unless having good reasons for doing so.

Local optimizers:

Folder containing user-defined optimizer functions.

10.2. Server

The *Server* section contains settings related to the *Instrument Server*.

Start server on program startup:

If True, the server starts listening for incoming connections when the program starts up. If False, the user has to start the server manually. Default value is True.

TCP port:

TCP port used for communication between the *Instrument Server* and the clients. Default port is 9406.

Notification TCP port:

TCP port used for sending notifications between the various program parts. The communication only occurs on the local computer. The default port is 9407.

Data transfer format:

Format use for data transfer over the network. Binary is faster, whereas text is human-readable and better for debugging purposes. Default is Binary.

Server timeout:

Maximum waiting time before the server returns an error. This value should be reasonable long, in case an instrument takes a long time to perform an operation. Default value is 1,000,000 seconds.

Restrict client IP addresses:

Restrict allowed clients according to the list defined below. Default is True.

Allowed clients:

List of IP numbers of allowed clients. Request from computers with IP numbers outside the list will be rejected. Note that it is possible to define wild cards, for example `"192.168.*"` will allow connections from any client with IP starting with 192.168. Default value is *localhost*, which only allows connections from the same computer that is running the server.

Allow instruments to be controlled from driver configuration window:

If True, instrument settings can be updated directly from the driver configuration window while a driver is running. If False, the controls are grayed out once the driver is started, which makes it less likely that incorrect/too large instrument values are outputted by mistake. Default is True.

Keep instrument drivers running after the measurement ends:

Starting up a driver may take a few seconds, depending on system. Therefore, stopping and starting the driver between measurements may slow down experiments, which can be avoided by keeping the driver running after a measurement ends. Default is True.

Change background color for active instruments:

Change background color of driver dialog for active instruments, to highlight that changes to any parameter of the instrument driver window will directly update the instrument hardware. Default is True.

Change background color for instruments in Measurement dialog:

Use different background color for instrument configuration dialogs in the Measurement program than in the Instrument server, to make it clear which program the dialog belongs to. In the *Instrument Server*, the instrument configuration dialog is used to *directly* control the hardware settings, meaning that any changes to the dialog will directly affect the state of the hardware. In contrast, in the *Measurement* program the dialog is used to set up a configuration that will be used in a specific *Measurement*, but no changes are made to the hardware until the measurement is started. To avoid confusion, if this setting is True the *Instrument driver* configuration windows have a different background color when opened within the *Measurement* program and in the *Instrument Server*. Default is True.

Instrument log level:

Amount of information to log when performing instrument communication. The log can be viewed by selecting *"Log/View Instrument Log..."* in the *Instrument Server* menu bar. Default value is Basic.

Network log level:

Amount of information to log when performing network communication. The log can be viewed by selecting *"Log/View Network Log..."* in the *Instrument Server* menu bar. Default value is Basic.

10.3. Measurement

This section contains settings related to the *Measurement* program.

Sort step items before starting Measurement:

If checked, step items are sorted according to instrument type before starting a Measurement.

Default units, step sequences:

Default units when defining a new step sequence in the *Measurement Setup* dialog.

Default units, viewing data:

Default units when viewing data in the *LogBrowser* and the *LogViewer*.

Default sweep units:

Set if sweep rates should be defined in terms of rate per second or rate per minute in the *Instrument Server* and *Measurement Editor* programs. If set to *Instrument default*, the program will use the default units defined in the settings of each instrument driver (see Section SweepDriver). Note that this setting only affects the sweep units shown in the dialog windows, the sweep units used within a particular instrument driver implementation is always set by the configuration file of the driver (see Section SweepDriver).

Graph refresh interval:

Refresh interval for graph. Use larger values if the user interface becomes unresponsive. Default value is 80 ms.

Default live colormap:

Default colormap for viewing image data in the live graph shown during measurements.

10.4. Log Viewer

Default colormap:

Default colormap when viewing data as images in the *LogBrowser* and the *LogViewer* dialogs.

Default cursor type:

Default cursor type in all graphs.

Default complex representation:

Default format for representing complex scalar data.

Default complex representation, vector:

Default format for representing complex vectors, typically from instruments such as spectrum analyzers and vector analyzers.

Default panel configuration, 2 channel:

Default multi-panel graph configuration for showing two log channels.

Default panel configuration, 3 channel:

Default multi-panel graph configuration for showing three log channels.

Default panel configuration, 4 channel:

Default multi-panel graph configuration for showing four log channels.

Save current view when closing Log Viewer:

Automatically save current view when closing the Log Viewer.

10.5. Logger

Logger folder:

Database folder for saving logging data from the Logger program.

Number of points in Acquire graph:

Number of points shown in the live logger graph.

Alarm de-activation range:

Range at which an out-of-range alarm de-activates.

Dark mode:

If checked, the visualizer will plot data on a dark background.

Refresh interval in Logger Visualize:

Data refresh interval in Logger visualize.

10.6. Advanced

Application library:

Path to main Labber application.

Python distribution:

Path to custom Python executable. The Python distribution must be running Python 3.5 or later. Leave blank to use the built-in Labber Python distribution. For Windows, pick the executable `pythonw.exe` instead of `python.exe` to avoid creating a console window for each driver process. For more information, see Section PythonDistExternal.

Temporary items:

Folder for storing settings and temporary items. Do not alter this item unless having good reasons for doing so.

Show error if setting the value of an inactive quantity:

If unchecked, the program will not show an error if trying to set the value of an inactive quantity.

Send status updates to clients:

Send status updates from Instrument server to log clients during slow operations such as sweeping.

Interval for checking swept instruments:

Time interval between checks when testing if a swept instrument has reached the final value.

VISA library:

Path to VISA library. Leave blank to use default library.

Delay for wait dialog:

Shortest delay time for showing the wait dialog. Default value is 2 seconds.

Show error dialog in script mode:

If unchecked, no error dialog will be shown if an error occurs during a scripted Measurement. This can be useful if no user interactions is required to handle errors.

Run queued experiments in separate process:

If checked, queued measurements will run in a separate instance of the Measurement program. This may cause conflicts if queued experiments and measurement from the user interface are started at the same time.

11. Scripting

This chapter describes how to write scripts to perform sequence of experiments. Scripting is useful for running multiple experiments after each other, or for defining sequences where some properties of a measurement is updated depending on the result of a previous measurement.

In a typical scripting setup, the user would first create a number of *Measurement* configurations using the standard *Measurement* configuration dialog, and then save those configurations files to a folder on disk. The script would then be programmed to execute those configurations, either as they are or by first updating one or multiple parameters of the *Measurement* configurations.

The most basic way of implementing scripting is to call the *Measurement* program with command-line arguments, as described in Section console below. The advantage of this method is that one can use any programming language that supports calling an external program for writing the script, the disadvantage is that the function calls can become rather long and difficult to read. If you plan to script experiments using the programming language *Python*, there are a number of helper functions that will simplify the procedure. These helper functions are described in Section scriptPython.

11.1. Console options

In addition to the user-interface based *Measurement* configuration dialog, it is also possible to start experiment from the command line. The program is called `Measurement-Console.exe`, and is located in the main program folder (see Section Installation for the folder structure). The command-line arguments are:

```
Measurement-Console.exe [-h] -i INPUT_PATH [-u CHANNEL VALUE TYPE]
             [-m CHANNEL] [-o OUTPUT_PATH] [-e EXPORT_PATH] [-r CHANNEL]
```

-h, –help:

Show a help message and exit

-i INPUT_PATH:

Path to the measurement configuration file to execute or rearrange.

-u CHANNEL VALUE TYPE:

Update the step item `CHANNEL` with a new value. The `TYPE`-argument defines what property of the step item to update, and must be one of `SINGLE, START, STOP, CENTER, SPAN, STEP, N_PTS`. Note that scripted measurements do not raise an error if updating an inactive step item. Instead, the step item is automatically switched to the new step type. For example, if the original step type is `SINGLE`, and the user updates the `START` value, the step type is changed to `START-STOP`. Note that it is up to the user to ensure that all other relevant quantities are updated as well (in the example, the `STOP` value and the `STEP` or `N_PTS` value).

-m CHANNEL:

Specifies the master channel name. Values of all other updated channels will be defined by look-up tables relative to the master channel values.

-o OUTPUT_PATH:

Specifies the path of the output log file. If not given, the data will be be saved to the input measurement configuration file.

-e EXPORT_PATH:

After completed measurement, export the last trace to the specified text file. Any previous contents in that file will be overwritten. This is useful for creating scripts where future measurements depend on the results of previous measurements.

-r CHANNEL:

Re-arrange a log with N 1-dimensional entries of length M to a 2-dimensional log with dimensions (N, M). The `CHANNEL` determines which data to use when defining the second dimension. It is also possible to rearrange into a multi-dimensional log by specifying multiple channels, but if so lists of step values for each dimension need to be specified as well. For example, to rearrange a log with 6 entries into a multi-dimensional log with 3*2 entries, use `-r "Channel 1" "1.0, 2.0, 3.0" "Channel 2" "1.0, 2.0"`. Note that the internal order of the new dimensions is defined by the order in which they appear in the step list of the original *Measurement* configuration file, *not* by the order they are listed after the `-r` command. Also, note that no measurement will be performed when running the program with this option.

If no arguments are given, the program will open the standard user interface window for configuring the experiment.

11.2. Scripting using Python

The Python scripting helper functions are part of the *Labber API*, located in the *Script* folder of the main program directory (see Section Folders for an overview of folder locations). To provide easy access to the *Labber API*, it's recommended to add the *Script* folder to your Python path. For more information on the Python API, see Section PythonAPI.

The helper functions in the *ScriptTools* module are designed for repeatedly performing a number of *Measurements* that each contain one-dimensional sweeps, and where one or multiple parameters of the *Measurement* configurations are updated between each measurement. The functions are best explained by an example, which we'll take from the domain of superconducting qubits. For the purpose of this example, we can view the qubit as a slightly anharmonic oscillators whose frequency tunes with applied magnetic flux. The qubit is read out by coupling it to microwave resonators, and the coupling is arranged in a way that changing the qubit frequency will cause a slight shift of the resonator frequency.

Now, say that we want to probe the qubit frequency as a function of applied flux. The difficulty is that the changing the flux will affect both the qubit and the resonator frequencies, which means that we can not use a fixed-frequency read-out tone. Instead, we need to implement the following procedure:

1. **Set new magnetic flux value**
2. **Measure resonator**
3. **Find resonance frequency f_0 of resonator**
4. **Measure qubit, while keeping the resonator at f_0**
5. **Repeat for all values of magnetic flux**

The file *ExampleScript.py* in the *Script* folder contain an example script for performing the sequence described above. The script assumes that the user has created two *Measurement* configurations, one for measuring the resonator, and one for measuring the qubit, and that both *Measurement* configurations have a single-valued step item called 'Flux bias' that control the magnetic flux.

12. Instrument drivers

This chapter describes the definition of instrument drivers and instructions for how to create custom drivers. The general driver structure is visualized in Fig. 12.1.
The *Communication* part describes the interface and address used for communication, and is normally handled by the *Labber Instrument server*. The *Model and options* part provides a way to enable/disable certain features of a driver depending on the instrument model/installed options. Finally, the list of *Quantities* define all properties and settings available on the instrument.

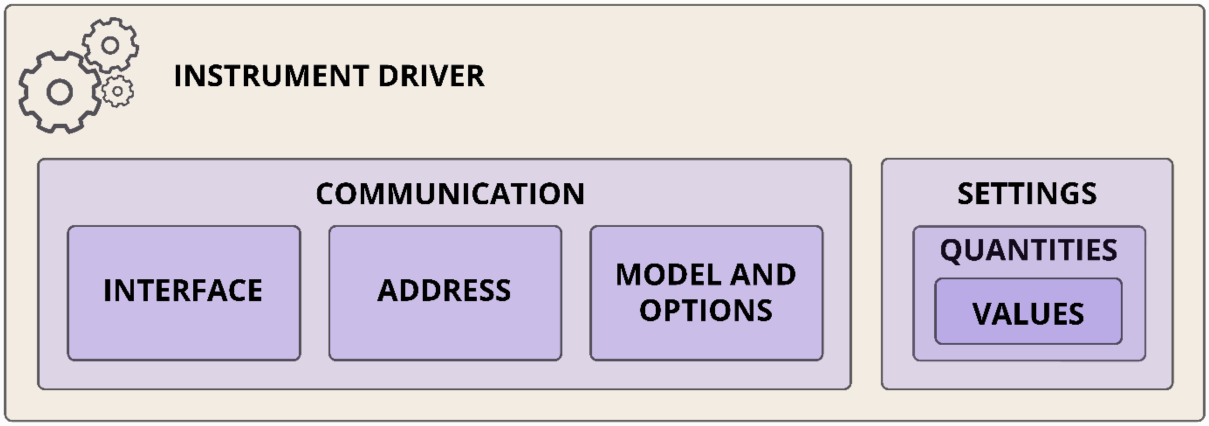


Fig. 12.1. Structure of an Instrument driver. The Communication part is handled by the Instrument server, while the Model and Options and the Quantities are defined in the driver configuration file.

12.1. Driver definition files

The driver definition file is a file specifying the instrument name, vendor, model and options as well as a list of quantities. For basic instrument drivers, where the value of each *quantity* can be set or read using a single text command over GPIB, serial, USB or ethernet using the *VISA* protocol, all information about the instrument and the communication is contained in the definition file. For more advanced drivers, for example network analyzers that capture vector data, the driver definition file needs to be complemented with *Python* source code for implementing the more advanced instrument operations (see Section PythonDriver). The Python code must be Python 3 compatible.

The driver definition files provided by *Labber* are located in the *“Instrument drivers”* folder (see Section PrefsFolder for information on folder locations). The definition files are plain text files using the *INI* file format, which consists of a number of “sections”, each containing a list of “properties”. The driver file requires implementing sections for *General settings*, *Model and options* and *VISA Settings*, see below for more information about each section. It is recommended to use one of the already present drivers configuration files as a template. For an example of creating an instrument driver from scratch, see Section PythonDriver.

When creating a new driver, the definition file should be placed in the *local* driver folder (the folder named *“Local drivers”* in the *Preferences* window), instead of the global one (*“Instrument drivers”* in *Preferences*). This allows the user’s own drivers to be kept separately from the drivers provided by *Labber*, and it also prevents drivers written by the user from being deleted when updating the *Labber* program to a newer version.
The *INI* configuration file can be placed directly in the *“Drivers”* folder, or within a subfolder of that directory. Using a subfolder is the recommended approach, since it gives a natural place to store extra files related to the driver.

Note that even when making additions/changes to an existing driver from the global folder, the best practice is to copy that driver file from the global folder to the local folder, and only make changes to the local version. If drivers with the same names exist in both the local and the global driver folders, *Labber* will always use the driver in the *“Local drivers”*-folder.

12.1.1. Signal Generators and Signal Analyzers

Signal Generators and *Signal Analyzers* are drivers that are used to generate or analyze waveforms. The drivers do not perform any instrument communication, which means that the *Model and options* and the *VISA Settings* parts of the *INI* file do not need to be defined. To define that a driver is a *Signal Generators* or a *Signal Analyzers*, set the corresponding item in the *General settings*-part of the *INI* file as described below. See Section Signals for more information about how *Signal Generators* or a *Signal Analyzers* are used in an experiment.

12.1.2. General settings

The *General settings*-section define name and version of the driver. Note that it is the *name* property in this section that sets the driver name, not the name of the driver definition *INI* file.

name:

The name is shown in all the configuration windows.

version:

The version string should be updated whenever changes are made to this config file.

driver_path:

Name of folder containing the code defining a custom driver. Do not define this item or leave it blank for any standard driver based on the built-in VISA interface.

interface:

Pre-defined communication interface for instrument, default is `GPIB`.

address:

Pre-defined address for instrument, default is an empty string.

startup:

Pre-defined startup option for instrument, default is `Set config`.

signal_generator:

Set to *True* if driver is a *Signal Generator*. Default is *False*.

signal_analyzer:

Set to *True* if driver is a *Signal Analyzer*. Default is *False*.

controller:

Set to *True* if driver is a *Controller*. Default is *False*. For more information, see Section. ControllerDriver below.

support_hardware_loop:

Set to *True* if driver supports hardware looping. Default is *False*.

support_arm:

Set to *True* if driver supports hardware arming. Default is *False*.

use_32bit_mode:

Set to *True* if driver should run in a 32-bit Python environment. Default is *False* (run in 64-bit). For more information, see Section PythonDist below.

12.1.3. Model and options

The *Model and options*-section provides a way to enable/disable certain features of a driver depending on the instrument model/installed options.

model_str_1, model_str_2, etc:

List of models supported by the driver.

check_model:

If *True*, the driver checks the instrument model id at startup (*True* or *False*). The model is checked by sending the `model_cmd` command (see below) over the *VISA* interface. Default is *False*.

model_cmd:

Command used to check the instrument model. Default command is `*IDN?`.

model_id_1, model_id_2, etc:

Model strings expected to be returned by the instrument by the `*IDN?` call. If not defined, the program assumes `model_str_1, model_str_2, etc` as default values

option_str_1, option_str_2, etc:

List of available instruments options. The options are shown as checkbox controls in the driver configuration window.

check_options:

If *True*, the driver checks the installed instrument options at startup (*True* or *False*). The option is checked by sending the `option_cmd` command (defined below). Default is *False*.

option_cmd:

If `check_options` is set to *True*, define command for getting the options from the instrument.

option_id_1, option_id_2, etc:

If `check_options` is set to *True*, supply valid id option strings that the instrument returns when sending the `option_cmd`. The list of `option_id` should match the elements in the list `option_str`.

12.1.4. VISA Settings

This section contains configuration of the *VISA* protocol. The *VISA* protocol enables text-based communication with instruments over GPIB, USB, serial and ethernet interfaces.

use_visa:

Enable or disable communication over the VISA protocol (*True* or *False*). If *False*, the driver will not perform any instrument operations (unless there is a custom *Python* driver, see Section PythonDriver).

reset:

Reset the interface (not the instrument) at startup (*True* or *False*). Default is *False*.

query_instr_errors:

Query instrument errors (*True* or *False*). If *True*, every command sent to the device will be followed by an error query. This is useful when testing instruments, but may degrade performance by slowing down the instrument communication.

error_bit_mask:

If `query_instr_errors` is *True*, set bit mask for checking status byte errors (default is 255, include all errors). The bits signal the following errors:

0: Operation

1: Request control

2: Query error

3: Device error

4: Execution error

5: Command error

6: User request

7: Power on

error_cmd:

Command string to be sent to instrument when querying for instrument error messages.

init:

Initialization commands are sent to the instrument when starting the driver. `*RST` will reset the device, `*CLS` clears the interface.

final:

Final commands sent to the instrument when closing the driver.

str_true:

String used for sending boolean *True* to the instrument, default is `1`.

str_false:

String used for sending boolean *False* to the instrument, default is `0`.

str_value_out:

Conversion string used for converting value to string to be sent to the instrument. Default is `%.9e`, which creates 9-digit string using exponential notation. To create strings with floating-point notation, use `%.9f` instead.

str_value_strip_start:

Number of characters to strip from the beginning of the string returned from the instrument, before trying to convert to a number. Default is `0`.

str_value_strip_end:

Number of characters to strip from the end of the string returned from the instrument, before trying to convert to a number. Default is `0`.

always_read_after_write:

If `True`, the program will automatically read the response from the instrument after each write command. Useful for instruments that always reply to all commands. Default is `False`.

The following entries are optional, they provide detailed settings for the communication interface. Note that the values provided in the *INI* file will be the default setting for the driver, but the user can always change the settings by going to the *Communication* settings of the instrument driver user interface and clicking *"Show advanced interface settings"*.

timeout:

Time (in seconds) before the timing out while waiting for an instrument response. Default is 5 seconds.

term_char:

Termination character used by the instrument, valid values are `Auto`, `None`, `CR`, `LF`, `CR+LF`.

send_end_on_write:

Assert end during transfer of last byte of the buffer

suppress_end_on_read:

Suppress end bit termination on read

baud_rate:

Communication speed for serial communication. Default is 9600.

data_bits:

Number of data bits for serial communication. Default is 8.

stop_bits:

Number of stop bits for serial communication. Default is 1, possible values are 1, 1.5 and 2

parity:

Parity used for serial communication, possible values are `No parity`, `Odd parity`, `Even parity`.

gpib_board:

GPIB board number. Default is 0.

gpib_go_to_local:

Make GPIB instrument automatically go to local after closing. Default is `False`.

tcpip_specify_port:

Use specific TCPIP socket port. Default is `False`.

tcpip_port:

TCPIP socket port. Only relevant if `tcpip_specify_port` is `True`.

12.2. Quantities

All quantities are defined in separate sections, with the name of the quantity given by the section header. The properties of a quantity are defined by a number of keywords, see below for a list the possible options. Only the `datatype` keyword is mandatory, the other ones are optional.

datatype:

The data type should be one of `DOUBLE, BOOLEAN, COMBO, STRING, COMPLEX, VECTOR, VECTOR_COMPLEX, PATH` or `BUTTON`. Only `DOUBLE`, `BOOLEAN` and `COMBO` datatypes can be stepped in a measurement. The `BUTTON` datatype does not have an associated value, and can therefore not be controlled from the *Measurement* program. It is typically used to manually force an instrument to perform a certain task.

label:

Label shown next to control in user interface. If not specified, the label defaults to the name of the quantity.

unit:

Unit for the quantity.

def_value:

Default value.

tooltip:

Tool tip shown when hovering the mouse over the control in the driver GUI.

low_lim:

Lowest allowable value. Defaults to `-INF`.

high_lim:

Highest allowable values. Defaults to `+INF`.

x_name:

X-axis label for a vector data. Only valid if `datatype` is `VECTOR` or `VECTOR_COMPLEX`.

x_unit:

X-axis unit for a vector data. Only valid if `datatype` is `VECTOR` or `VECTOR_COMPLEX`.

combo_def_1, combo_def_2, ...:

Options for a pull-down combo box. Only used when `datatype` is `COMBO`.

group:

Name of the group where the control belongs.

section:

Name of the section where the control belongs.

state_quant:

Quantity that determines this control’s visibility.

state_value_1,state_value_2, ...:

Values of `"state_quant"` for which the control is visible.

model_value_1, model_value_2, ...:

Values of `"model"` for which the control is visible. The value must match one of the models defined in the *Model and Options*-section described above.

option_value_1, option_value_2, ...:

Values of `"option"` for which the control is visible. The value must match one of the options defined in the *Model and Options*-section described above.

permission:

Sets read/writability, options are `BOTH, READ, WRITE` or `NONE`. Default is `BOTH`.

show_in_measurement_dlg:

> This setting is optional. If `True`, the quantity will be automatically shown when adding the instrument to a *Measurement* configuration. This is useful for instrument that contain a lot of quantities, but where most are not likely to be stepped in a measurement.

set_cmd:

> Command used to send data to the instrument. Put "<*>" where the value should appear. If "<*>" does not occur in the string, the value will be added after the command.

get_cmd:

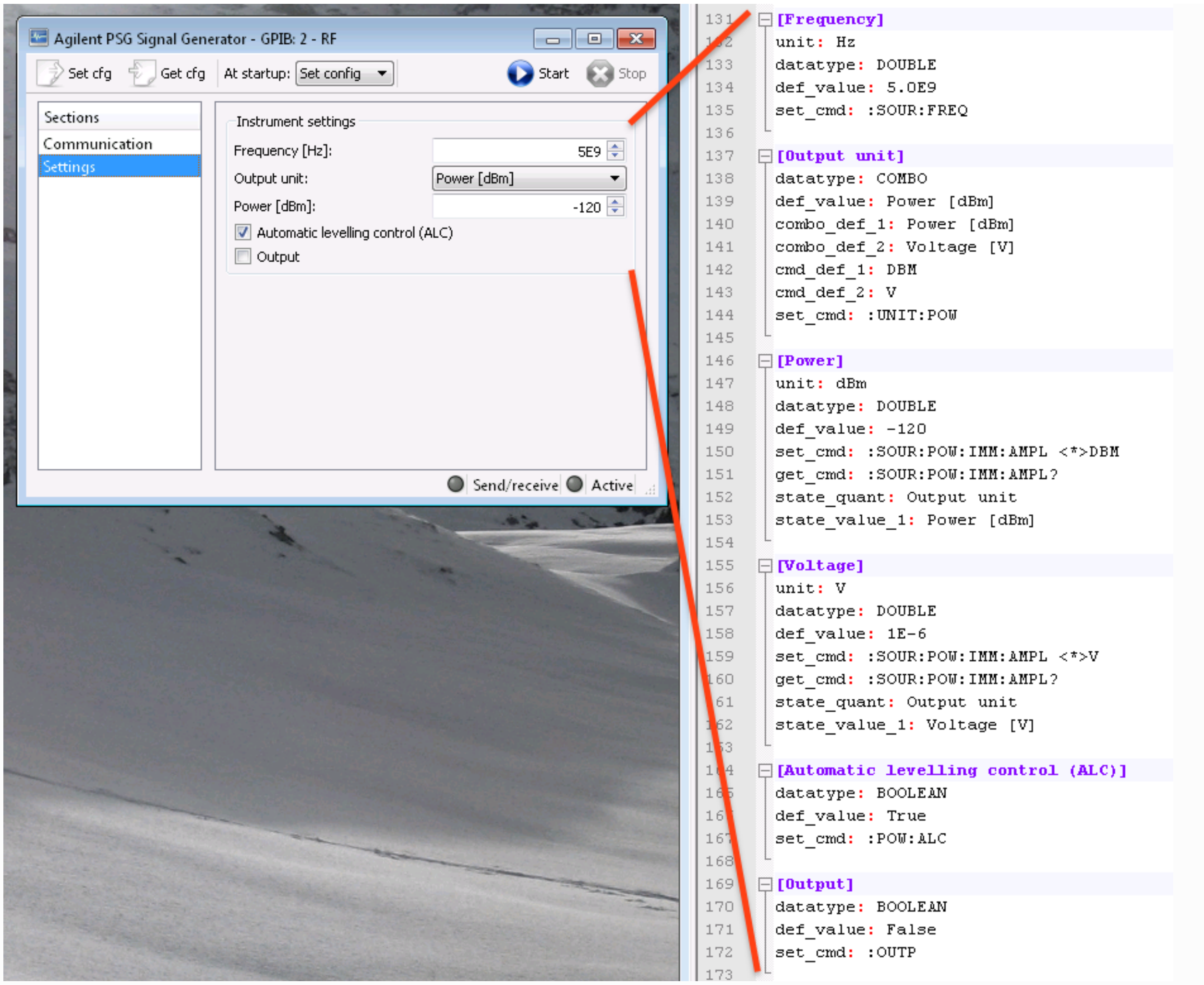
> Command used to get the data from the instrument. Default is `set_cmd?`.

cmd_def_1, cmd_def_2, ...:

> List of strings that define what is sent to/read from an instrument for a quantity that is defined as a a list of multiple options. Only used when `datatype` is `COMBO`.

See Section SweepDriver for a list of extra properties that need to be defined for instruments that support sweeping.

The *Instrument Server* uses the list of quantities to create the controls in the driver dialog window, as shown in Fig. 12.2.



Frequency [Hz]: 5E9
Output unit: Power [dBm]
Power [dBm]: -120
☑ Automatic levelling control (ALC)
☐ Output

```
[Frequency]
unit: Hz
datatype: DOUBLE
def_value: 5.0E9
set_cmd: :SOUR:FREQ

[Output unit]
datatype: COMBO
def_value: Power [dBm]
combo_def_1: Power [dBm]
combo_def_2: Voltage [V]
cmd_def_1: DBM
cmd_def_2: V
set_cmd: :UNIT:POW

[Power]
unit: dBm
datatype: DOUBLE
def_value: -120
set_cmd: :SOUR:POW:IMM:AMPL <*>DBM
get_cmd: :SOUR:POW:IMM:AMPL?
state_quant: Output unit
state_value_1: Power [dBm]

[Voltage]
unit: V
datatype: DOUBLE
def_value: 1E-6
set_cmd: :SOUR:POW:IMM:AMPL <*>V
get_cmd: :SOUR:POW:IMM:AMPL?
state_quant: Output unit
state_value_1: Voltage [V]

[Automatic levelling control (ALC)]
datatype: BOOLEAN
def_value: True
set_cmd: :POW:ALC

[Output]
datatype: BOOLEAN
def_value: False
set_cmd: :OUTP
```

Fig. 12.2. An Instrument driver dialog, shown together with the corresponding instrument definition file.

12.3. Custom drivers - Python code

Custom drivers are required when single-line command strings `get_cmd` and `set_cmd` as defined in the instrument definition file are too simple to read or write a value to an instrument. This is often the case for instruments like network analyzers or oscilloscopes, which contained vector-valued quantities that depend in complicated ways on other settings of the instrument.

The process of creating a custom driver is best described by an example. We are going to create a driver that generates a sinusoid, but without doing any actual instrument communication (the driver will be a *Signal Generator*, as describe in Section Signals). For an example involving instrument communication, see the drivers for one of the network analyzers or oscilloscopes in the *Instrument Drivers* folder.

12.3.1. Creating the driver definition file

Every driver, even the custom ones, require a definition file. We start with the *General settings*-section:

```
[General settings]

# The name is shown in all the configuration windows
name: Simple Signal Generator

# The version string
version: 1.0

# Name of folder containing the code defining a custom driver
driver_path: SimpleSignalGenerator

# Define that the driver is a Signal Generator
signal_generator: True
```

Note that we define the `driver_path`: this signals that there is a custom driver available for this instrument. When starting the driver, the *Instrument Server* will look for the *Python* file *“SimpleSignalGenerator/SimpleSignalGenerator.py”* in the *Instrument Drivers* folder, or for the file *“SimpleSignalGenerator.py”* in the folder where the *INI* configuration file is located. See Section PythonCode below for more information on how to implement the code for custom drivers.

This particular instrument driver does not do any instrument communication and therefore does not have any model or option definitions, so we can skip the *Model and options*-section and the *VISA settings*-section.

Next, we need to define the quantities of the driver. For this example, we want to be able to define the amplitude, frequency and phase of the signal to be generated. In addition, we want to add the option of adding white noise to the signal.

```
[Frequency]
datatype: DOUBLE
unit: Hz
def_value: 10.0

[Amplitude]
```

```
datatype: DOUBLE
unit: V
def_value: 1.0

[Phase]
datatype: DOUBLE
unit: deg
def_value: 0.0

[Add noise]
datatype: BOOLEAN
def_value: False

[Noise amplitude]
datatype: DOUBLE
unit: V
def_value: 0.1
state_quant: Add noise
state_value_1: True

[Signal]
datatype: VECTOR
permission: READ
x_name: Time
x_unit: s
```

Note that the *Noise amplitude* quantity will only be visible if *Add noise* is *True*. The last quantity *("Signal")* represent the signal we want to generate in the *Python* code.
The `permission` of this quantity is set to `READ`, to indicate that this quantity can only be read, not written.

For your convenience, this example driver *INI* definition file and the corresponding *Python* code are available under *Examples* in the *Instrument Drivers* folder.

12.3.2. Implementing the *Python* code

Once the *INI* file has been created, we need to implement the *Python* code that generates the signal. The code should define a subclass of either the `InstrumentDriver.InstrumentWorker` or the `VISA_Driver` class, depending on if the driver will use the *VISA* protocol for communication or not. The `VISA_Driver` class is a subclass of `InstrumentDriver.InstrumentWorker`, and details for how to subclass the `VISA_Driver` is described in Section SubClassVISA below. *Labber* is running Python 3 for all instrument drivers, make sure that all code is Python 3 compatible. See Section PythonDist below for more information about the Python distribution.

The new class should re-implement the four functions `performOpen`, `performClose`, `performSetValue` and `performGetValue`, which are called when an instrument is started, stopped, and called for setting or getting an instrument value, respectively. To describe the procedure, we create a *Python* class for the *Simple Signal Generator*-example shown above:

```python
import InstrumentDriver
import numpy as np

class Driver(InstrumentDriver.InstrumentWorker):
    """ This class implements a simple signal generator driver"""

    def performOpen(self, options={}):
        """Perform the operation of opening the instrument connection"""
        pass

    def performClose(self, bError=False, options={}):
        """Perform the close instrument connection operation"""
        pass
```

As described in the previous section, the *Simple Signal Generator* is only for demonstration purposes and will not involve any actual instrument communication, so we subclass `InstrumentDriver.InstrumentWorker` instead of `VISA_Driver`. In this example, the functions `performOpen` and `performClose` don't do anything.

The code for the more interesting functions `performSetValue` and `performGetValue` follow below:

```python
def performSetValue(self, quant, value, sweepRate=0.0, options={}):
    """Perform the Set Value instrument operation. This function should
    return the actual value set by the instrument"""
    # just return the value
    return value

def performGetValue(self, quant, options={}):
    """Perform the Get Value instrument operation"""
    # proceed depending on quantity
    if quant.name == 'Signal':
        # if asking for signal, start with getting values of other controls
        amp = self.getValue('Amplitude')
        freq = self.getValue('Frequency')
        phase = self.getValue('Phase')
        add_noise = self.getValue('Add noise')
        # calculate time vector from 0 to 1 with 1000 elements
        time = np.linspace(0,1,1000)
        signal = amp * np.sin(freq*time*2*np.pi + phase*np.pi/180.0)
        # add noise
        if add_noise:
            noise_amp = self.getValue('Noise amplitude')
            signal += noise_amp * np.random.randn(len(signal))
        # create trace object that contains timing info
        trace = quant.getTraceDict(signal, t0=0.0, dt=time[1]-time[0])
        # finally, return the trace object
        return trace
    else:
        # for other quantities, just return current value of control
        return quant.getValue()
```

The functions `performSetValue` and `performGetValue` take a `quant` object as a first parameter. The object represents the quantity to be read/set, and all properties of the quantity (as defined in the *INI* configuration file) can be accessed from the object's data members. This is used in the `performGetValue`-function, where the object variable `quant.name` is accessed to find out which quantity to read. See Section quantObj below for more info about the `quant` objects.

The `options` variable present in both `performSetValue` and `performGetValue`-definitions is a *Python* dictionary that contains additional options for setting/getting a value. It is used to provide a way to determine if a driver is called multiple times within a single step of a *Measurement* (see functions `isFirstCall` and `isFinalCall` in the list of driver helper functions in Section driverObj below).

12.3.3. Helper functions for `quant` objects

The `quant` object represents an instrument quantity, and it provides a few helper functions that are useful when writing drivers:

quant.getValue():

The function returns the current value of the quantity. Note that it will just return the local value stored the driver, no instrument communications is performed when calling this function.

quant.getValueIndex(value=None):

The function returns the value as an index number, only useful for quantities with `datatype=COMBO`. If `value=None`, the function will return the local value stored the driver. Note that no instrument communications is performed when calling this function.

quant.setValue(value, rate=None):

The function sets the current value of the quantity. Note that it will just update the local value stored in the driver, no instrument communications is performed when calling this function.

quant.getTraceDict(y, x0=0.0, dx=1.0, x1=None, x=None, logX=False):

Returns a python dictionary containing the numpy array `y`, together with additional x-scale info. The x-scale information can be supplied either as start value and step size `(x0, dx)`, as start and stop values `(x0, x1)`, or as a full vector (input parameter `x`, must have same length as `y`). If using the start/stop notation `(x0, x1)`, it is possible to set `logX` to `True` to create a trace with logarithmic interpolation between the start/stop values. These dictionaries are used to pass waveform data between drivers with vector-valued quantities, like *Signal Generator* and *Signal Analyzers*.

quant.getCmdStringFromValue(value=None):

Convert the input value to a string formatted for sending to the instrument. If the input parameter `value` is `None`, the current value is used.

quant.getValueFromCmdString(sValue):

Inspect the input string `sValue` coming from the instrument and return a numerical value.

12.3.4. Helper functions for `driver` objects

The base driver object `InstrumentDriver.InstrumentWorker` provides the following helper functions. The `options` variable present in the functions `isFirstCall` and `isFinalCall` is a *Python* dictionary with additional options that is passed to the `performSetValue` and `performGetValue`-functions when calling the driver from outside.

getName():

Return name of instrument, as defined in the user-interface dialog.

getInterface():

Return instrument interface, as defined in the dialog. The interface type is one of `GPIB`, `TCPIP`, `USB`, `Serial`, `VISA`, `Other`, `None`.

getAddress():

Return address of instrument, as defined in the user-interface dialog. This is function can be used to determine when opening communication to an instrument.

getCommunicationCfg():

Return communication configuration as a dictionary, with the following keys: `Timeout`, `Term. character`, `Send end on write`, `Suppress end bit termination on read`, `Baud rate`, `Data bits`, `Stop bits` and `Parity`. The configuration items are described in Section CommunicationCfg above.

getValue(quant_name):

The function is used to access the current local value of any quantity of the driver. The function is used repeatedly in the example above for getting the amplitude, frequency and phase when creating the sinusoid.

getValueArray(quant_name):

Same as above, but will return current value as a numpy array instead if the quantity is vector-valued. Otherwise, it'll return an empty numpy array.

getValueIndex(quant_name):

Get value of quantity as numerical index. Only useful for quantities with `datatype=COMBO`.

getCmdStringFromValue(quant_name):

Get command string for current value of quantity with name `quant_name`. See `quant.getCmdStringFromValue` in section above for more info.

setValue(quant_name, value, sweepRate=None):

The function is used to set the local value of any quantity of the driver. No hardware communication will take place; to actual set the instrument value, use the function `sendValueToOther` defined below.

readValueFromOther(quant_name, options={}):

The function will read the value of another quantity from the instrument. In contrast to the `getValue` mentioned above, this function will perform actual hardware communication to retrieve the current value from the instrument. The function will return the updated value.

sendValueToOther(quant_name, value, sweep_rate=0.0, options={}):

The function will communicate with the hardware to update the value to another quantity of the instrument. The function will return the updated value.

getModel():

Get model string.

setModel(model_name):

Set model string.

getOptions():

Get list of strings describing installed options.

setInstalledOptions(list_of_options):

Set list of strings describing installed options.

isConfigUpdated(bReset=True):

Returns true if any non-read-only quantity of the instrument has been updated since the last call to this function where `bReset` was `True`.

isFirstCall(options):

If a driver is used in a *Measurement* and there are multiple quantities of that driver that will be updated within a single step, it can be advantageous to delay outputting data to an instrument until all local driver quantities have been updated. This function returns `True` if the current call is the first one within the current measurement step.

isFinalCall(options):

Same as above, but returns `True` if the current call is the last one.

isStopped():

Return `True` if the user stopped the measurement. If the instrument communication is expected to take a long time, it's recommended to periodically call this function to ensure that the driver remains responsive to user interaction.

isHardwareTrig(options):

Return `True` if the caller is in hardware trig mode.

isHardwareLoop(options):

Return `True` if the caller is in hardware loop mode.

getHardwareLoopIndex(options):

Get the current hardware loop index. The function returns a tuple (`index`, `n_pts`), where `index` is the index for the current call, and `n_pts` is the total number of points of the hardware loop.

log(message, level=20):

Log a message to the instrument logger. The log level is an integer ranging from 30 (warning, always shown) to 10 (debug, only shown in debug mode).

wait(wait_time=0.05):

Pause execution and put the process to sleep for the given time (in seconds).

getValueFromUserDialog(value=None, text='Enter value:', title='User input'):

Show user interface dialog to ask the user for an input value. The function returns the value entered by the user.

reportStatus(message):

Report status update to the *Instrument Server* and connected clients. The argument `message` should be a string.

reportProgress(quant, progress):

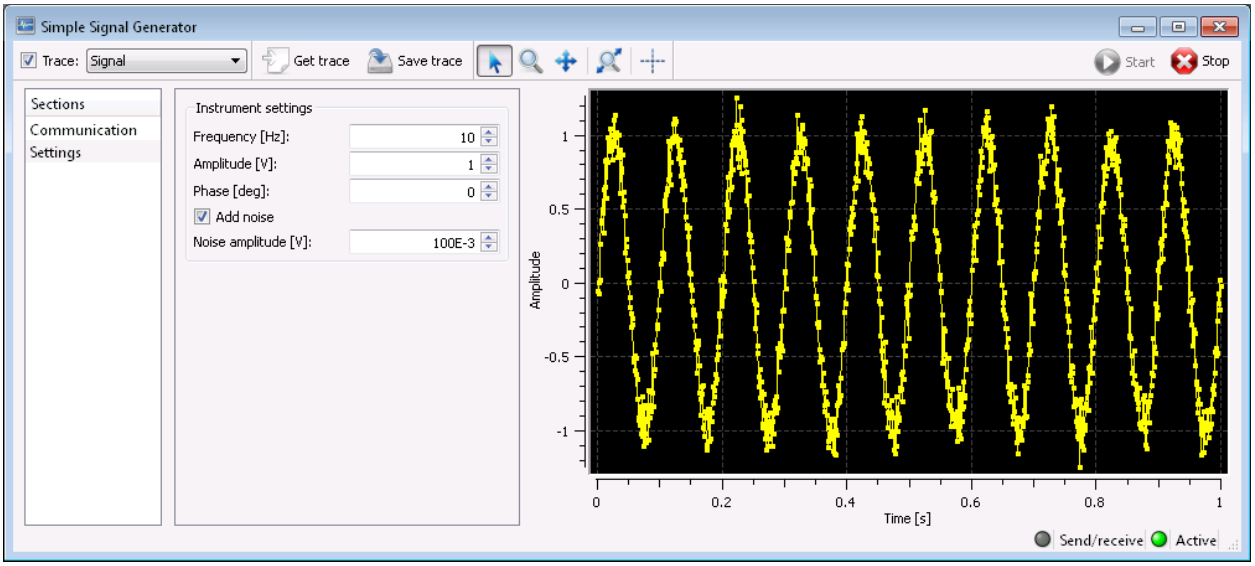
Report progress update when setting/getting the value of the quantity `quant` to the *Instrument Server* and connected clients. The argument `progress` should be a floating point value between 0.0 and 1.0. The function is used to provide feedback to the user when performing slow instrument operations, for example when sweeping a magnetic field.

reportCurrentValue(quant, value):

Report current value of the quantity `quant` to the *Instrument Server* and connected clients. The function is used to provide feedback to the user when performing slow instrument operations, for example when sweeping a magnetic field.

12.3.5. Testing the driver

As stated previously, this example driver *INI* definition file and the corresponding *Python* code are available under *Examples* in the *Instrument Drivers* folder. To test the driver, move the *INI* file and the folder with the *Python* code to reside directly in the *Instrument Drivers* folder. Next, start the *Instrument Server* and add a new instrument. If the driver is defined properly, the *Simple Signal Generator* should show up in the instrument driver list. Select the new driver and although the driver doesn't perform any instrument communication, we still need to provide an address.
Select *"Other"* under *"Interface"* and type any string in the *"Address"* text box. This to ensure that every instrument has a unique address so that the *Instrument Server* can access multiple instances of the *Simple Signal Generator*, if needed. Finally, click *"OK"* to close the dialog.



Instrument settings

Frequency [Hz]: 10

Amplitude [V]: 1

Phase [deg]: 0

☑ Add noise

Noise amplitude [V]: 100E-3

Fig. 12.3. The example instrument driver Simple Signal Generator.

The new instrument should appear in the main *Instrument Server* list. Double-click the instrument name will bring up the driver configuration window, as shown in Fig. 12.3. To test the code, start the driver with the *"Start"* button and make sure the *"Trace"*-checkbox is checked to view the sinusoid. To control parameters while the driver is running, either just update one of the controls, or go to the server window, expand the *"Simple Signal Generator"*-item in the instrument list and use the *"Set Value"*-button to set a new value. If the *"Update continuously"*-control in the driver dialog is checked, you will see the trace change in real time as parameters are modified.

12.4. Subclassing the *VISA* driver

The previous example was a little bit unusual, since no actual instrument communication was performed in the driver. A more common situation would be where most communication can be handled with simple text-based commands as defined in the driver *INI* configuration file, but where a few advanced quantities need special functionality. For these cases, the easiest way to proceed is to subclass the `VISA_Driver` and only re-implement code for the special cases. The code below shows an example of what such a driver would look like:

```
from VISA_Driver import VISA_Driver

class Driver(VISA_Driver):
    """ This class re-implements the VISA driver"""

    def performOpen(self, options={}):
        """Perform the operation of opening the instrument connection"""
        # calling the generic VISA open to make sure we have a connection
        VISA_Driver.performOpen(self, options=options)
        # do additional initialization code here...
        pass

    def performClose(self, bError=False, options={}):
        """Perform the close instrument connection operation"""
        # calling the generic VISA class to close communication
        VISA_Driver.performClose(self, bError, options=options)
        # do additional cleaning up code here...
        pass

    def performSetValue(self, quant, value, sweepRate=0.0, options={}):
        """Perform the Set Value instrument operation. This function should
        return the actual value set by the instrument"""
        # check quantity name
        if quant.name == 'Some_Special_Operation':
            # special case, perform special code to set value
            pass
        else:
            # otherwise, call standard VISA case
            value = VISA_Driver.performSetValue(self, quant, value, sweepRate,
options)
        return value

    def performGetValue(self, quant, options={}):
        """Perform the Get Value instrument operation"""
        # check quantity name
        if quant.name == 'Some_Special_Operation':
            # special case, perform special code to get value
            value = 0.0
        else:
            # for all other cases, call generic VISA driver
            value = VISA_Driver.performGetValue(self, quant, options)
        return value
```

Note that both the `performOpen` and `performClose` functions have to call the generic VISA class, to make sure that the communication is properly initiated.

12.4.1. Helper functions for drivers subclassing the *VISA* driver

In addition to the helper functions provided by the generic driver object (described in Section driverObj above), the `VISA_Driver` provides the following helper function:

writeAndLog(sCmd, bCheckError=True):

The function will send the command string `sCmd` to the instrument.
If `bCheckError` is *True*, an error check is performed after the command has been sent.

write(sCmd, bCheckError=True):

Same as above, but no entry will be created in the *Instrument Log*, regardless of the log level. See Section logs for more information about the logging feature.

write_raw(data):

Write raw data bytes to the instrument, without interpreting termination characters, etc. No logging is performed.

reply = askAndLog(sCmd, bCheckError=True):

The function will send the command string `sCmd` to the instrument and wait for a reply. The reply is returned as a text string. If `bCheckError` is *True*, an error check is performed after the command has been sent and data has been received.

reply = ask(sCmd, bCheckError=True):

Same as above, but no entry will be created in the *Instrument Log*, regardless of the log level. See Section logs for more information about the logging feature.

reply = read(n_bytes=None, ignore_termination=False):

Read a total of `n_bytes` from the device, ignoring any termination characters.
If `n_bytes` is `None`, the complete buffer is read. If `ignore_termination` is set to `True`, the program will not check for or remove termination characters.

queryErrors():

Check for instrument errors by checking the event status register (`*ESR?`). An exception is raised if the instrument reports an error. The check only takes place if the item `query_instr_errors` in the VISA settings of the *INI* file is set to `True`.

12.5. Support for sweeping

Some quantities, for example the B-field of a magnet, require the output to be changed with a well-defined sweep rate whenever the value is updated. *Labber* provides supports for swept quantities, but such drivers require a few extra configuration settings compared to standard drivers. The extra setting are described in the subsection below. For more information on how swept experiments are implemented in the *Measurement Setup* dialog, see Section SweepModeSetup.

12.5.1. Sweeping - Driver definition file

In addition to the properties listed in in Section Quantities, the driver *INI*-file of an instrument that supports sweeping needs to define the following properties for a sweepable quantity:

sweep_cmd:

Command used to sweep data. Use “<sr>” for sweep rate or “<st>” for sweep time, and “<*>” for the value. Note that sweep rate will be defined in terms of change per second or change per minute, as set by the `sweep_minute`-setting defined below. If the instrument does not have a built-in command for sweeping, a similar effect can be achieved by repeatedly using the `set_cmd` to incrementally change the instrument value. To enable this feature, set `sweep_cmd` to `"***REPEAT SET***"`, followed by the time interval between setting values (in seconds). If no time interval is defined, default is 0.1 seconds.

sweep_check_cmd:

Command used to check if the instrument is currently in sweep mode. The instrument should return `True` or `1` if the instrument is sweeping towards a value. If `sweep_check_cmd` is not defined, the program will determine if an instrument is in sweep mode by continuously reading the current value and comparing it against the target value with resolution `sweep_res`, as defined below.

sweep_res:

Attainable resolution when sweeping an instrument, in absolute units. Default value is 10^{-10}, to avoid float rounding errors. This parameter is not used if the `sweep_check_cmd` is defined.

stop_cmd:

Command used to stop a sweep.

sweep_rate:

Default sweep rate, in rate per second or rate per minute (as set by the `sweep_minute` parameter defined below). If this value is non-zero, sweeping will be turned on automatically for this quantity. Default value is `0`.

sweep_minute:

If `True`, sweep rates are defined in terms of value rate per minute, otherwise in rate per second. Default is `False` (rate per second).

sweep_rate_low:

Minimal sweep rate. Default is `0`.

sweep_rate_high:

Maximal sweep rate. Default is `+Inf`

Note that the existence of the `sweep_cmd`-parameter defines whether a quantity is sweepable or not. If a quantity is sweepable, the *Instrument Server*, *Instrument Driver* and the *Measurement Setup* configuration dialogs will contain a few extra options for controlling the sweep rates. If the `sweep_cmd`-parameter is defined but the `set_cmd`-parameter is not, the driver will not allow direct setting of output values. This is useful for instruments like magnets, whose output currents must always be swept at a certain rate.

12.5.2. Sweeping - Python code

If the sweeping functionality of a driver cannot be implemented using the built-in functionality based on the parameters in the driver definition file listed above, it is possible to write custom *Python* code for carrying out the sweeping. To begin with, the `performSetValue`-function for setting an instrument value (described in Section PythonCode and Section SubClassVISA above) needs to be implemented to support sweeping:

def performSetValue(self, quant, value, sweepRate=0.0, options={}):

When re-implementing the `performSetValue`-function for a swept quantity, it is important that the code inspects the `sweepRate` parameter to see if the user wants to set the value directly (`sweepRate=0.0`), or perform sweeping (`sweepRate>0.0`). Note that in sweep mode (`sweepRate>0.0`), the function should not wait for the sweep to finish, since the sweep checking/waiting is handled by the *Instrument Server*. The `sweepRate` parameter is defined in terms of change per second or change per minute, as set by the `sweep_minute` configuration parameter defined in the section above.

In addition to the four standard functions `performOpen`, `performClose`, `performSetValue` and `performGetValue` described in Section PythonCode, drivers that support sweeping may also re-implement the following functions:

def checkIfSweeping(self, quant, options={}):

The function should return `True` if the instrument is currently sweeping to the target value. The standard implementation will either send the `sweep_check_cmd` to the instrument or continuously read the current value and compare to the target, as described in Section SweepDriver above.

def performStopSweep(self, quant, options={}):

This function should stop the current sweep. The default implementation will send the `stop_cmd` to the instrument, as described in Section SweepDriver above.

12.6. Controller drivers

To make a controller driver, start by setting the key `controller` in the driver configuration file to `True`. This will make *Labber* automatically add a few quantities such as *Period* and *Input/output* signals for handling the controller operation. Note that these quantities will be added automatically, and shall not be included in the driver `.ini` file.

For the controller to operate properly, the driver `.py` file must implement the `performGetValue` for the *Output value*-quantity. The function should typically read the value of the *Input value* control, and then apply the proper control logic to generate the output value. For an example of controller driver, see the *PID Controller* driver provided with *Labber*.

12.7. Hardware arming and triggering

In hardware trigger mode, log instrument will be armed to wait for a hardware trigger before starting to acquire data. The function `isHardwareTrig(options)` can be used by both instruments outputting and instruments reading values to check if the measurement is in hardware trig mode. Instruments that supports hardware arming need to define the `support_arm` parameter in the *General settings* of the driver definition file (see Section DriverINIGeneral above), and implement the following function:

def performArm(quant_names, options={}):

> The function should arm the instrument, to make it ready to acquire values for the list of quantities defined by **`quant_names`**.

The function `performArm` is called before issuing the trigger starting the measurement. See Section HardwareTrig for more information how hardware triggering is configured in the *Measurement* program.

12.8. Hardware looping

Some instruments can perform looping of values within the instrument hardware. This allows for implementing more efficient looping than with a computer, since there will be no need for the computer to send new values to the instrument at each step value. For a more detailed description of how hardware looping works and how it is configured in the *Measurement* program, see Section HardwareLoop.

Hardware looping requires that both the instrument outputting and the instrument reading values support hardware looping, and that the instrument reading values supports hardware arming, as defined by the `support_hardware_loop` and `support_arm` parameters in the *General settings* of the driver definition file (see Section DriverINIGeneral above).

12.8.1. Hardware looping - outputting values

In hardware looping mode, the `performSetValue` function will be called n times for a step sequence containing n points. At the final call, the instrument should be configured to start outputting values when a trigger is issued. The function `isHardwareLoop(options)` can be used to check if the measurement is in hardware loop mode, and the function `(index, n_pts) = getHardwareLoopIndex(options)` can be used to get the current hardware loop index and the total number of point `n_pts`.

12.8.2. Hardware looping - reading values

In addition to defining the `support_hardware_loop` and the `support_arm` parameters in the driver definition file, the driver *Python* file needs to implement the `performArm` function for arming the instrument to acquire multiple values. The number of values to expect is given by the output `n_pts` of the function `(index, n_pts) = getHardwareLoopIndex(options)`. After the instrument has been armed and a trigger has been sent, the function `performGetValue` will be called multiple times to acquire the results.

In the same way as for instruments outputting values, the functions `isHardwareLoop(options)` and `(index, n_pts) = getHardwareLoopIndex(options)` can be used to check if the measurement is in hardware loop mode, and to get the current hardware loop index and the total number of point `n_pts`, respectively.

12.9. Python distribution

When starting an instrument driver, *Labber* will launch a dedicated driver process and execute its Python code in the new process. By default, *Labber* will use the default, built-in Python distribution, which currently is a 64-bit version of Python 3.6. However, this may change to a newer Python version in a future release of *Labber*.

12.9.1. Python distribution, 32-bit version

For compatibility reasons, *Labber* is also shipped with a 32-bit version of the same Python distribution (Windows only), to allow control of older instruments for which only 32-bit Windows DLLs/drivers are available.

To activate the 32-bit Python version for a specific driver, open the driver's configuration window in the *Instrument Server*, go to the *"Communication"*-section, click *"Show advanced interface settings"*, and check the *"Run in 32-bit mode"*-box prior to starting the instrument driver. Note that each instance of an instrument driver is running in its own, separate process, which makes it is possible to have some drivers run in 32-bit mode, while others are running in 64-bit mode. To set an instrument's default setting to run in 32-bit mode, use the `use_32bit_mode`-flag described in Section DriverINIGeneral above.

12.9.2. External Python distribution

Sometimes it is convenient to use an external Python distribution instead of *Labber*'s built-in one. For example, a driver may be relying on a number of external Python packages , and it is convenient to install/update those packages using a Python package manager instead of manually copying them into the folder location of the *Labber* driver.

To use an external Python distribution, open *Labber's Preferences* dialog, go to the *"Advanced"*-section and point the *"Python distribution"* control to the file representing the `python` executable for a given Python environment. The Python environment must be running Python 3.5 or later, and it is recommended to use the *Anaconda/miniconda* package manager for configuring the environment. For *Anaconda/miniconda* distributions, the `python` executable is located directly in the root of each environment folder (`"pythonw.exe"`, Windows), or under `"bin/python"` (MacOS version). Note that the external Python distribution will be used for all instrument drivers, except for the ones running in 32-bit mode (Windows only).

The external Python distribution must contain the following packages:

- numpy
- scipy
- h5py
- pycrypto
- future
- pyvisa

To set up a valid environment using *Anaconda/miniconda*, run the following command from the command line:

```
conda install pip numpy scipy h5py pycrypto future scikit-learn dill
pip install pyvisa qtpy scikit-optimize
```

For more information about the *Anaconda/miniconda* Python distributions, see `https://www.continuum.io/`.

12.9.3. Troubleshooting, external Python distribution

If a driver process terminates immediately upon starting or if a dialog pops up with a *"Broken pipe"*-message, there are most likely missing packages in the external Python distribution. To find out which package is missing, quit the *Instrument Server* and restart it from a terminal window to get access to the standard error output.

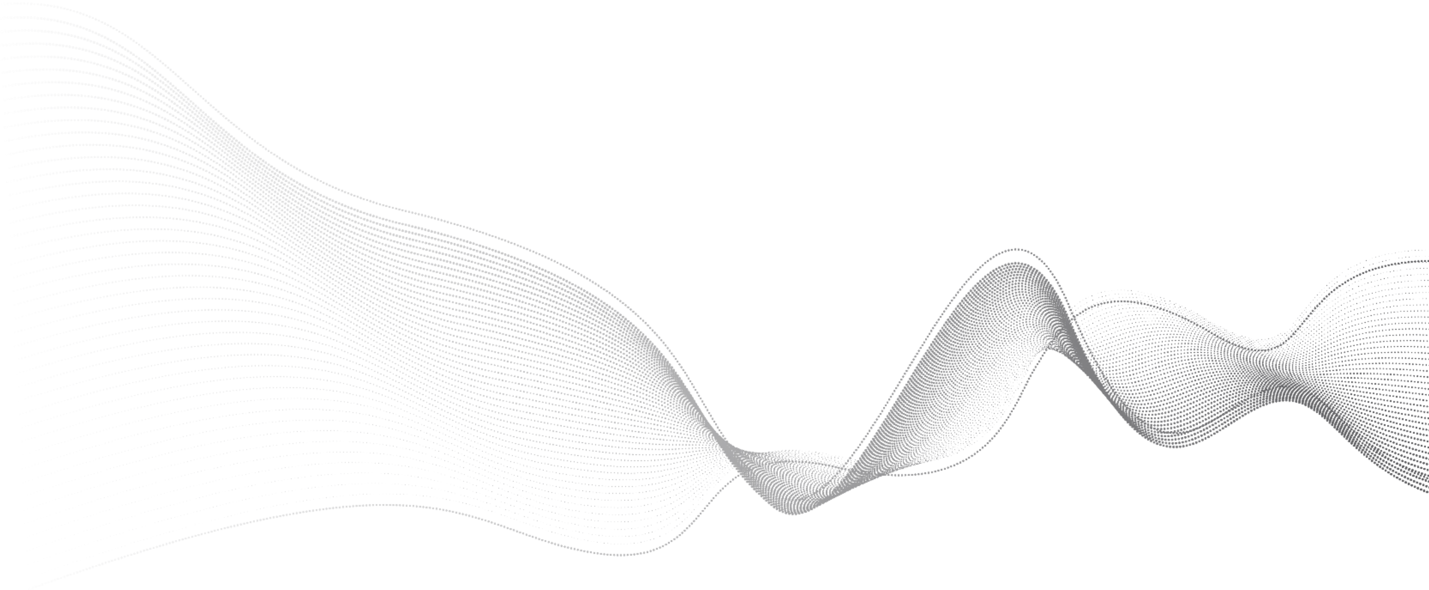
- For Windows, open a terminal window, `cd` to the location of the *Instrument Server* application, the run the application `InstrumentServer-Console.exe` from the command line. In addition, the *"Python distribution"* variable mentioned in Section PythonDistExternal above should to point to the file `"python.exe"` instead of `"pythonw.exe"`.
- For macOS, open a terminal window, `cd` into the location

  `/Applications/Labber/InstrumentServer.app/Contents/MacOS`, then run the *Instrument Server* by typing `./InstrumentServer` in the terminal window.

- On Windows, there have been reports of incompatibilities with certain versions of Anaconda. Anaconda3 v. 4.4.0 is the most recent working version to have been tested to work.

Labber

APPENDIX: PYTHON API



Python API

The *Labber* Python API (application program interface) provides Python classes and functions for controlling instruments in the *Labber* Instrument Server, for reading and writing *Labber* log files, and for scripting *Labber* measurements.

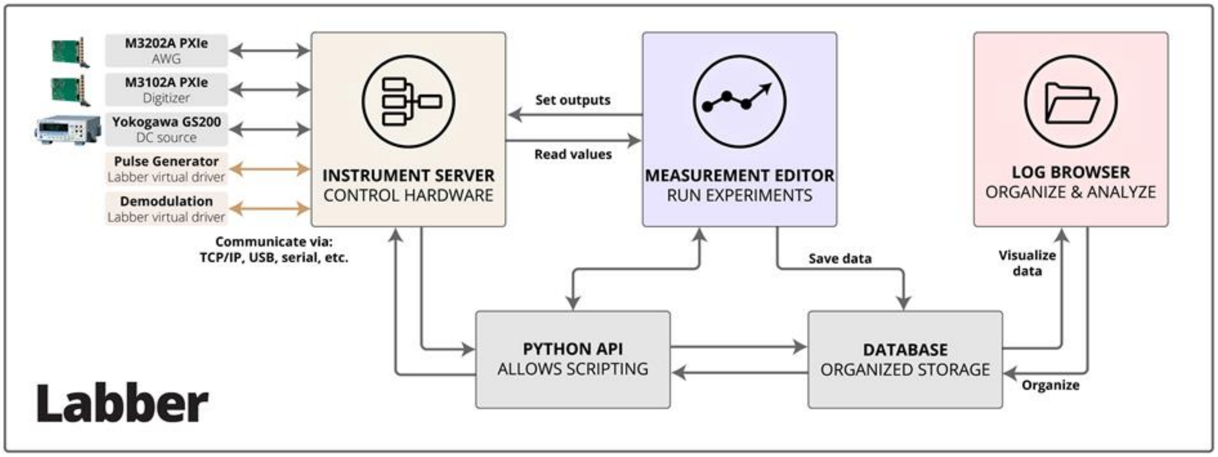


Fig A1. Overview and structure of the components in the Labber software package, including the Python API.

A1. Installation

The API is included when installing Labber, and the files are located in the *"Script"* folder of the main program directory. To access the API, add the *"Script"* folder to your Python path.

A1.1. Requirements

The Labber API requires the following Python packages:

- Python 2.7, or Python 3.4 or later
- NumPy
- h5py
- PyQt4 or PyQt5
- qtpy
- msgpack
- sip
- future

A1.2. Testing the API

To test the Labber API installation, execute the following code in a Python console:

```python
import Labber
print (Labber.version)
```

A1.3. Upgrading from earlier versions

Note that in Labber version 1.1 and later, the *ScriptTools* module has been moved into the `Labber` module. To make scripts written for older versions of Labber work with version 1.1 and later, replace `import ScriptTools` with `from Labber import ScriptTools`.

A2. Instrument server

The instrument server API provides functions that allow Python to communicate with the Labber Instrument server. The API provides functionality both for communicate and control instruments as well as scheduling measurements using the Instrument server's built-in scheduler.

A2.1. Labber client

The instrument control API uses a client object to communicate with the Instrument server. To initialize the client, use the `connectToServer` function. The following example will connect to an instrument server on the local computer and list all available instruments.

```
import Labber
# connect to server
client = Labber.connectToServer('localhost')
# get list of instruments
instruments = client.getListOfInstrumentsString()
for instr in instruments:
    print(instr)
# close connection
client.close()
```

A2.2. Scheduling measurements

The function `schedule_measurement` is used to schedule a measurement using the Instrument server's queueing system. For measurement that are scheduled to run immediately, the function will wait until the measurement has finished, and return the full path of the final measurement file. For measurement that are scheduled for the future, the function will return directly without waiting.

Note that the server timeout should be set to `None`, to allow the client to wait for a long time for the measurement to finish before timing out. The following example illustrates the procedure.

```
import Labber
# connect to server
client = Labber.connectToServer('localhost', timeout=None)
# schedule experiment, wait to finish
output_file = client.schedule_measurement('~/Desktop/Test.hdf5')
print('Final output file:', output_file)
# close connection
client.close()
```

A2.3. Connecting to instruments

To set or read values from an individual instrument, first use the function `connectToInstrument` of the Labber client object to access the instrument. The function will return an instance of a InstrumentClient object, from which the entire configuration or individual quantities of the instrument can be set or read.

The following example will first connect to a Labber Instrument server, and then connect to a dc voltage source and a voltmeter using a GPIB interface. Next, it will output voltages with the dc source, and measure the corresponding response with the voltmeter.

```python
import Labber
import time, numpy as np
# connect to server
client = Labber.connectToServer('localhost')
# connect to specific instruments
volt_source = client.connectToInstrument('Yokogawa 7651 DC Source',
              dict(interface='GPIB', address='6'))
volt_meter = client.connectToInstrument('Agilent 34401 Multimeter',
             dict(interface='GPIB', address='1'))

# start drivers
volt_meter.startInstrument()
volt_source.startInstrument()

# put zero voltage to source and turn on the output
volt_source.setValue('Voltage', 0.0)
volt_source.setValue('Output', True)

# do a loop from zero to one volt
for volt in np.linspace(0.0, 1.0, 11):
    # set value to voltage source and wait
    volt_source.setValue('Voltage', volt)
    time.sleep(0.2)
    # read value from voltmeter
    measVolt = volt_meter.getValue('Voltage')
    # print result
    print('Set value: %.2f V, measured value: %.2f V' % (volt, measVolt))

# close client
client.close()
```

See the reference documentation below for more information on functions available in the InstrumentClient class.

A2.4. Blocking vs. non-blocking clients

Labber supports two types of clients. The standard client type discussed so far is a *blocking* client, which will block program execution while waiting for a response from the server. Blocking clients are typically used in scripts where a number of instrument values are set or read in a pre-defined sequential order.

Function calls to a *non-blocking* client, on the other hand, will not wait for a response from the instrument server. Instead, the client uses a system of callback functions to notify the program that a new instrument value has been set or read. The advantage of a non-blocking client is that the main program thread will not be blocked while waiting for a result, and that multiple instrument operations can be performed in parallel. Non-blocking clients are typically used in user-interface driven applications, where the dialogs and user-interface elements need to remain responsive.

To create a non-blocking client, use the same `connectToServer`-function described above but with the argument `wait_for_reply` set to `False`. The following script will perform the same procedure of setting and reading voltages as the blocking-client script shown above, but it will do it in a non-blocking framework. Instead of directly returning values, the `connectToServer`, `connectToInstrument`, `setValue` and `getValue` functions will use *callback* functions to retrieve values.

Note that the event handling of the non-blocking client and the callback functionality is handled by the Qt framework.

```
import Labber
from qtpy.QtCore import QCoreApplication
import functools
import numpy as np

class TestNonBlockClient():
    def __init__(self):
        # keep track of instrument references
        self.dInstr = dict()

    def connect(self):
        # connect to server, call 'connectionOpen' upon completion
        self.client = Labber.connectToServer('localhost', wait_for_reply=False,
                      callback_open=self.connectionOpen,
                      callback_network_error=self.on_error,
                      callback_instrument_error=self.on_error)

    def connectionOpen(self, data):
        # callback after connection has been established
        print('Connection open')
        self.nStart = 0
        # connect to instruments, call 'connected' upon completion
        newCallback = functools.partial(self.connected, 'Agilent 34401')
        self.client.connectToInstrument('Agilent 34401 Multimeter',
                                        dict(address='1', interface='GPIB'),
                                        callback=newCallback)
        newCallback = functools.partial(self.connected, 'Yokogawa 7651')
        self.client.connectToInstrument('Yokogawa 7651 DC Source',
                                        dict(address='3', interface='GPIB'),
                                        callback=newCallback)

    def on_error(self, message):
        # print error
        print('Error: %s.\n\n' % message)

    def connected(self, sHardware, instr):
        # keep track of instruments
        self.dInstr[sHardware] = instr
        # start driver, call 'started' upon completion
        newCallback = functools.partial(self.started, sHardware)
        instr.startInstrument(callback=newCallback)

    def started(self, sHardware, data):
        # check that both instruments have been started
        self.nStart += 1
        if self.nStart==2:
            self.yoko = self.dInstr['Yokogawa 7651']
            self.volt = self.dInstr['Agilent 34401']
            # set yoko loop
            self.lLoop = np.linspace(0.0, 1.0, 11)
            self.nIndex = -1
            # start loop
            self.loop()

    def loop(self):
        # check if looping is done
        self.nIndex += 1
        if self.nIndex < len(self.lLoop):
            # keep looping
            self.yoko.setValue('Value', self.lLoop[self.nIndex],
                               callback=self.stepDone)
        else:
            print('Finished!')

    def stepDone(self, data):
        # voltage has been set, read response
        self.yoko.getValue('Value', callback=self.logDone)

    def logDone(self, measVolt):
        # print result
        volt = self.lLoop[self.nIndex]
        print('Set value: %.2f V, measured: %.2f V' % (volt, measVolt))
        # keep looping
        self.loop()


if __name__ == '__main__':
    # start Qt event loop
    app = QCoreApplication([])
    # create test object and connect to server
    test = TestNonBlockClient()
    test.connect()
```

A2.5. Function definitions

`Labber.connectToServer`*(address='localhost', wait_for_reply=True, port=None, timeout=10, callback_open=None, callback_network_error=None, callback_instrument_error=None, binary_transfer_format=None)*

Connect to Labber Instrument server and return a Labber client object.

There are two version of Labber clients, blocking and non-blocking ones. Blocking clients will wait for the instrument server to send a reply before returning, whereas non-blocking client will return immediately and call callback functions once the values are available.

Parameters

- **address** (*str*, *optional*) – IP address of Labber Instrument server. Default is localhost.
- **wait_for_reply** (*bool*, *optional*) – If True, the function will return a blocking client. Default is True.
- **port** (*int*, *optional*) – Port number for server communication. Default is 9406.
- **timeout** (*int or float*, *optional*) – Longest time to wait for the server to reply. Default is 10 seconds.
- **callback_open** (*function*, *optional*) – Callback function called after communication has been established. The function should have a single boolean argument, which will state if the connection was successful or not. Only relevant if *wait_for_reply* is False, ie for non-blocking clients.
- **callback_network_error** (*function*, *optional*) – Callback function called in case of network error. The function should take a single argument that will contain the error message. Only relevant if *wait_for_reply* is False, ie for non-blocking clients.
- **callback_instrument_error** (*function*, *optional*) – Callback function called in case of instrument error. The function should take a single argument that will contain the error message. Only relevant if *wait_for_reply* is False, ie for non-blocking clients.

- **binary_transfer_format** (*bool*, *optional*) – If True, data between the client and the server is sent as binary data instead of text. The value must match the Data transfer format in the Labber Instrument server preferences. Default is True.

Returns

client – Labber client object, either blocking or non-blocking version.

Return type

Client object

Examples

Open connection to server and list connected instruments.

```
>>> import Labber
>>> client = Labber.connectToServer('localhost')
>>> instruments = client.getListOfInstrumentsString()
>>> print(instruments)
>>> client.close()
```

A2.6. Class definitions

A2.6.1. Blocking client

The client object should not be initialized directly. Instead, use the connectToServer()-function defined above.

class`Labber.LabberBlockingClient`*(sAddress='localhost', port=9406, timeout=10, binary_transfer_format=None, convert_to_unicode_if_py2=True)*

Bases: `object`

Labber client, blocking execution while waiting for server response.

Parameters

- **sAddress** (*str*, *optional*) – IP address of Labber Instrument server. Default is localhost.
- **port** (*int*, *optional*) – Port number for server communication. Default is 9406.
- **timeout** (*int or float*, *optional*) – Longest time to wait for the server to reply. Default is 10 seconds.
- **binary_transfer_format** (*bool*, *optional*) – If True, data between the client and the server is sent as binary data instead of text. Default is True.

`close`()

Close the connection to the server.

`connectToInstrument`(*sHardware*, *dComCfg*, *bCreateNew=False*)

Connect to an instrument object on the instrument server.

Parameters

- **sHardware** (*str*) – Name of instrument hardware to connect to.
- **dComCfg** (*dict*) –

  Dictionary describing the communication address of the instrument. Either the *name* key or the *interface*`+`*address* keys must be defined. The dictionary is defined by the following keys:

Namestr

Name of instrument.

interface{'GPIB', 'TCPIP', 'USB', 'Serial', 'VISA', 'Other', 'None'}

Communication interface.

addressstr

Instrument address string

startup{'Set config', 'Get config', 'Do nothing'}

Operation to perform at instrument startup.

lockbool

If True, instrument will be locked while in use.

- **bCreateNew** (*bool*, *optional*) – If True, a new instrument will be created if the requested one is not already present. Default is False.

Returns

instr – Object representing an instrument on the Labber instrument server.

Return type

InstrumentClient object

`createInstrument`(*sHardware*, *dComCfg*)

Create an instrument on the instrument server.

Parameters

- **sHardware** (*str*) – Name of instrument hardware to connect to.

- **dCComCfg** (*dict*) –

  Dictionary describing the communication address of the instrument. The dictionary is defined by the following keys:

namestr

Name of instrument.

interface{'GPIB', 'TCPIP', 'USB', 'Serial', 'VISA', 'Other', 'None'}

Communication interface.

addressstr

Instrument address string.

startup{'Set config', 'Get config', 'Do nothing'}

Operation to perform at instrument startup.

lockbool

If True, instrument will be locked while in use.

`getListOfInstruments()`

Get a list of instruments present on the Labber instrument server.

Returns

instruments – List of instruments on the server. Each element of the list is a two-element tuple (name, comcfg), where *name* is the hardware name and *comcfg* is a dict with communication settings.

Return type

list of tuple

`getListOfInstrumentsString()`

Get a list of instruments present on the Labber instrument server.

Returns

instruments – List of strings describing instruments on the server.

Return type

list of str

`schedule_measurement`(*path_to_configuration*, *output_path=None*, *priority=False*, *scheduled=None*, *period=None*, *command_args=[]*)

Schedule measurement using the instrument server queueing system.

For measurement that are scheduled to run immediately, the function will wait until the measurement has finished, and return the full path of the final measurement file. For measurement that are scheduled for the future, the function will return None directly without waiting.

Parameters

- **path_to_configuration** (*str*) – Path to Labber measurement configuration to run, saved in either .labber, .json or .hdf5 format.
- **output_path** (*str*, *optional*) – Path for output measurement file. Default is None, in which case the resulting output file is put in the Labber database.
- **priority** (*bool*, *optional*) – Priority in scheduling system. Default is False.
- **scheduled** (*float*, *optional*) – Scheduled time for measurement to run, in number of seconds passed since epoch. Default is None, which schedules immediately.
- **period** (*float*, *optional*) – Periodicity of measurement, measured in seconds. Default is None, in which case the measurement will only run once.
- **command_args** (*list*, *optional*) – Command-line arguments to pass on to the Measurement engine. Only used if scheduled and period are None.

Returns

Path of output file if scheduled is None and period is None, else None.

Return type

str

A2.6.2. Non-blocking client

The client object should not be initialized directly. Instead, use the connectToServer()-function defined above.

class`Labber.LabberClient`(*callbackNetworkError*, *callbackInstrError*, *sAddress='localhost'*, *port=None*, *timeout=None*, *callbackOpen=None*, *callbackMessage=None*, *parent=None*, *convert_to_unicode_if_py2=True*, *binary_transfer_format=None*)

Bases: `PyQt5.QtCore.QObject`

Labber client, non-blocking version.

Parameters

- **callbackNetworkError** (*function*) – Callback function called in case of network error. The function should take a single argument that will contain the error message.
- **callbackInstrError** (*function*) – Callback function called in case of instrument error. The function should take a single argument that will contain the error message.
- **sAddress** (*str*, *optional*) – IP address of Labber Instrument server. Default is localhost.
- **port** (*int*, *optional*) – Port number for server communication. Default is 9406.
- **timeout** (*int or float*, *optional*) – Longest time to wait for the server to reply. Default is 10 seconds.
- **callbackOpen** (*function*, *optional*) – Callback function called after communication has been established. The function should have a single boolean argument, which will state if the connection was successful or not.
- **callbackMessage** (*function*, *optional*) – Callback function for status updates from the server. The function should have a single string argument with the status.
- **binary_transfer_format** (*bool*, *optional*) – If True, data between the client and the server is sent as binary data instead of text. Default is True.

`close`(*bForce=False*)

Close the connection to the server.

Parameters

bForce (*bool*, *optional*) – If True, the connection is shut down without waiting for it to close. Default is False.

`connectToInstrument`(*sHardware*, *dComCfg*, *callback*, *bCreateNew=False*)

Connect to an instrument object on the instrument server.

Parameters

- **sHardware** (*str*) – Name of instrument hardware to connect to.
- **dComCfg** (*dict*) –

Dictionary describing the communication address of the instrument. Either the *name* key or the *interface*`+`*address* keys must be defined. The dictionary is defined by the following keys:

namestr

Name of instrument.

interface{‘GPIB’, ‘TCPIP’, ‘USB’, ‘Serial’, ‘VISA’, ‘Other’, ‘None’}

Communication interface.

addressstr

Instrument address string

startup{‘Set config’, ‘Get config’, ‘Do nothing’}

Operation to perform at instrument startup.

lockbool

If True, instrument will be locked while in use.

- **callback** (*function*) – Callback function called after the instruments has been created. The first argument will be an InstrumentClient object representing the instrument on the Labber instrument server.
- **bCreateNew** (*bool, optional*) – If True, a new instrument will be created if the requested one is not already present. Default is False.

`createInstrument`(*sHardware, dComCfg, callback=None*)

Create an instrument on the instrument server.

Parameters

- **sHardware** (*str*) – Name of instrument hardware to connect to.
- **dComCfg** (*dict*) –

Dictionary describing the communication address of the instrument. The dictionary is defined by the following keys:

namestr

Name of instrument.

interface{‘GPIB’, ‘TCPIP’, ‘USB’, ‘Serial’, ‘VISA’, ‘Other’, ‘None’}

Communication interface.

addressstr

Instrument address string

startup{'Set config', 'Get config', 'Do nothing'}

Operation to perform at instrument startup.

lockbool

If True, instrument will be locked while in use.

- **callback** (*function*, *optional*) – Callback function called after the instruments has been created. The first argument will be an InstrumentClient object representing the instrument on the Labber instrument server.

`firstCallbackStatus`(*callbackProgress*, *callbackCurrentValue*, *data*)

First callback occurring after the server sends back status updates. The function will handle errors and then call the next callback

`getListOfInstruments`(*callback*)

Get a list of instruments present on the Labber instrument server.

Parameters

callback (*function*) – Callback function called after the list of instruments has been retrieved. The first argument will be the list of instruments.

Returns

instruments – List of instruments on the server. Each element of the list is a two-element tuple (name, comcfg), where *name* is the hardware name and *comcfg* is a dict with communication settings.

Return type

list of tuple

`getListOfInstrumentsString`(*callback*)

Get a list of instruments present on the Labber instrument server.

Parameters

callback (*function*) – Callback function called after the list of instruments has been retrieved. The first argument will be the list of instruments.

Returns

instruments – List of strings describing instruments on the server.

Return type

list of str

A2.6.3. Instrument client

The InstrumentClient represents an instrument on the server. Note that the InstrumentClient object should not be initialized directly, but rather created using the `connectToInstrument` or `createInstrument` functions of a *LabberClient* object.

class`Labber.InstrumentClient`(*client, instrRef, ldQuant, dOption, block=True*)

The InstrumentClient is representing an instrument on the server.

`abortCurrentOperation`(*callback=None*)

Abort current operation, but keep instrument running.

Parameters

callback (*function, optional*) – Callback function called after the instruments has been aborted. Only relevant for non-blocking clients.

`arm`(*quantities, callback=None, options={}*)

Arm instrument to prepare for later hardware-triggered data acquisition

Parameters

- **quantities** (*list of str*) – Name of quantities that will be acquired when the instrument is triggered.
- **callback** (*function, optional*) – Callback function called after the instrument has been armed. Only relevant for non-blocking clients.

`disconnectFromInstr`(*callback=None*)

Disconnect from instrument.

Parameters

callback (*function, optional*) – Callback function called after the instruments has been disconnected. Only relevant for non-blocking clients.

`getInstrConfig`(*callback=None*)

Get values from the driver.

Parameters

callback (*function*, *optional*) – Callback function called after the instrument config has been retrieved. Only relevant for non-blocking clients.

Returns

values – Dictionary with instrument values. The keys are names of instrument quantities. Note that only blocking clients will return a value.

Return type

dict

`getLocalInitValuesDict`()

Get instrument values as recorded at instrument initialization.

Returns

values – Dictionary with instrument values. The dict keys are names of the instrument quantities.

Return type

dict

`getLocalOptionsDict`()

Get instrument options as recorded at instrument initialization.

Returns

options – Dictionary representing instrument options. The dictionary is defined by the following keys:

modelstr

Instrument model number/name.

optionslist of str

List of strings describing installed options.

Return type

dict

`getValue`(*sQuant*, *callback=None*, *callbackProgress=None*, *callbackCurrentValue=None*, *options={}*)

Get value of the specified quantity

Parameters

- **sQuant** (*str*) – Name of quantity to set.
- **callback** (*function*, *optional*) – Callback function called after the instrument value has been retrieved. Only relevant for non-blocking clients.
- **callbackProgress** (*function*, *optional*) – Callback function for progress updates from the server. The function must take a single argument, which will be a float between 0.0 and 1.0 indicating progress. Only relevant for non-blocking clients.
- **callbackCurrentValue** (*function*, *optional*) – Callback function for value updates from the server. The function must take a single argument (current value), and is used to show the current value during slow operations, like averaging. Only relevant for non-blocking clients.

Returns

value – Value of the instrument. Note that only blocking clients will return a value.

Return type

float, bool or numpy array.

`isRunning`(*callback=None*)

Check if instrument driver is running.

Parameters

callback (*function*, *optional*) – Callback function called after the instruments has been checked. Only relevant for non-blocking clients.

Returns

isRunning – True if instrument is running. Note that only blocking clients will return a value.

Return type

bool

`setInstrConfig`(*dValues={}*, *callback=None*, *always_update_all=True*)

Send values to the driver.

Parameters

- **dValues** (*dict*) – Dictionary with new values. The keys are names of instrument quantities.
- **callback** (*function*, *optional*) – Callback function called after the instrument config has been set. Only relevant for non-blocking clients.
- **always_update_all** (*bool*, *optional*) – If True, the instrument settings are updated even if values have not changed compared to the local settings stored in the driver.

Returns

values – Dictionary with actual values. The keys are names of instrument quantities.

Return type

dict. Note that only blocking clients will return a value

`setValue`(*sQuant*, *value*, *rate=0.0*, *wait_for_sweep=True*, *callback=None*, *callbackProgress=None*, *callbackCurrentValue=None*, *options={}*)

Set new value to the specified quantity

Parameters

- **sQuant** (*str*) – Name of quantity to set.
- **value** (*float*, *bool or numpy array*) – New value.
- **rate** (*float*, *optional*) – Sweep rate.
- **wait_for_sweep** (*bool*, *optional*) – If True and *rate* is non-zero, the instrument is waiting for a sweep to finish.
- **callback** (*function*, *optional*) – Callback function called after the instrument value has been set. Only relevant for non-blocking clients.
- **callbackProgress** (*function*, *optional*) – Callback function for progress updates from the server. The function must take a single argument, which will be a float between 0.0 and 1.0 indicating progress. Only relevant for non-blocking clients.

- **callbackCurrentValue** (*function*, *optional*) – Callback function for value updates from the server. The function must take a single argument (current value), and is used to show the current value during slow operations (sweeping). Only relevant for non-blocking clients.

Returns

value – Actual value of the instrument. Note that only blocking clients will return a value.

Return type

float, bool or numpy array.

`startInstrument`(*dOption=None*, *callback=None*)

Start the instrument.

Parameters

- **dOption** (*dict*, *optional*) –

  Dictionary representing instrument options. The dictionary is defined by the following keys:

modelstr

Instrument model number/name.

optionslist of str

List of strings describing installed options.

- **callback** (*function*, *optional*) – Callback function called after the instruments has been started. Only relevant for non-blocking clients.

`stopInstrument`(*bForceQuit=False*, *callback=None*)

Stop the instrument.

Parameters

- **bForceQuit** (*bool*, *optional*) – If True, the instrument is shut down without waiting for it to close. Default is False.
- **callback** (*function*, *optional*) – Callback function called after the instruments has been stopped. Only relevant for non-blocking clients.

`waitForSweep`(*sQuant*, *value=None*, *callback=None*, *options={}*, *callbackCurrentValue=None*)

Wait for swept instrument to reach final point or certain value.

A3. Log files

The LogFile class provides functionality for reading and writing data from Labber log files.

A3.1. Reading data from Labber

Labber log files are accessed using the *LogFile* class, which contain a number of functions for reading and writing data (see class definition below). A *LogFile* object is created by passing the path to a Labber log file as the first argument.

A3.1.1. Log information

A Labber log file contains both instrument settings and measured data, as well as metadata information from the database such as *User*, *Tags*, *Project* and *Comment*. The following example will print basic information about the log file *TestLog.hdf5*:

```python
import Labber

f = Labber.LogFile('TestLog')
print('Number of entries:', f.getNumberOfEntries())

print('Step channels:')
step_channels = f.getStepChannels()
for channel in step_channels:
    print(channel['name'])

print('Log channels:')
log_channels = f.getLogChannels()
for channel in log_channels:
    print(channel['name'])

print('User:', f.getUser())
print('Tags:', f.getTags())
print('Project:', f.getProject())
print('Comment:', f.getComment())
```

The *LogFile* class also contains functions for setting log metadata, see the class definition below.

A3.1.2. Log data

A Labber log file contains data from one or multiple *channels*. The data is organized into log *entries*, where each entry contains a one-dimensional vector of values for each channel. The entries correspond to the traces shown in the Labber *Log Viewer* program.

The *LogFile* class provides a number of functions for accessing the data, as illustrated in the example below:

```
import Labber

f = Labber.LogFile('TestLog')

# get values of all channels for a specific entry (in this case first entry)
d = f.getEntry(0)
for (channel, value) in d.items():
    print(channel, ":", value)

# get entry as x,y data, let Labber determine which channels to read
(x,y) = f.getTraceXY()

# get data for all entries for a specific channel as a 2D numpy array
data = f.getData('Voltage')

# get last recorded value of a specific channel in the measurement config
# this function also works for channels that are not step items
value = f.getChannelValue('Integration time')

# get last recorded values of all channels
# useful for extracting all instrument settings
value = f.getChannelValuesAsDict()
```

For more information on the various class methods, see the class definition below.

A3.2. Creating Labber log files

The API provides functionality for creating Labber log files that can be opened by the Labber *Log Browser* and *Log Viewer* programs. This makes it possible to add custom data to the Labber database, such as simulation results or data acquired outside of the Labber *Measurement* program.

The following lines of Python code will create a log file with sinusoid signals with different frequencies in the Labber database.

```
import Labber
import numpy as np

# create step data
vTime = np.linspace(0,1,501)
vFreq = np.linspace(1,10,100)
# define step channels
lStep = [dict(name='Time', unit='s', values=vTime),
         dict(name='Frequency', unit='Hz', values=vFreq)]
# define log channels
lLog = [dict(name='Signal', unit='V', vector=False)]
```

```
# create log file
f = Labber.createLogFile_ForData('TestSinusoid', lLog, lStep)

# add log entries
for freq in vFreq:
    data = {'Signal': np.sin(2*np.pi*freq*vTime) }
    f.addEntry(data)
```

Note that log files created with the function `createLogFile_ForData` can only be used with the *Log Browser* and *Log Viewer* programs. It is not possible to open or run such a file in the Labber *Measurement* program.

A3.3. Function definitions

`Labber.getTraceDict`*(value=[], x0=0.0, dx=1.0, x1=None, logX=False, x=None)*

Create a dict with metadata for Labber (x,y) traces.

Parameters

- **value** (*list or np.array*) – Vector of y-values for trace data.
- **x0** (*float, optional*) – x-value for first data point in trace vector. Default is 0
- **dx** (*float, optional*) – Step size for x data. If specified, the x-vector starts at “x0”, and every subsequent point is spaced by “dx”. Default is 1
- **x1** (*float, optional*) – x-value for last data point in trace vector. If specified, the “dx” parameter is ignored, and the x-vector will be a linear ramp between “x0” and “x1”.
- **logX** (*bool, optional*) – If True, the values between x0 and x1 are interpolated logarithmically. Only valid if “x0” and “x1” are specified.
- **x** (*list or np.array, optional*) – Vector of x-value to match the y-values. The input must have the same number of elements as the “values” parameter. If specified, the values of “x0”, “dx”, “x1” and “logX” are ignored.

Returns

d – Python dict with values and metadata for describing a (x,y) trace.

Return type

dict

`Labber.createLogFile_ForData`*(name, log_channels, step_channels=[], use_database=True)*

Create a log file for custom data storage in the Log database.

Parameters

- **name** (*str*) – Name or path of log file.
- **log_channels** (*list of dict*) –

  List of dict describing the log channels. The list corresponds to the log channels in the Measurement dialog. The dictionary is defined by the following keys:

namestr

Name of channel.

unitstr, optional

Unit of channel.

complexbool, optional

If True, the channel contains complex data. Default is False.

vectorbool, optional

If True, the channel contains vector data. Default is True.

x_namestr, optional

Label of the x-axis for vector data. Default is “Index”.

x_unitstr, optional

Unit of x-values for vector data.

- **step_channels** (*list of dict, optional*) –

  List of dict describing the step channels. The list corresponds to the Step sequence in the Measurement dialog. If step_values is left undefined, the resulting log file will contain a collection of traces without a uniform pre-definied dimensionality. The dictionary is defined by the following keys:

namestr

Name of channel.

values1D numpy array

Step values for step channels. The length of the vector defines the dimensionality of the data in the resulting log file.

unitstr, optional

Unit of channel.

combo_defslist of str, optional

Enumerator labels for quantity. If specified, Labber will define the channel to be of "COMBO" datatype, and the "values" data must be integer values between 0 and len(combo_defs) - 1.

- **use_database** (*bool*, *optional*) – If True, the log file is put in the central log database, otherwise the path set by the log name. Default is True.

Returns

log – LogFile object representing the newly created log.

Return type

LogFile object

Examples

Example 1: Create log without step values or fixed dimensions. Note that entries do not need to have the same length.

```
>>> import Labber
>>> lLog = [dict(name='x'), dict(name='y')]
>>> l = Labber.createLogFile_ForData('TestLog', lLog)
```

To add two entries to the log defined above:

```
>>> x = np.linspace(0,1,501)
>>> data = {'x': x, 'y': np.sin(2*np.pi*5*x) }
>>> l.addEntry(data)
>>> x = np.linspace(-1.2,1.2,201)
>>> data = {'x': x, 'y': x**2 }
>>> l.addEntry(data)
```

Example 2: Create log file using pre-defined step values. In this example, the data dimensions are defined by the step channels, and all entries need to have the same length as the first step channel. Note the the presence of `vector=False` for the *Signal* channel, which notifies that the entry size is defined by the first step channel.

```
>>> import Labber
>>> vTime = np.linspace(0,1,501)
>>> vFreq = np.linspace(1,10,10)
>>> chTime = dict(name='Time', unit='s', values=vTime)
>>> chFreq = dict(name='Frequency', unit='Hz', values=vFreq)
>>> chSig  = dict(name='Signal', unit='V', vector=False)
>>> f = Labber.createLogFile_ForData('TestData', [chSig], [chTime, chFreq])
```

To add data to the log defined above:

```
>>> for freq in vFreq:
```

```
>>>     data = {'Signal': np.sin(2*np.pi*freq*vTime)}
>>>     f.addEntry(data)
```

Example 3: Create log file using pre-defined step values, but allow individual entries to have different lengths. Compared to *Example 2* above, we use the “getTraceDict” function to define the x-values for the vector-valued data.

```
>>> import Labber
>>> import numpy as np
>>> frequencies = np.linspace(1, 10, 10)
>>> channel_f = dict(name='Frequency', unit='Hz', values=frequencies)
>>> channel_y = dict(name='Signal', unit='V', x_name='Time', x_unit='s')
>>> f = Labber.createLogFile_ForData('TestData', [channel_y], [channel_f])
```

To add data to the log defined above:

```
>>> t = np.linspace(0, 1, 501)
>>> for freq in frequencies:
>>>     y = np.sin(2 * np.pi * freq * t)
>>>     trace_dict = Labber.getTraceDict(y, x0=t[0], x1=t[-1])
>>>     data = {'Signal': trace_dict}
>>>     f.addEntry(data)
```

A3.4. LogFile class

class`Labber.LogFile`*(file_name, instrument_units=False)*

Bases: `object`

The class handles reading and writing data to and from Labber log files.

Parameters

- **file_name** (*str*) – Labber hdf5 file with log data.
- **instrument_units** (*bool*, *optional*) – If True, data from the log file is returned in instrument units instead of physical units. Default is False.

`addEntry`*(data)*

Add one entry to log file.

Parameters

data (*dict*) –

Dictionary with data. The keys should match the channel names, and the values should be 1D numpy arrays or dicts with Labber (x,y) trace data created with the “getTraceDict”-function

For scalar channels, the length of the array must match the size of the innermost step loop.

If the log contains channels with both scalar and vector data, the dict value for channels that contain vector data should be an iterable with numpy arrays or trace dicts.

`getChannelValue`(*channel_name*)

Get value of a channel at the end of the measurement.

Parameters

channel_name (*str*) – Name of channel for getting value.

Returns

value – Channel value as recorded after finishing the measurement.

Return type

float, string, or dict

`getChannelValuesAsDict`(*include_all_quantities=False*)

Get value of all channels at the end of the measurement.

Parameters

include_all_quantities (*bool*) – If False, only channels defined in the Measurement dialog are returned. Otherwise, all quantities of all instruments are included

Returns

channels – Dict with all channel values. The key is the channel name.

Return type

dict

`getComment`(*log=-1*)

Get comment from log file.

Parameters

log (*int*, *optional*) – Log number within the log file. Default is -1 (last log)

Returns

comment – String with comment

Return type

str

`getData`(*name=None*, *entry=None*, *inner=None*, *log=-1*)

Retrieve data from the log file and return it as a numpy array.

Parameters

- **name** (*str*, *optional*) – Name or index of the channel with data. If None, data for the first log channel will be returned.
- **entry** (*int or iterable*, *optional*) – Entry number within log to retrieve. If None, all elements will be returned.
- **inner** (*int or iterable*, *optional*) – Index of the inner-most loop values to retrieve. If None, all elements will be returned.
- **log** (*int*, *optional*) – Log number within the log file. Default is -1 (last log)

Returns

data – Depending on the input arguments, the output data will be a floating point number or a one- or two-dimensional numpy array.

Return type

float or np.array

`getEntry`(*entry=-1*)

Retrieve an entry from the log file and return a dict with values.

Parameters

entry (*int*, *optional*) – Entry number to retrieve, as shown in the Log Viewer. Default is -1, which will get the last trace in the file.

Returns

d – Dictionary with entry data. Keys are the channel names, the values are floats, numpy arrays or dicts with vector data. In addition, the dictionary contains a key “timestamp”, which contains a timestamp (seconds since epoch) for the entry.

Return type

dict

getFilePath(*tags*)

Get path of hdf5 log file.

Returns

path – Full path and name of hdf5 log file.

Return type

str

getLogChannels()

Get log channels in the log file.

Returns

log_channels – List of dicts representing log channels. The dictionaries contain the following keys:

namestr

Name of channel.

unitstr

Unit of channel.

complexbool

If True, the channel contains complex data.

vectorbool

If True, the channel contains vector data.

Return type

list of dict

Examples

Get list of log channels from the log file *TestLog*.

```
>>> import Labber
>>> l = Labber.LogFile('TestLog')
>>> print(l.getLogChannels())
[{'name': 'Signal', 'unit': 'V', 'complex': False, 'vector': False}]
```

`getNumberOfEntries`(*name=None*, *log=None*)

Get number of entries in the log file for the given channel.

Parameters

- **name** (*str*, *optional*) – Name of channel for data count. Default is first log channel.
- **log** (*int*, *optional*) – Log configuration number within the log file. Default is None, which will count all entries for all logs.

Returns

n – Number of entries.

Return type

int

`getNumberOfLogs`()

Get number of individual log configurations in the log file.

Returns

n – Number of log configurations.

Return type

int

`getProject`()

Get project name from log file.

Returns

project – String with project name.

Return type

str

`getStepChannels`()

Get step channels in the log file.

Returns

log_channels – List of dicts representing step channels. The dictionary contains the following keys:

namestr

Name of channel.

unitstr

Unit of channel.

values1D numpy array

Step values for step channels.

complexbool

If True, the channel contains complex data. Always False for step channels.

vectorbool

If True, the channel contains vector data. Always False for step channels.

Return type

list of dict

Examples

Get list of step channels from the log file *TestLog*

```
>>> import Labber
>>> l = Labber.LogFile('TestLog')
>>> print(l.getStepChannels())
[{'name': 'Time', 'unit': 's', 'complex': False, 'vector': False,
  'values': array([ 0.    ,  0.002,  ..., 0.998, 1.    ])},
 {'name': 'Frequency', 'unit': 'Hz', 'complex': False, 'vector': False,
  'values': array([ 1., 5., 10.])}]
```

`getTags`()

Get tag list from log file.

Returns

tags – List of strings with tags.

Return type

list of str

`getTraceXY`(*y_channel=None*, *x_channel=None*, *entry=-1*)

Retrieve a trace with (x,y) data from the log file .

Parameters

- **y_channel** (*str or int*, *optional*) – Name or log index of the channel with y-data. Default is first log channel.
- **x_channel** (*str or int*, *optional*) – Name or step index of the channel with x-data. Default is first step channel.
- **entry** (*int*, *optional*) – Entry number to retrieve, as shown in the Log Viewer. Default is -1, which will get the last trace in the file.

Returns

(x,y) – A tuple with x and y data as 1-d numpy arrays.

Return type

tuple

`getUser`()

Get user from log file.

Returns

name – String with user name.

Return type

str

`setComment`(*comment*, *log=-1*, *set_all=True*)

Set comment in log file.

Parameters

- **comment** (*str*) – String with comment.
- **log** (*int*, *optional*) – Log number within the log file. Default is -1 (last log).
- **set_all** (*bool*, *optional*) – Set comment of all log numbers within the log file. Default is True.

`setProject`(*project*)

Set project name in the log file.

Parameters

project (*str*) – String with project name.

`setTags`(*tags*)

Set list of tags in the log file.

Parameters

tags (*list of str*) – List of string with tags.

`setUser`(*name*)

Set user name in the log file.

Parameters

name (*str*) – String with user name.

A4. Script tools

The helper functions in the *ScriptTools* module are designed for repeatedly performing Measurements that each contain one-dimensional sweeps, and where one or multiple parameters of the Measurement configurations are updated between each measurement.

A4.1. Initialization

The *ScriptTools* functions call the *Measurement* executable for performing the measurements. Before the tools can be used, the path to the executable must be set using the function setExePath() defined below.

Note that in Labber version 1.1 and later, the ScriptTools module has been moved into the `Labber` module. To make scripts written for older versions of Labber work with version 1.1 and later, replace `import ScriptTools` with `from Labber import ScriptTools`.

A4.2. Example

The *ScriptTools* functions are best explained by an example, which we'll take from the domain of superconducting qubits. For the purpose of this example, we can view the qubit as a slightly anharmonic oscillators whose frequency tunes with applied magnetic flux. The qubit is read out by coupling it to microwave resonators, and the coupling is arranged in a way that changing the qubit frequency will cause a slight shift of the resonator frequency.

Now, say that we want to probe the qubit frequency as a function of applied flux. The difficulty is that the changing the flux will affect both the qubit and the resonator frequencies, which means that we cannot use a fixed-frequency read-out tone. Instead, we need to implement the following procedure:

1. **Set new magnetic flux value.**
2. **Measure resonator.**
3. **Find resonance frequency of resonator.**
4. **Measure qubit, while keeping the resonator at resonance frequency.**
5. **Repeat for all values of magnetic flux.**

The Python code below shows an example script for performing the sequence described above. The script assumes that the user has created two *Measurement* configurations, one for measuring the resonator, and one for measuring the qubit, and that both *Measurement* configurations have a single-valued step item called ’Flux bias’ that control the magnetic flux.

```
import os
import numpy as np

from Labber import ScriptTools

# define list of points
vFlux = np.linspace(-1E-3, 1E-3, 101)

# define measurement objects
sPath = os.path.dirname(os.path.abspath(__file__))
MeasResonator = ScriptTools.MeasurementObject(\
                os.path.join(sPath, 'TestResonator.hdf5'),
                os.path.join(sPath, 'TestResonatorOut.hdf5'))
MeasQubit = ScriptTools.MeasurementObject(\
            os.path.join(sPath, 'TestQubit.hdf5'),
            os.path.join(sPath, 'TestQubitOut.hdf5'))
# set the Primary channel that defines the third data dimension
MeasResonator.setPrimaryChannel('Flux bias')
MeasQubit.setPrimaryChannel('Flux bias')

# go through list of points
for n1, value_1 in enumerate(vFlux):
    print('Flux [mA]:', 1000*value_1)
    # set flux bias
    MeasResonator.updateValue('Flux bias', value_1)
    MeasQubit.updateValue('Flux bias', value_1)
    # measure resonator
    (x,y) = MeasResonator.performMeasurement()
    # for this example, y is complex, take absolute value
    y = abs(y)
    # look for peak position
    print('Resonator position [GHz]:', x[np.argmax(y)]/1E9)
    # set new frequency position
    MeasQubit.updateValue('RF - Frequency', x[np.argmax(y)])
    # measure qubit
    (x,y) = MeasQubit.performMeasurement()
```

A4.3. Function definitions

`Labber.ScriptTools.setExePath`(*path*)

Set path to the Measurement program, must be done before running scripts

Parameters

path (*str*) – Path to Measurement.exe program. On Windows, the path is typically ‘C:Program FilesLabberProgram’.

`Labber.ScriptTools.load_scenario_as_dict`*(file_name)*

Load Labber measurement scenario from binary .labber or .json file

Parameters

file_name (*str*) – Path to Labber measurement scenario file (.labber or .json).

Returns

d – Python dict describing measurement scenario.

Return type

dict

`Labber.ScriptTools.save_scenario_as_binary`*(config, file_name)*

Save Labber measurement scenario as binary .labber file

Parameters

- **config** (*dict*) – Python dict describing Labber measurement scenario.
- **file_name** (*str*) – Path to output file.

`Labber.ScriptTools.save_scenario_as_json`*(config, file_name)*

Save Labber measurement scenario as .json file

Parameters

- **config** (*dict*) – Python dict describing Labber measurement scenario.
- **file_name** (*str*) – Path to output file.

A4.4. MeasurementObject class

class`Labber.ScriptTools.MeasurementObject`*(sCfgFileIn, sCfgFileOut)*

Bases: `object`

Class for updating measurement objects and running experiments

Parameters

- **sCfgFileIn** (*str*) – Path of template config file that defines the Measurement.

- **sCfgFileOut** (*str*) – Path to output file that will be created when running the Measurement. This should typically be different from the template file, since the dimensionally of the configuration may change as data is added.

`performMeasurement`(*return_data=True*, *use_scheduler=True*)

Perform measurement and return (x,y)-tuple.

The function will start the application Measurement.exe.

Parameters

- **return_data** (*bool*, *optional*) – If True, the function will return a tuple with (x,y) data (see below). If False, the function will return the actual path of the output data file. Default is True.
- **use_scheduler** (*bool*, *optional*) – If True, the measurement will be executed using Labber's internal scheduler. If False, a separate instance of the Measurement program will be launched to run the measurment. Default is False.

Returns

(x,y) – A tuple with x and y data as 1-d numpy arrays. The x-data is taken from the first step channel, the y-data is taken from the first log channel.

Return type

tuple

`rearrangeLog`(*channel_name*, **extra_arg*)

Re-arrange a log with N entries of length M to a 2D log with dim (N, M)

The "channel_name" determines which data to use when defining the second dimension. It is also possible to rearrange into a multi- dimensional log by specifying multiple channels, but if so, lists of step values for each dimension need to be specified as well. For example, to rearrange a log with 6 entries into a multi-dimensional log with 3*2 entries, use: rearrangeLog("Channel 1", [1.0, 2.0, 3.0], "Channel 2", [1.0, 2.0])

Parameters

- **channel_name** (*str*) – Path to log file.

- **values** (*list of float, optional*) – Step value of channel_name. If not specified, the values will be taken from log file.

`setPrimaryChannel`(*channel_name*)

Specify the primary channel name.

Values of all other updated channels will be defined by look-up tables relative to the primary channel values.

Parameters

channel_name (*str*) – Name of master channel.

`setOutputFile`(*filename*)

Set output file when performing the measurement

Parameters

filename (*str*) – Path to output file.

`updateValue`(*channel_name, value, itemType='SINGLE'*)

Update a single value in the config file.

The values are kept track of internally until the Measurement.exe program is called.

Parameters

- **channel_name** (*str*) – Name of channel to update.
- **value** (*float*) – New value to set to channel.
- **itemType** (*str, optional*) – Step item parameter to set, must be one of { `single`, `start`, `stop`, `center`, `span`, `step`, `n_pts` }. Default is `single`.

A5. Configurations

The classes and the function in the *config* module allow Labber Measurement *scenarios* to be modified or created from scratch. In most cases, the recommended workflow is to create a template scenario in the Measurement program with all the instruments and signal connections used in the setup, then load the scenario into the API and use the functions *add_step* and *add_log* which channels to step over or log.

A5.1. Example

A Labber measurement scenario is represented by an instance of the class `Scenario`, which is described in more detail in the Scenario class section below. The `Scenario` object contains lists of `Instrument`, `Channel`, `StepItems` and log channel objects, as well as a number of settings and other configuration parameters.

We illustrate the process of creating a scenario with an example. The goal is to create a measurement that will generate a sine waveform with the *Simple Signal Generator* driver, send it to the *Signal demodulation* driver over a signal connection, then perform the demodulation and run the experiment for a few different values of the signal and demodulation frequency.

A5.1.1. Example - Full code

The code used to generate the example scenario is shown below:

```python
from Labber import Scenario
import numpy as np

# create and add instruments
s = Scenario()
instr_signal = s.add_instrument('Simple Signal Generator', name='Sine')
instr_demod = s.add_instrument('Signal Demodulation', name='Demod')

# set a few instrument settings
instr_demod.values['Use phase reference signal'] = False
instr_demod.values['Length'] = 1.0

# add signal connections between channels
s.add_connection('Sine - Signal', 'Demod - Input data')

# add step items, values can be defined with np array or keywords
s.add_step('Sine - Frequency', np.linspace(0, 10, 51))
s.add_step('Demod - Modulation frequency', start=1, stop=9, step=4)
```

```
# add log channels
s.add_log('Demod - Value')

# set metadata
s.comment = 'Comment for log'
s.tags.project = 'My project'
s.tags.user = 'John Doe'
s.tags.tags = ['Tag 1', 'Tag 2/Subtag']

# set timing info
s.wait_between = 0.01

# set log name and save to disk
s.log_name = 'Test signal demodulation'
s.save('demodulation_scenario')
```

The example will output a file *demodulation_scenario.labber*, which can then be opened in the Measurement program or executed using the ScriptTools API.

A5.1.2. Example - Detailed description

To describe the various function in more detail, we go through the example line-by-line. We start by creating an empty Scenario and printing the resulting object:

```
>>> from Labber import Scenario
>>> s = Scenario()
>>> print(s)
Scenario:
    instruments: [<Instrument>], #0 items
    channels: [<Channel>], #0 items
    step_items: [<StepItem>], #0 items
    log_channels: [<str>], #0 items
    tags: Tags:
        project:
        user:
        tags: [<str>], #0 items
    settings: Settings:
        send_in_parallel: True
        log_parallel: True
        arm_trig_mode: False
        trig_channel:
        hardware_loop: False
        limit_hardware_looping: False
        n_items_hardware_loop: 1
        update_instruments_if_unchanged: True
        only_send_signal_if_updated: True
        data_compression: 4
        logger_mode: False
    optimizer: Optimizer:
        method: Nelder-Mead
        max_evaluations: 200
        minimization_function: y[0]
        target_value: -inf
        relative_tolerance: inf
        method_settings: {}
    log_name:
    comment:
    wait_between: 0.0
    time_per_point: 0.1
    version: 1.8
```

The print statement lists the properties of the *Scenario* object. These properties fully configure the scenario, and are further described in the *Scenario* class description in the Scenario class section below.

The properties can be directly modified using standard *Python* notation. For example, the following lines will modify the log comment and the delay setting step channels and measureing log channels in a measurement.

```
>>> s.comment = 'This is a log comment'
>>> s.wait_between = 0.01
```

The first thing we want to do is to add a few instruments to the scenario. This can be done by creating an `Instrument` objects and directly setting it to the `instruments` property of the `Scenario` object. However, it is easier to use the helper function `add_instrument()`:

```
>>> instr_signal = s.add_instrument('Simple Signal Generator', name='Sine')
>>> instr_demod = s.add_instrument('Signal Demodulation', name='Demod')
```

This will create the two instruments and add them to the scenario. To modify the settings of the instruments, we directly update the `values` property of the `Instrument` object:

```
>>> instr_demod.values['Use phase reference signal'] = False
>>> instr_demod.values['Length'] = 1.0
```

Note that the key must match the instrument quantity as defined in the driver definition file. Also note that any undefined quantity values will be initiated to the default values as given in the driver definition file.

The next step is to set the signal connection between the sine waveform and the demodulation input. We do this with the helper function `add_connection()`:

```
>>> s.add_connection('Sine - Signal', 'Demod - Input data')
```

We haven't explicitly defined the channels `Sine - Signal` and `Demod - Input data` used in the signal connection above. The default name for channels follow the convention `<instrument name> - <quantity>`, but it is straightforward to change the name by retrieving a channel with the `get_channel()`-function and then changing its `name` property.

At this point, we are ready to set up the channels to step, and the log channels to measure. This is done with the `add_step()` and `add_log()` functions:

```
>>> s.add_step('Sine - Frequency', np.linspace(0, 10, 51))
>>> s.add_step('Demod - Modulation frequency', start=1, stop=9, step=4)
>>> s.add_log('Demod - Value')
```

Note that the step values can be defined either as a numpy array, or using the keywords `single`, `start`, `stop`, `step`, `n_pts` of the *StepItem* as defined in Section Scenario class below.

The final thing we need to do is to set the log name and save the scenario to disk:

```
>>> s.log_name = 'Test signal demodulation'
>>> s.save('demodulation_scenario')
```

The resulting file can then be opened in the *Measurement* program or executed using the *ScriptTools* API.

A5.2. Scenario class

The *Scenario* class contains both properties and helper functions for modifying the configuration.

A5.2.1. Labber.Scenario

class`labber.config.scenario.Scenario`*(file_name=None)*

Class representing a Labber scenario.

The class can be instantiated either as an empty scenario or by loading the Labber scenario provided in the file_name input parameter.

Parameters

- **instruments** (*Instrument*, *list of*) – Configuration of instruments in use in the scenario.
- **channels** (*Channel*, *list of*) – Channels used in the scenario.
- **step_items** (*StepItem*, *list of*) – Step items defining channels and values to step or sweep over.
- **log_channels** (*str*, *list of*) – List of channels to be measured at each step.
- **tags** (*Tags*) – Tags associated with the Labber scenario.
- **settings** (*Settings*) – Measurement settings specific to the scenario.

- **optimizer** (*Optimizer*) – Optimizer settings.
- **log_name** (*str*) – Name of log
- **comment** (*str*) – Comment for scenario
- **wait_between** (*float*) – Time to wait between setting step items and measuring log channels
- **time_per_point** (*float*) – Estimate for time per point, used to calculate duration
- **version** (*str*) – Version of Labber used to create scenario.

`__init__`(*file_name=None*)

Initialize scenario

Parameters

file_name (*str*, *optional*) – File with scenario to load, eithers in .json or .labber format.

`add_connection`(*source*, *target*)

Add signal connection between two channels in the scenario.

Parameters

- **source** (*str or Channel*) – Source channel for connection.
- **target** (*str or Channel*) – Target channel for connection.

`add_instrument`(*driver_name*, ***kwargs*)

Add instrument to scenario.

Optional keyword arguments are passed on to the Communication object constructor.

Parameters

driver_name (*str*) – Name of driver, must match name in driver database.

Returns

Newly created instrument.

Return type

Instrument

`add_log`(*channel*, *index=None*)

Add log item to scenario.

Parameters

- **channel** (*str och Channel*) – Channel for log item. The channel doesn't need to be defined.
- **index** (*int*) – Index of new log item in list. If not given, item is added to end.

`add_step`(*channel*, *values=None*, *index=None*, ***kwargs*)

Add step item to scenario.

If the parameter 'values' is not given, additional keywords arguments can be used to initialize the range defining the step item.

Parameters

- **channel** (*str och Channel*) – Channel for step item. The channel doesn't need to be defined.
- **values** (*numpy array*, *list of float*, *or float*) – Values for step item.
- **index** (*int*) – Index of new step item in list. If not given, item is added to end.

Returns

Newly create step item

Return type

StepItem

`channel_names`()

Get list of channels added to the scenario.

The function only returns channels that are active or have been manually added to the configuration. An active channel is used as a step item, log item, or used in a signal connection.

Returns

List of channel names.

Return type

List[str]

`get_channel`(*name*)

Get channel by name.

The function can be used to retrieve both active channels and unnamed channels that have not yet been added to the scenario.

For unnamed channels, the name must be of the format “Instrument name - Quantity”. If the instrument/quantity combination is present in the configuration, a new channel will be created and automatically added to the scenario.

Parameters

name (*str*) – Name of channel.

Returns

Channel from scenario.

Return type

Channel

`get_config_as_dict`()

Create a dict containing the scenario configuration.

Returns

Configuration of scenario.

Return type

dict

`get_instrument`(*name*)

Get instrument by name.

Parameters

name (*str*) – Name of instrument to retrieve.

Returns

Instrument from scenario.

Return type

Instrument

`get_step`(*name*)

Get step item by name.

Parameters

name (*str*) – Name of step item to retrieve.

Returns

Step item from scenario.

Return type

StepItem

`instrument_names`()

Get list of instruments present in scenario.

Returns

List of instrument names.

Return type

List[str]

`load`(*file_name*)

Load scenario from file.

Parameters

file_name (*str*) – File with scenario to load, eithers in .json or .labber format.

`log_names`()

Get list of channel names used as log items.

Returns

List of log names.

Return type

List[str]

`remove_channel`(*name*)

Remove channel from scenario.

Note that the function will only remove the channel - the corresponding instrument quantity will still be part of the scenario.

Parameters

name (*str*) – Name of channel to remove.

`remove_connection`(*channel*)

Remove signal connection scenario.

Parameters

channel (*str or Channel*) – Channel for which to remove connection, can be source or target.

`remove_instrument`(*name*)

Remove instrument from scenario.

Parameters

name (*str*) – Name of instrument to remove.

`remove_log`(*channel*)

Remove log channel from scenario.

Parameters

channel (*str or Channel*) – Log channel to remove.

`remove_step`(*name*)

Remove step item from scenario.

Parameters

name (*str*) – Name of step item to remove.

`save`(*file_name, save_as_json=False*)

Save Labber scenario to file, either as .labber or .json format.

Parameters

- **file_name** (*str*) – Path to output file.

- **save_as_json** (*bool, optional*) – If True, save to json if no extension is given, by default False

Returns

Final file name, with correct extension

Return type

str

`set_log_position`(*channel, index*)

Set position of log item.

Parameters

- **channel** (*str or Channel*) – Channel for log item to move
- **index** (*int*) – New position for log item in log list

`set_step_position`(*channel, index*)

Set position of step item tied to channel.

Parameters

- **channel** (*str or Channel or StepItem*) – Channel for step item to move
- **index** (*int*) – New position for step item in step list

`signal_connections`()

Get a list of signal connections active in scenario.

Returns

Signal connections, given as list of (source name, target name).

Return type

list of tuple

`step_names`()

Get list of channel names used as step items.

Returns

List of step names.

Return type

List[str]

A5.3. Scenario module

This module contains functions and classes for generating Labber scenarios.

A5.3.1. Enumerations

class`labber.config.scenario.LimitAction`

Enumeration class for actions when channel exceeds limit.

`CONTINUE`= *'Continue to next step item'*

Continue to next step item

`NOTHING`= *'Nothing'*

Do nothing

`STOP`= *'Stop, stay at current values'*

Stop, stay at current values

`STOP_RESET`= *'Stop, go to init/final configuration'*

Stop, go to init/final configuration

A5.3.2. Channel

class`labber.config.scenario.Channel`(***kwargs*)

Class representing a channel in a Labber scenario.

Parameters

- **name** (*str*) – Channel name.
- **instrument** (*str*) – Instrument used for channel.
- **quantity** (*str*) – Instrument quantity represented by channel.
- **unit_physical** (*str*) – Physical unit of channel
- **unit_instrument** (*str*) – Instrument unit of channel
- **gain** (*float*) – Channel gain, where Instr. value = (Phys. value * Gain + Offset) * Amp

- **offset** (*float*) – Channel offset, where Instr. value = (Phys. value * Gain + Offset) * Amp
- **amp** (*float*) – Channel amplification, where Instr. value = (Phys. value * Gain + Offset) * Amp
- **limit_high** (*float*) – High limit for channel values
- **limit_low** (*float*) – Low limit for channel values
- **limit_action** (*LimitAction*) – Action to take when log channel value exceeds limits
- **signal_source** (*str*) – Channel used as source in signal connections.

`get_config_as_dict()`

Return the configuration as a dict.

Note that the class variable `_parameter_names` define the list of attributes to include in the dict.

Return type

`dict`

`get_name()`

Get name of channel.

If no name is given, the name will be created from the instrument in the form "Instrument - Quantity".

Returns

Name of channel.

Return type

str

`set_signal_source`(*channel_source=None*)

Set channel used as source in signal connection for this channel.

Parameters

channel_source (*str or Channel*) – Channel to be set as source signal. If None, the current signal connection will be removed.

A5.3.3. Settings

class `labber.config.scenario.Settings`(***kwargs*)

Class representing the settings of a Labber scenario.

Parameters

- **send_in_parallel** (*bool*) – Send values in parallel to multiple instruments.
- **log_parallel** (*bool*) – If True, all channels are measured in parallel
- **arm_trig_mode** (*bool*) – Turn arm/trig mode on/off
- **trig_channel** (*str*) – Trig channel used in arm/trig mode
- **hardware_loop** (*bool*) – Turn hardware loop mode on/off
- **limit_hardware_looping** (*bool*) – Limit hardware looping to first step item.
- **n_items_hardware_loop** (*int*) – Number of step items in hardware loop.
- **update_instruments_if_unchanged** (*bool*) – Update instruments at start even if values are unchanged.
- **only_send_signal_if_updated** (*bool*) – Only send signal if source instrument has been updated.
- **data_compression** (*int*) – Value ranges from 0 (no compression) to 9 (max compression)
- **logger_mode** (*bool*) – If True, object represents a Logger instead of Labber scenario

A5.3.4. Optimizer

class `labber.config.scenario.Optimizer`(***kwargs*)

Class representing optimizing settings of a Labber scenario.

Parameters

- **method** (*str*) – Algorithm used for optimization.
- **max_evaluations** (*int*) – Maximum number of function evalutions/measurements before terminating.
- **minimization_function** (*str*) – Function for optimizer to minimize.
- **target_value** (*float*) – Absolute value of minimization function that will terminate optimization.
- **relative_tolerance** (*float*) – Change in value between iterations that is acceptable for convergence.
- **method_settings** (*dict*) – Specific settings for the various optimizer methods.

A5.3.5. Tags

class`labber.config.scenario.Tags`(***kwargs*)

Class representing tags of a Labber scenario.

Parameters

- **project** (*str*) – Project name associated with scenario.
- **user** (*str*) – User name associated with scenario
- **tags** (*str, list of*) – List of tags registered to the scenario.

A5.4. Instrument module

A5.4.1. Enumerations

class`labber.config.instrument.Interface`

Enumeration class for defining the communication interface.

`ASRL`= *'Serial'*

Serial - address refers to com port on computer

`GPIB`= *'GPIB'*

GPIB - specify board number in advanced settings

`NONE`= *'None'*

No instrument communcation, address is not used

`OTHER`= *'Other'*

Other - address depends on implementation

`PXI`= *'PXI'*

PXI - specify chassis number in advanced settings

`TCPIP`= *'TCPIP'*

TCPIP - address is TCIPIP address

`USB`= *'USB'*

USB - address is USB device name

`VISA`= *'VISA'*

VISA - address is full VISA resource name

class`labber.config.instrument.Parity`

Enumeration class for defining parity for serial interfaces.

`EVEN_PARITY`= *'Even parity'*

`NO_PARITY`= *'No parity'*

`ODD_PARITY`= *'Odd parity'*

class`labber.config.instrument.Startup`

Enumeration class for defining the startup operation.

`DO_NOTHING`= *'Do nothing'*

Leave instrument configuration in its current state

`GET_CONFIG`= *'Get config'*

Read configuration from instrument at start

`SET_CONFIG`= *'Set config'*

Set instrument configuration at start

class`labber.config.instrument.Termination`

Enumeration class for defining termination characters.

`AUTO`= *'Auto'*

Use system default

`CR`= *'CR'*

Carriage return

`CRLF`= *'CR+LF'*

Carriage return + line feed

`LF`= *'LF'*

Line feed

`NONE` = *'None'*

No termination

A5.4.2. Communication

class `labber.config.instrument.Communication`(***kwargs*)

Class representing Labber communication settings.

Parameters

- **name** (*str*) – Instrument name, should be unique.
- **interface** (*Interface*) – Interface type for communication.
- **address** (*str*) – Instrument address, format depends on interface type.
- **startup** (*Startup*) – Operation to perform at instrument startup.
- **server** (*str*) – IP address of server at which instrument is located.
- **lock** (*bool*) – If set, instrument is locked from other users during operation.
- **show_advanced** (*bool*) – Show/hide advanced settings in the instrument configuration window
- **timeout** (*float*) – Maximum time to wait for instrument response.
- **term_char** (*Termination*) – Termination character used by the instrument.
- **send_end_on_write** (*bool*) – Assert end during transfer of last byte of the buffer.
- **lock_visa** (*bool*) – Prevent other programs from accessing the VISA resource.
- **suppress_end_on_read** (*bool*) – Suppress end bit termination on read.
- **tcpip_specify_port** (*bool*) – Use specific TCP port.
- **tcpip_port** (*int*) – TCP port number.
- **tcpip_use_vicp** (*bool*) – Use VICP instead of TCPIP protocol for Teledyne/Lecroy instruments.
- **baud_rate** (*float*) – Communication speed for serial communication.
- **data_bits** (*float*) – Number of data bits for serial communication.
- **stop_bits** (*float*) – Number of stop bits for serial communication, can be 1, 1.5 or 2.
- **parity** (*Parity*) – Parity used for serial communication.
- **gpib_board** (*int*) – The GPIB board number enumeration starts from zero.
- **gpib_go_to_local** (*bool*) – Send GTL over GPIB after closing instrument.
- **pxi_chassis** (*int*) – PXI chassis number.
- **use_32bit_mode** (*bool*) – Run driver in 32-bit mode, for backwards compatibility.

A5.4.3. Instrument

class`labber.config.instrument.Instrument`(***kwargs*)

Class representing the configuration of a Labber instrument.

Parameters

- **hardware** (*str*) – Hardware name, must match instrument driver name.
- **model** (*str*) – Instrument model, must match a model supported by the driver.
- **options** (*str*, *list of*) – Available instrument options, must match options supported by driver.
- **com_config** (*Communication*) – Communication/interface settings of instrument.
- **values** (*dict*) – Instrument value defining the configuration.
- **version** (*str*) – Version of instrument driver for which configuration is valid.

A5.5. Step module

A5.5.1. Enumerations

class`labber.config.step.FinalAction`

Final action after finishing step

`GOTO_FIRST`= *'Goto first point'*

Goto first point

`GOTO_VALUE`= *'Goto value...'*

Goto specific value

`STAY_FINAL`= *'Stay at final'*

Stay at final

class`labber.config.step.RangeInterpolation`

Enumeration class for interpolation type of a step item.

`LINEAR`= *'Linear'*

Linear interpoloation.

`LOG`= *'Log'*

Logarithmic interpoloation.

> `LOGDECADE`= *'Log, #/decade'*

Logarithmic interpoloation, fixed number of points/decade.

> `RESONATOR`= *'Lorentzian'*

Points are calculated to be equidistant in the complex plane.

class `labber.config.step.RangeStep`

Enumeration class for step type for a step item.

> `FIXEDSTEP`= *'Fixed step'*

Set fixed step size.

> `N_PTS`= *'Fixed # of pts'*

Set fixed number of points.

class `labber.config.step.RangeType`

Enumeration class for defining the range type of a step item.

> `CENTERSPAN`= *'Center - Span'*

Center and span values.

> `SINGLE`= *'Single'*

Single value

> `STARTSTOP`= *'Start - Stop'*

Start and stop values

class `labber.config.step.StepUnit`

Define step units

> `INSTRUMENT`= *'Instrument'*

Define value in instrument units

> `PHYSICAL`= *'Physical'*

Define value in physical units

class`labber.config.step.SweepMode`

Define sweep options of step item

`BETWEEN_PTS`= *'Between points'*

Sweep between fixed points

`CONTINUOUS`= *'Continuous'*

Continuous sweeping

`NO_SWEEP`= *'Off'*

No sweeping

A5.5.2. RangeItem

class`labber.config.step.RangeItem`*(init_value=None, **kwargs)*

Class representing a single Labber step range item.

Parameters

- **range_type** (*RangeType*) – Range type, can be SINGLE, STARTSTOP, or CENTERSPAN.
- **step_type** (*RangeStep*) – Step length defintion, can be either N_PTS or FIXEDSTEP.
- **single** (*float*) – Single point value.
- **start** (*float*) – Start point of range.
- **stop** (*float*) – End point of range.
- **center** (*float*) – Center point of range.
- **span** (*float*) – Span of range.
- **step** (*float*) – Step length between points.
- **n_pts** (*int*) – Number of points in the range.
- **interp** (*RangeInterpolation*) – Interpolation type for range.
- **sweep_rate** (*float*) – Sweep rate between points in the range.

`calc_values`()

Calculate values for step item.

Returns

Values of step item

Return type

numpy array

`set_config_from_dict`(*config*)

Update config and change range type depending on given settings

Parameters

config (*dict*) – Dictionary with updated values.

`update_parameters`()

Update all parameters (start/end/center/width/etc) to be consistent.

A5.5.3. RelationParameter

class`labber.config.step.RelationParameter`(*for_step_values=False, **kwargs*)

Class representing a Labber step item relation parameter.

Parameters

- **variable** (*str*) – Parameter name.
- **channel_name** (*str*) – Name of channel represented by parameter.
- **use_lookup** (*bool*) – Turn lookup-table on/off for parameter.
- **lookup** (*LookUpTable*) – Lookup-table for parameter.

A5.5.4. OptimizerItem

class`labber.config.step.OptimizerItem`(*kwargs*)

Class representing a Labber step item optimizer config.

Parameters

- **enabled** (*bool*) – Enable/disable optimization for this step item.
- **start_value** (*float*) – Start value for optimization process.
- **init_step_size** (*float*) – First step size for optimizer.
- **min_value** (*float*) – Lowest allowed value for optimizer parameter.
- **max_value** (*float*) – Highest allowed value for optimizer parameter.

- **precision** (*float*) – Targer precision for optimizer parameter value.

A5.5.5. StepItem

class`labber.config.step.StepItem`(*channel=None*, ***kwargs*)

Class representing a Labber step config item.

Parameters

- **channel_name** (*str*) – Name of channel.
- **wait_after** (*float*) – Time (in seconds) to wait after each point.
- **final_value** (*float*) – Value to set after last point. Only relevant if after_last = GOTO_VALUE
- **show_advanced** (*bool*) – Determines if advanced step config dialog is shown by default.
- **use_relations** (*bool*) – Turns relation equation on/off.
- **equation** (*str*) – Equation setting channel relations.
- **step_unit** (*StepUnit*) – Units for step values.
- **after_last** (*FinalAction*) – Final action after finishing last step.
- **sweep_mode** (*SweepMode*) – Define sweep options of step item.
- **use_outside_sweep_rate** (*bool*) – If True, outside sweep rate is set separately from rate between points.
- **sweep_rate_outside** (*float*) – Sweep rate outside sweep range, ie before first and after last point.
- **alternate_direction** (*bool*) – If True, every other step item is executed in reverse order.
- **range_items** (*RangeItem*, *list of*) – List with range items defining step values.
- **relation_parameters** (*RelationParameter*, *list of*) – List with parameters defining relations between channels.
- **optimizer_config** (*OptimizerItem*) – Optimzier configuration for step item.

`calc_values`()

Calculate and return step values.

Note that the output is the list of values from the range items before applying any relations.

Returns

Step values, befor applying any channel relations.

Return type

np.ndarray

`update_from_values`(*values*)

Update step item with given values.

Parameters

values (*numpy array or list or float*) – New values for step item.

A5.6. Lookup module

A5.6.1. Enumerations

class`labber.config.lookup.Interpolation`

Enumeration class for defining the interpolation type.

`CUBIC`= *'Cubic'*

Cubick interpolation

`LINEAR`= *'Linear'*

Linear interpolation

`NEAREST`= *'Nearest'*

Nearest x-value

`QUADRATIC`= *'Quadratic'*

Quadratic interpolation

`ZERO`= *'Zero'*

Closest lower x-value

A5.6.2. LookUpTable

class`labber.config.lookup.LookUpTable`(*xdata=[], ydata=[], interp=<Interpolation.LINEAR: 'Linear'>*)

Class representing a Labber lookup-table.

Parameters

- **interp** (*Interpolation*) – Interpolation function, default is linear.
- **xdata** (*ndarray*) – X-data for lookup table.
- **ydata** (*ndarray*) – Y-data for lookup table.

`calc_values`(*x*)

Calculate values y(x)

Property `x_sorted`

Sorted x-data used by the interpolation function

Return type

`ndarray`

property `y_sorted`

Y-data sorted by x-values as used by the interpolation function.

Return type

`ndarray`